Reducing hardware TCB in favor of certifiable virtual machine monitor.

François Serman

PhD Thesis in Computer Sciences
Université Lille 1 - École doctorale Sciences Pour l'Ingénieur
Informatique

Reducing hardware TCB in favor of certifiable virtual machine monitor

Defended by François Serman

francois@serman.me

December 8th, 2016

Reviewed by:

Pr. Issa Traoré
University of Victoria

Pr. Gael Thomas
Telecom SudParis

Examined by:

Pr. Isabelle Ryl
INRIA Paris

Dr. Dominique Bolignano
Prove&Run SAS

Pr. Gilles Grimaud
CRIStAL

Dr. Michael Hauspie
CRIStAL

EXTRA SMALL EXTRA SAFE

Université de Lille 1 SCIENCES ET TECHNOLOGIES

PROVE & RUN

Résumé en Français

Cette thèse a pour objet la conception d'un hyperviseur logiciel sécurisé, à vocation de certification. Les plus hauts niveaux de certification requièrent l'usage de méthodes formelles, qui permettent de démontrer la validité d'un produit par rapport à une spécification à l'aide de la logique mathématique. Le matériel prouvé n'existant pas, les mécanismes d'hypervision sont ici implémentés de façon logicielle. Cela contribue à réduire la base de confiance, et donc la quantité de modélisation et de preuve à produire. En outre, cela rend possible la virtualisation de systèmes sur des plateformes qui ne sont pas dotées de ces instructions de virtualisation.

Les principaux challenges sont d'une part, l'analyse du jeu d'instruction qui, malgré l'existence de documentation, comporte des ambiguïtés et des particularités dépendantes de l'implémentation, voire même, des comportements non définis. D'autre part, d'identifier les intensions d'un système invité étant donné un flot d'instructions discret afin de rester en interposition avec le matériel sous-jacent. Pour ce faire, le code machine de l'invité est analysé et les instructions menaçant l'intégrité ou la confidentialité du système sont remplacées par des trapes logicielles, permettant d'analyser le contexte afin de décider de laisser s'exécuter l'instruction ou non.

En reposant sur l'existence d'un processeur et d'un système de gestion de mémoire prouvés, seul du code privilégié est susceptible d'outrepasser les droits d'accès configurés par l'hyperviseur. Il n'est donc pas nécessaire d'hyperviser le code non privilégié. Les micro-noyau, généralement choisis pour leur légèreté, ont donc un second avantage une fois hypervisés : ils réduisent au minimum le surcoût de l'hypervision certifiée.

Le document s'articule autour d'un état de l'art sur les différents systèmes de virtualisa-

tion d'une part, et sur les processus de certification d'autre part. Ensuite, la problématique est détaillée. La contribution principale est ensuite présentée, une proposition de design d'hyperviseur est discutée, et ses performances sont analysées. La conclusion résume ensuite les principaux points, et ouvre sur des perspectives d'optimisation.

Summary

This thesis presents the design of a secured, software based hypervisor for certification purposes. The highest levels of certification require formal methods, which demonstrate the correctness of a product with regard to its specification using mathematical logic. Proven hardware is not available off-the-shelf. In order to reduce the Trusted Computing Base (TCB) and hence, the amount of specification and proofs to produce, virtualization mechanism are software-made. In addition, this enables virtualization on platforms which do not have virtualization-enabled hardware.

The challenge for achieving this goal is twofold. On one hand, despite an existing documentation, the instruction set to be analysed has tedious corner cases, implementation-dependant behaviour or even worse, undefined behaviour. On the other hand to infer the system behaviors has to be infered given a discrete instruction flow, in order to remain interposed between the guest and the underlying hardware. For achieving this, the guest's machine code is analysed, and sensitive instructions (which threaten confidentiality or integrity) are replaced by traps, which enable arbitration given the actual guest context.

Relying on hypothetically proven processor and memory management unit, only privileged code may bypass the configuration setup by the hypervisor and access the hardware. Thus, analysing unprivileged code is worthless in this case. Micro-kernel design which tends to offload most of the code in userspace, are suitable here. Using that paradigm reduces the overhead induced by certified virtualization.

This document begins with a state-of-the-art on virtualization systems, and certification processes. It is followed the detailed problem statement. Afterward the main contribution is presented. Its design is discussed and its performances evaluated. Finally, the conclusion

gives a summary of this work, and future work is suggested.

Abstract

In computer science, virtualization has been used for decades. At the beginning, it was achieved by operating systems which shared a single machine on multiple processes. Each of those were given a "virtual adress space" of arbitrary size as if it was actually available and dedicated to them. Nowadays when we speak about virtualization, we think about running multiple instances of operating systems on a single machine. This is exactly the same principle as above. In the 90s, this job was mainly achieved using complex pieces of software. After 2000, Intel and AMD included new instructions on their CPU to ease the process of virtualization. Today, this trending topic is also rising on portable devices, such as smartphones. The last ARM processors (ARM v8) now include virtualization mechanisms. But most of the ARM processor in the wild are ARMv7; thus hardware virtualization may not be available.

We present our work on hypervisors for ARM processors. Our aim was to produce the simplest hypervisor possible, in order to formalize its security properties. Thus, we decide not to rely on hardware features:

- Those features are done with microcode, whereas the core instruction set is wired logic. Because of its nature, microcode is more error prone: it is more complex, implies machine states, which makes it harder to verify than boolean functions.
- From a formal proof point of view, we would like to be able to specify the minimum requirements for a processor to actually satisfy our requirements.
- We want to address some processors (armv7 or micro-controllers) where those instructions are not available.

It seems important to unify all those specifications and requirements of the hardware to

improve reusability of this work in each layer: hardware specification, code production (in the compiler's backend) and in the hypervisor design itself.

The first part is a state of the art of virtualization on both x86 and ARM. Certified systems are also aborded. Then, the context of the thesis is presented. Afterwards, two parts of this thesis are presented: first an hypervisor for embedded system that does not rely on hardware virtualization features, and an intensive anlysis on the ARM instruction-set analysis. Finally, future work is presented in the conclusion of the document.

Remerciements

N'étant plus à une contradiction près, j'ai décidé de rédiger ces remerciements en français.

Je tiens tout d'abord à remercier la société Prove&Run, et particulièrement Dominique Bolignano, sans qui rien de cette aventure n'aurait été possible. Ensuite, merci à mes rapporteurs Issa Traoré et Gaël Thomas. Je remercie également mes encadrants Gilles Grimaud et Michael Hauspie pour leur suivi et leurs conseils ces dernières années, y compris avant la thèse.

Merci également à vous, lecteurs/lectrices et relectrices/relecteur : Mickey, Asli, Véronique

et Kate. Une mention spéciale à Oana Andreescu, pour sa première relecture rapide, et approfondie. Mon document n'aurait pas atteint ce stade de maturité sans ta relecture. Je t'en remercie sincèrement!

J'adresse également mes plus profonds remerciements à mes collègues Lillois, qui m'ont supporté autant d'années, avec une mention spéciale à mes co-bureaux. Merci également à mes collègues parisiens, particulièrement Vincent, Olivier (mon guide spirituel du standard C), Patrice pour une conversation qui m'a servi de déclic, et à Denis pour sa contribution régulière chez Schwartz.

De façon plus personnelle, je souhaite remercier Damien Riquet pour sa bonne humeur, et ses blagues drôles (il dit qu'il voit pas le rapport). Je voudrais également remercier Pierre Graux, mon seul et unique stagiaire, qui a eu le double plaisir de travailler sur l'hyperviseur, et dans emacs. Merci à Vincent Bénony pour sa contribution au code, que j'intitulerai version 3. Ne reste plus qu'à implémenter une plateforme x86 pour fructifier ce design. Merci également pour la licence d'utilisation pour Hopper[1].

Enfin, je voudrais remercier ici mes amis pour leurs moqueries, et donc leur soutien. Et pour terminer, merci à ma famille, présente ou absente. Sur un malentendu, on va peut-être quand même réussir à en faire quelque chose.

Si j'ai oublié quelqu'un, merci d'adresser une pull-request.

Ps: the game.

—BEGIN PGP SIGNATURE—
Version: GnuPG v2

iQIcBAEBCgAGBQJX3CxIAAoJED4y673idXxPMeUP/iA0l8crfHbfrOgEpPmLi42m
M0mecVIBB0xN72xKkJ79hWaqgpc51wZWbA63fhL9iGuIVMb/uSZpS58VOKVWMwFl
vXZsfuwI0IzBa1MVsG6QWWyqFX78IYqMqfC2+fG6ziY34+quCxCadenJkvYkb4Tv
tGgRSxoPjEWRoNdqS4p/i5r5qlKVmKOo0xXu1uM1h3gYL/eht4d3g/sA2lJS2nhL
mxg0NmpVz7nJWbxj7zfBCc9wmzZYTSRMDx+6i6YQH2Zec4S2RkRrdjKtF9mFC5jb
CPPM2QVb3G5eJm/sB7LtRBtUeJGA5jyynuJAwmD/Pv5lYj30VAOt82Ax/BYXrNuV
jkFDRFtGqru2DQGft3O3uPeA01u85a8UZpXZnnFwVugXAgY3Inv4ln2Ytt65+UhC
RIsd9TGG5La3tXvB5xuEVSPGVtRvV7s/PaZ0NivV1kERMTKasTpNFbspV/bHfG8A
1vvG1PWvjYmr/VUjEtAk0eyu9FUzPkjQdRkMyoLpQRfzD2ovS/oQfStvi1pZ98FM
7vZNvL70rFTzoYA8QLk355Tk0pWyyU3UffBZIFe8fjlmWFy/z/D2f6tDe45iKyGt
UZ2NBOWKX5X7R7HSUfe0q1Lh7XMNzAPJPwlP8Y9EQyTPJ91Wh3Cf/J8/2IJzxnBb
US7mQdiLhnoiiORniq8W
=4CTb

—END PGP SIGNATURE—

[1] https://www.hopperapp.com

Table of contents

1 Introduction **15**

1.1 Context 16

1.2 Claim 16

1.3 Document structure 17

2 State of the art **19**

2.1 Lexicon 20

2.1.1 Virtualization or abstraction? 20

2.1.2 Example of virtual systems in computing 21

2.1.3 Example of abstractions in computing 22

2.2 System virtualization mechanisms 27

2.2.1 Prologue 27

2.2.2 Software approach 31

2.2.3 Hardware approach 37

2.2.4 Hybrid approaches 40
2.2.5 Summary 42
2.3 Certified systems and Common Criteria 43
2.3.1 Landscape 43
2.3.2 Common Criteria 44

3 Problem statement **55**
3.1 Hypervisors and security 56
3.1.1 Introduction 56
3.1.2 How does virtualization instructions enable security? 56
3.1.3 How do formal methods provide additional security? 58
3.1.4 Existing hypervisors with security focus 58
3.2 Software security and certified products 59
3.2.1 A market requirement 59
3.2.2 Certified products 60
3.3 Certification's impact on design 61
3.3.1 How to gain confidence in software? 62
3.3.2 Issues with end to end proof 66
3.4 A new architecture for embedded hypervisors 68
3.4.1 What is to be formalized? 68
3.4.2 Factorizing formalization: confound ISA and hardware specification 70
3.4.3 Factorizing the interpreter by executing code directly on the CPU . 71
3.5 Discussion 73

4 Contribution **75**

4.1 Prototype 76
4.1.1 Foreword: general design of the VMM 76
4.1.2 Design 76
4.1.3 Analysing guest code 77
4.1.4 Interrupt handling 81
4.1.5 Tracking privileges changes 82
4.1.6 Metrics 83
4.2 Performance overview 86
4.2.1 Benchmarks 86
4.2.2 Evaluation 86
4.2.3 Results 86
4.2.4 Analysis 88
4.3 Hypervising cost – Micro benchmarks overview 90
4.3.1 Benchmarks 90
4.3.2 Results 90
4.3.3 Discussion 90
4.4 Conclusion 92

5 In depth implementation details **93**

5.1 General concepts about the ARM architecture and instruction set 94
5.1.1 The ARM architecture overview 94
5.1.2 The ARM instruction set families 95
5.1.3 The ARM instruction set encoding 96
5.2 Analyser's implementation: the instruction scanner 100

5.2.1 Instruction behaviors 100
5.2.2 Instruction matching 100
5.3 Arbitration's implementation 104
5.3.1 Trap to the hypervisor: the callback mechanism 104
5.3.2 Tracking indirect branches 106
5.3.3 Condition validation 106
5.3.4 Conditional blocks 106
5.3.5 Summary 107
5.4 Example on a simple guest 107
5.4.1 description 107
5.5 Summary 107

6 Conclusion and future work **111**

6.1 Synthesis 112
6.2 Future work 113
6.2.1 Increasing the hypervisor performances 113
6.2.2 Reducing the context-switch cost 113
6.2.3 Expectations and foreseeable issues with a formal verification 114
6.2.4 Security assessment 116
6.3 Personal feedback 118

CHAPTER 1

Introduction

If you know the system well enough,
you can do things that aren't
supposed to be possible

Linus Torvalds

1.1 Context

My thesis was funded by Prove&Run SAS. This company was created in 2009 by Dominique Bolignano. It aims to increase security using formally proven solutions. Prove&Run has developed a toolchain to edit and prove computer programs written in Smart, their modeling language. Smart is a strongly typed, functional language. After writing Smart source-code, Prove&Run's toolchain will generate C code, which can be compiled using traditional tools such as GCC. This language was successfully used to write a specialized and secured operating system for embedded devices. It is now used to create a Trusted Execution Environement (TEE), and a hypervisor.

The academic side of my thesis was done in Lille, in 2XS team. 2XS is a member of CRIStAL, the research center in computer sciences, signal processing, and automation of the University of Lille. 2XS stands for eXtra Small, eXtra Safe. This team was created after a fork from POPS, a larger team which was involved in sensor networks and system design. 2XS has been involved in system design, while FUN has kept the WSN part. In 2XS, several axis are studied, but they often involve two parties, and thus co-design.
Between hardware and system first. Because of the embedded systems constraints, developers have to know exactly on which target their code is supposed to run. A second axis of that part, is the energy consumption. One of the short-term goals is to be able to produce an energy consumption cartography of individual blocks of code, and to make this information actually usable by a developer.
Between OS developers and Proof developers. A new topic is arising in 2XS: producing formal proofs on systems. A subset of the team is currently writing a mesovisor, which provides isolation between partitions. This mesovisor can be used to make a hypervisor, to run several systems in one single machine. After a proof of concept in C and a Haskell model, it was decided to write proofs directly in Coq, and to extract Coq into C language, in order to make it actually run.

For three years, I've been in 2XS 4 days a week, and one day at Prove&Run. During the fourth year, I have been at 2XS 2 days a week for six months.

1.2 Claim

This thesis presents a novel design to implement a hypervisor for ARM processors, suitable for the highest levels of certifications. Such levels require both a model of the product, and an implementation. Moreover, a formal proof of correctness between the model and the implementation is also required. Because no proofs on the hardware are available, we claim that the hardware mechanisms should be limited to the minimum. In particular, virtualization extensions are black boxes, which cannot be verified.

This thesis presents a proof of concept of a minimal hypervisor that limits the hardware requirements to the minimum. Only the ARM core-instruction set is used, and the CP15 co-processor for the MMU[1]. We believe that this simple design can ease the evaluation of such a product. With a proper guest, this hypervisor can achieve less than 20% overhead.

1.3 Document structure

Here is the outline of the document: First comes the state of the art. This chapter covers theoretical background of virtualization and existing solutions. Then comes a presentation of the common-criteria, an internationally recognized consortium, which evaluates the security of IT products. In the second part comes the problem statement. A tour will be made on the landscape of certified products, and their issues. Afterward I will insist on the importance of keeping a connection between each layers in the stack: hardware, compiler and sofware. The third part is an in-depth description of the hypervisor. This section also presents the results on several benchmarks, and comment those results. Finally a conclusion will sumarize the document, and perspectives will be given, mainly on how to optimize performances. I will finish with a personal feedback on those four years.

[1]Because a MMU has been successfully evaluated at the highest evaluation level, we decided that we could rely on such features

CHAPTER 2

State of the art

> The problem with engineers is that they tend to cheat in order to get results. The problem with mathematicians is that they tend to work on toy problems in order to get results. The problem with program verifiers is that they tend to cheat at toy problems in order to get results.
>
> *science jokes, ver 6.7 mar 1, 1995*
> http://lithops.as.arizona.edu/~jill/humor.text

Contents

2.1 Lexicon **20**
2.2 System virtualization mechanisms **27**
2.3 Certified systems and Common Criteria **43**

2.1 Lexicon

Because of trending topics on cloud computing, virtualization, hypervisors and so on, we will provide some definitions here. We begin with real life comparisons to give an intuition on two key concepts: virtualization and abstraction. Afterwards, we will illustrate these two concepts with computing-related examples.

2.1.1 Virtualization or abstraction?

In optics, a virtual image refers to an image of an actual object which appears to be located at the point of apparent divergence. In practice, the rays never converge, which is why a virtual image cannot be projected on screen (see Fig 2.1). A virtual object is a representation of an actual object that could exist, but does not.

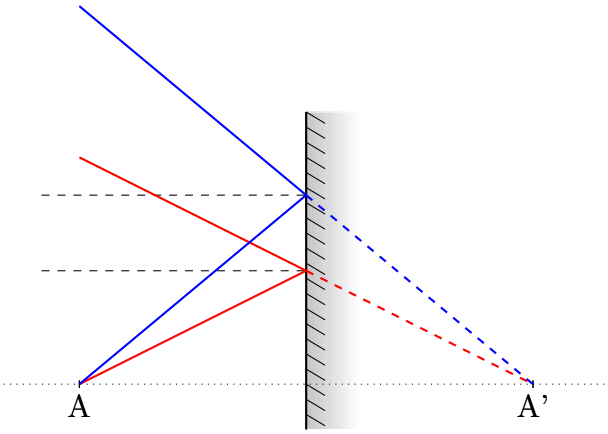


Figure 2.1: A is a source sending light through a mirror. All the rays seem to come from A', the virtual image of A. Thus, it seems that A is located behind the mirror, when in fact it is in front of it.

In real life, talking about television is an abstraction. Everybody (almost) has an object called television, which can represent slightly different objects. It is a screen with speakers that displays animated pictures and sounds. When talking about a television, we refer to it has its function, not what it is. For instance, no information is given on its size, resolution, display technology etc...

Definition 1 (Virtualization)**.** *Virtualizing an object consists of creating a virtual version of that object. A virtual object does not physically exist as such, but is made by software to appear to do so.*

Definition 2 (Abstraction)**.** *Abstraction comes from the latin verb "abstrahere" which means "draw away". An abstraction considers something as a general quality or characteristic, apart from concrete realities, specific objects, or actual instances.*

2.1.2 Example of virtual systems in computing

Computer science makes a heavy use of abstractions, and virtualization. The limit of those concepts is sometimes fuzzy, but it often corresponds to links between different layers. The following sections illustrate those concepts applied to computer hardware, and more specifically memory and microprocessor.

2.1.2.1 Virtual memory

A common usage for virtualization in computing is **virtual memory**. Tanenbaum [1] provides an overview of virtual memory. The basic idea behind virtual memory is that combined size of program, data and stack may exceed the amount of physical memory available. The operating system keeps the parts of the program that are currently in use in the main memory, and the rest on disk. This mechanism is called swapping.
Virtual memory also allows multiprogramming, that is multiple processes running at once. Of course, if only one CPU is available, only one process will run at a time. The scheduler will implement time sharing on the CPU, to make users feel like several processes run concurrently.
Another issue tackled by virtual memory is relocation. When compiling a program, no information are provided to the linker regarding usable addresses. Additionally, programs shoud run on any system without memory restriction (size or available address space). To achieve that, virtual addresses are used. Processes manipulate virtual addresses, which are not the actual addresses where data is physically stored. The latter are called physical addresses. The operating system ensures that the memory management system is configured accordingly so that virtual addresses are mapped on the physical addresses associated to the running process. Each process is free to use any virtual address, as long as there is no conflict in physical addresses[1]. The physical address conflict or swapping (in case of memory exhaustion) is handled by the operating system.

2.1.2.2 Virtual CPU

Another example is **virtual CPU**. There are several CPU emulators: Bochs [2] or Qemu [3], Valgrind [4], EM86 [5]. I will focus only on Qemu and Bochs, which are the most spread emulators.
According to Bochs' description, "Bochs is a highly portable open source IA-32 (x86) PC emulator written in C++, that runs on most popular platforms. It includes emulation of the Intel x86 CPU, common I/O devices, and a custom BIOS." Basically, the code mainly consists of a large decoding-loop which models the fetch-decode-execute actions of

[1] This can be a way to implement efficient shared memory though.

the CPU [6].
"QEMU is a FAST! processor emulator using a portable dynamic translator." [7] . It uses dynamic translation to native code for reasonable speed. When it first encounters a piece of code, QEMUconverts it to the host instruction set. Usually dynamic translators are very complicated and highly CPU dependent. QEMUuses some tricks which make it relatively easily portable and simple while achieving good performances [8].

Despite the obvious design differences (BOCHS beeing static based and QEMU dynamic), they both are an exact replica of the emulated CPU. That is, emulated code cannot make the difference between native on-cpu execution and an emulated one.

2.1.2.3 The processes case

Another example is **processes** [9]. Processes are fundamental for multiprogramming systems (also known as multitask) [10]. They are the objects manipulated by the scheduler to implement context-switching. A process is an instance of a computer program being executed. It contains an image of the associated program, and a state of the underlying hardware:

- virtual interrupts (signals);
- virtual memory;
- virtual CPU (general purpose and status registers);
- hardware access through software interruptions.

Accessing hardware through interruptions characterizes it in term of features, not as a specified piece of hardware. For instance, performing a *write* syscall makes no difference whether the data will be written on a mechanical hard drive or on a USB flash drive. This turns the process into an abstraction of the hardware rather than a virtualized representation.

Remark. *virtual addresses can also be seen as an abstraction. On x86, PAE paging translates 32-bit linear addresses to 52-bit physical addresses [11].*

2.1.3 Example of abstractions in computing

We have seen that the limit between abstraction and virtualization can be thin. In this section, we will illustrate the use of abstractions in well known areas of computing.

2.1.3.1 TCP/IP

We use TCP over IP on a daily basis for many different usages like browsing internet, sending emails, synchronize agendas etc... TCP is a `reliable` transport protocol, built on top of IP, the network layer in the OSI stack (ISO/IEC 7498-1). It provides an end-to-end service to applications running on end hosts. If you send a message over TCP, it will eventually be delivered without being altered. IP on the other hand, provides an unreliable datagram service and must be implemented by all systems addressable on the Internet. TCP deals with problems such as packet loss, duplication, and reordering that are not detected nor corrected by the IP layer. It provides a reliable flow of data between two hosts. It is concerned with things such as dividing the data passed to it from the application into appropriately sized chunks for the network layer below, acknowledging received packets, and setting timeouts to ensure that the other end acknowledges packets that are sent, and because this reliable flow of data is provided by the transport layer, the application layer can ignore all these details [12].

TCP is what computer scientists like to call an abstraction: a simplification of something much more complicated that is going on under the covers [13]. It lets the user communicate with remote or local nodes without handling complex problems such as packets fragmentation, retransmission, reordering etc... Hence, TCP provides an abstraction for a reliable communication channel.

2.1.3.2 Operating systems

Operating systems can be depicted by two unrelated functions. Namely, extending the machine and managing resources [1]. The latter is not relevant for us now, since it only ensures (fair if possible) sharing of resources among different processes. The former, on the other hand is precisely an abstraction as defined by Def. 2. Fig. 2.2 illustrates a command run in an operating system. It looks very simple and straightforward. In fact, the operating system executes many operations to perform this simple task, as shown in Fig 2.3.

Code

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
...
```

Figure 2.2: Reading the `/etc/passwd` file from command line

In this shell session, the user asks the system to display the content of the file `/etc/passwd`. The filename itself is an abstraction which exposes complex file layouts under a conveniant,

human-readable file hierarchy. The user can browse this hierarchy using a path from the root item to the file separating each node from another with a delimiter. In fact, most filesystems address files using *inode numbers*. When typing this command, several operations happen: the current shell will fork to create a new child process, then exec to replace the current (child) shell process by cat, with the parameter `/etc/passwd`. Cat will then `open (2) /etc/passwd`, `read (2)` its content, and `write (2)` it back on its standard output, and finally `close (2)` the file. That abstraction is provided by processes, as claimed in 2.1.2.3 (p 22). The operating system's abstraction is hidden behind the `read` implementation. For this example, we assume that `/etc/passwd` is actually stored on disk (and not on an NFS share for instance). As shown in Figure 2.3, there are quite many functions called by `sys_read`. The kernel provides an abstraction of the underlying filsystem (so that read can work on every filesystem). The common part is handled by the VFS layer. Afterwards, the VFS deals with memory pages, and constructs `bio` which are gathered in `requests`. Later on, those requests are submitted to the disk. Disk specificities (cache size, buffer size, mechanical or flash-based) are handled by lower level code.

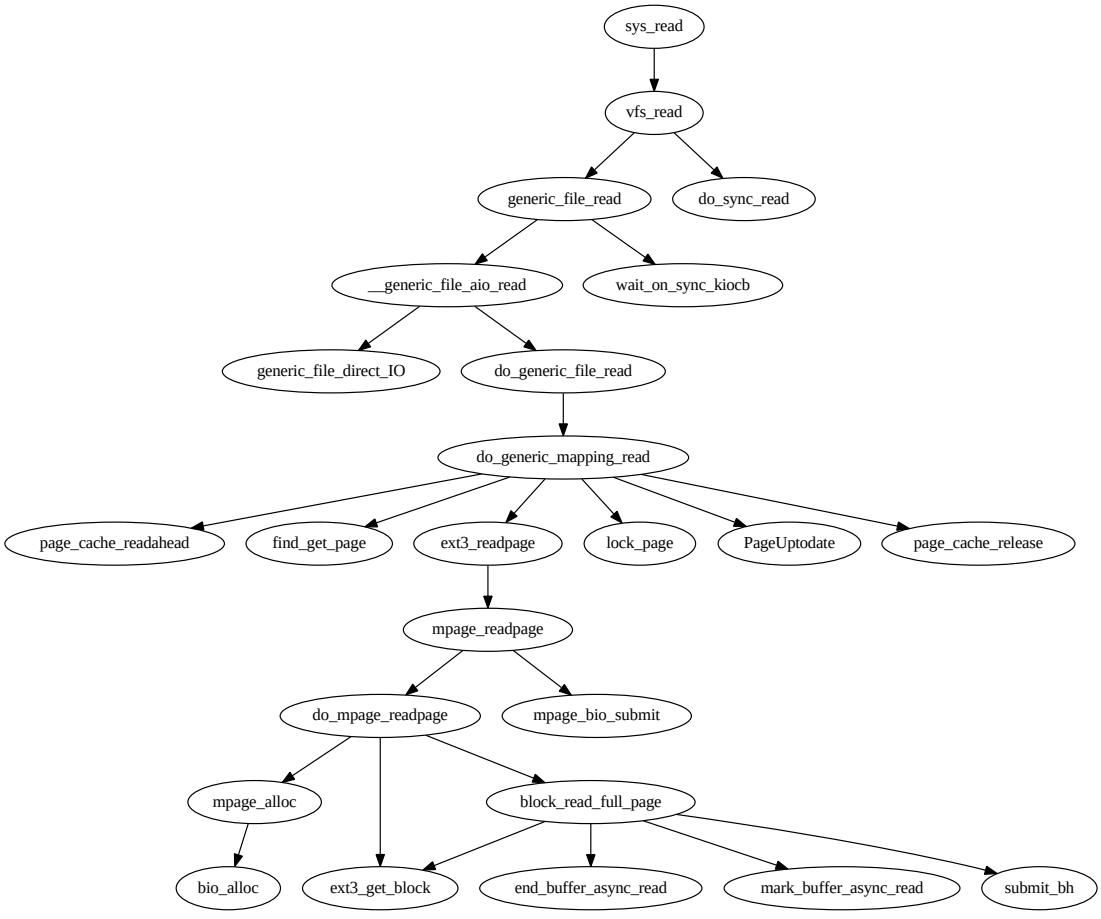


Figure 2.3: Call trace of sys_read inside the linux kernel.

2.1.3.3 Java Virtual Machine

Java is a popular programming language which uses an extra layer in form of JVM (Java Virtual Machine), which makes it platform independent. The developer writes `.java` code compiled to `.class` files, which will be executed on top of the JVM. The JVM performs the translation from byte-code to the host machine's language on the fly [14]. Because it runs on different platforms (from PDA to servers), Sun Microsystems has promoted Java with the famous slogan: "Write Once, Run Anywhere" [15]. Figure 2.4 depicts the differences between traditional platform dependent executable, and High-Level Language VM (HLL) *à la* java [16].

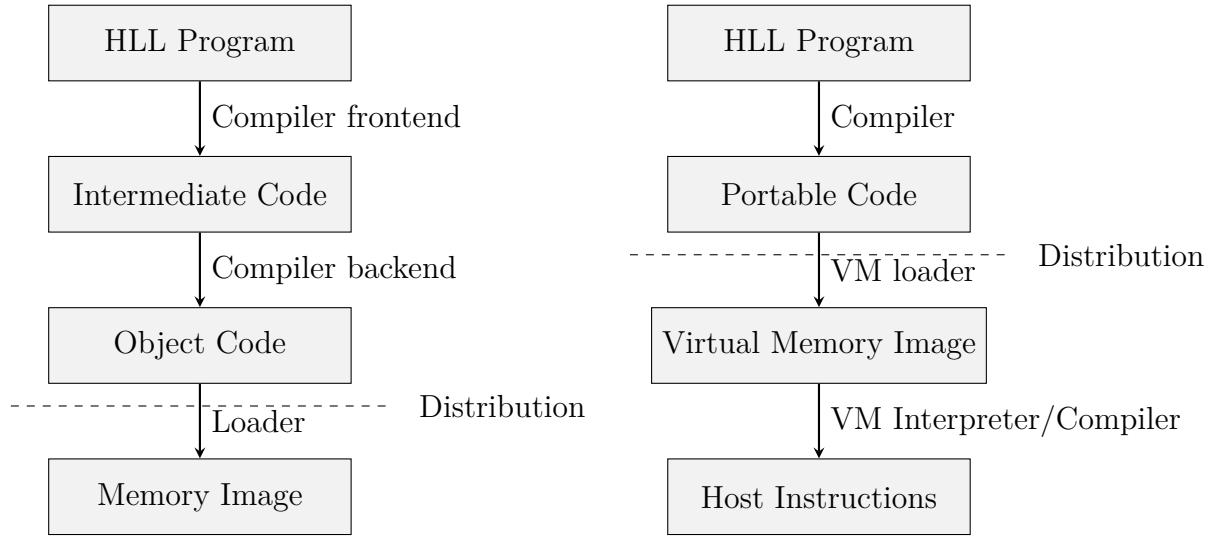


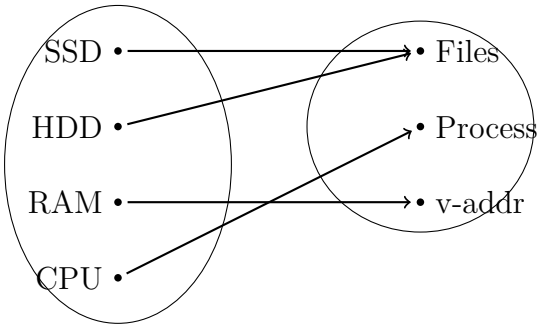
Figure 2.4: On the left, a conventional system where platform-dependent object code is distributed; on the right side a HLL VM environment where portable intermediate code is executed by a platform-dependent virtual machine.

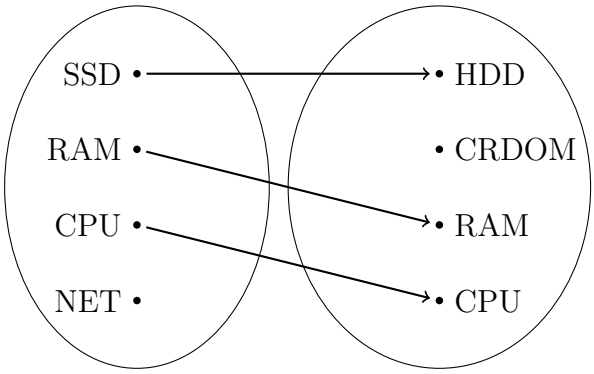
According to [17], the Java Virtual Machine is called "virtual" because it is an abstract computer defined by a specification [18]. The Java Virtual Machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time. In particular, it defines data types (which may not exist on the underlying hardware) and instructions involving higher level objects, such as *invokespecial* . Thus, besides its name, JVM is rather an abstraction.

2.1.3.4 Conclusion

Mathematical functions can be surjective (every element of the codomain is mapped to by at least one element of the domain), injective (every element of the codomain is mapped

to by at most one element of the domain) or bijective (every element of the codomain is mapped to by exactly one element of the domain). With regard to these definitions, the abstraction function would be surjective: the abstraction is a subset of the actual hardware. In the contrary, the virtualization function would be injective: virtualized domain is a superset of the hardware.





2.2 System virtualization mechanisms

Because cloud solutions are so widely spread, we use virtualization every day : either using software as a service (SAAS) such as Gmail, Dropbox or Netflix, or using the platform as a service (PAAS) like Amazon EC2 or Microsoft Azure. Cloud computing is just a marketing word which refers to on-demand delivery of IT resources and applications via the Internet with pay-as-you-go pricing[2]. However, virtualization is not a new concept in computing. In 1973, Robert P. Goldberg and Gerald J. Popek published the first articles on virtualization, which are still used to classify virtualization solutions.

In this chapter, we present an overview of virtualization, and its fundamental concepts. Both Intel and ARM will be described to have a concrete application of theoretical concepts. Afterward, we present the common criteria as a target for a security evaluation.

2.2.1 Prologue

2.2.1.1 General definitions

Nowadays, virtualization is a key concept in computer science, and let us build large and complex systems. In particular, Virtual Machine Monitor uses virtualization to present the illusion of several virtual machines (VMs), each running a separate operating system instance. An Hypervisor (or Virtual Machine Monitor) is a piece of software or hardware that creates and runs virtual machines. Hypervisors are traditionally classified into two categories after Goldberg's thesis [19]:

Type 1: Type 1 hypervisors run directly on the host's hardware. To provide security (with regard to Confidentiality / Integrity / Availability), they must control all accesses on the hardware. This tedious task is usually performed by commodity kernel. *e.g.*: Xen [20], Hyper-V [21], ESX Server [22].

Type 2: Type 2 are the most widely used hypervisors. They rely on an existing operating system (Host OS) and only need to implement virtualization mechanism. In particular, all the hardware initialization, sharing and arbitration are already performed by the Host OS. *e.g.*: KVM [23], Qemu [8], Bhyve [24], VirtualBox [25].

In their now famous paper "Formal Requirements for Virtualizable Third Generation Architectures" [26], Popek and Goldberg have presented requirements and definitions for

[2]Courtesy to Amazon (https://aws.amazon.com/what-is-cloud-computing/)

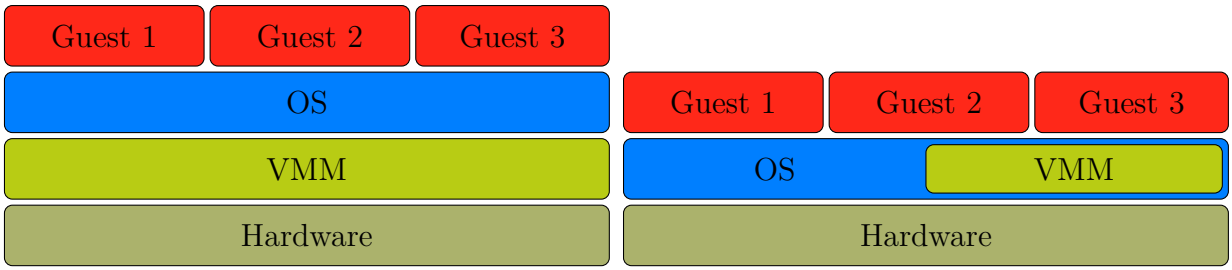


Figure 2.5: Type 1 (left) hypervisors are bare-metal software. They have a direct access to the hardware. In contrast, Type 2 (right) rely on an existing operating system.

efficient virtualization. First of all, they define a machine model defined by a processor, a memory, an instruction set, and general purpose registers. The state of such a machine can be represented by the following:

$$S = \langle E, M, P, R \rangle$$

where:

E is an executable storage ; a conventional word addressed memory of fixed size.
M is the CPU mode (**S**upervisor and **U**ser).
P is the program counter which acts as an index into E.
R is a set of registers.

The ISA is classified into 3 groups:

Privileged instructions:
: they can only run in supervisor mode. In user mode, they trap.

Control sensitive instructions:
: they attempt to change the configuration of resources in the system.

Behavior sensitive instructions:
: their results depend on the configuration of resources.

A trap can be seen as an uncatchable error thrown by the hardware. This restores control to the hypervisor, which can analyse the origin of the trap. Traps can occur when a privileged instruction is executed in user mode, or when an invalid memory access is performed.

The hypervisor should exibit three essential characteristics. First, the VMM provides an environment for programs which is essentially identical to the original machine; secondly, programs, that run in this environment, show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.

Property 1 (Equivalence/Fidelity)**.** *Any program K executing performs in a manner indistinguishable from the case when the VMM did not exist and K had whatever freedom of access to privileged instructions that the programmer had intended.*

This means that the virtualization layer should be invisible to the guest. The guest's behaviour should be the same as if it had been run on the original machine directly, with the possible exception of differences caused by the availability of system resources and differences caused by timing dependencies.

Property 2 (Efficiency)**.** *A statistically dominant subset of the virtual processor's instructions has to be executed directly by the real processor, with no software intervention by the VMM.*

In particular, all innocuous instructions are executed by the hardware directly, and do not trap.

Property 3 (Resource control)**.** *It must be impossible for that arbitrary program to affect the system resources. The VMM is said to have complete control of these resources if (1) it is not possible for a program running under it in the created environment to access any resource not explicitly allocated to it, and (2) it is possible under certain circumstances for the VMM to regain control of resources already allocated.*

With these concepts defined, Popek and Goldberk have stated a theorem that provides a sufficient condition to guarantee the virtualizability of an instruction set.

Theorem 1. *A virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions (*sic*).*

This theorem states that each instruction that could affect the correct functioning of the VMM (sensitive instructions) always traps and passes control to the VMM. It is the basis of the `trap-and-emulate` virtualization principle, which lets every innocuous instruction run natively, and whenever a sensitive instruction arrives, the VMM traps, and emulates that instruction instead of executing it on the hardware.

2.2.1.2 x86 virtualizability

According to [27], there are seventeen instructions in x86 that are sensitive but not privileged.

- SGDT, SIDT, SLDT (a GP is raised in 64bits if CR4.UMIP = 1 and CPL > 0)
- SMSW leaks PE / MP / EM / TS / ET / NE flags in bits 0-5 from CR0

- PUSHF and POPF (reverse each other's operation) PUSHF pushes lower 16 bits from the EFLAGS onto the stack and decrements the stack pointer by 2. EFLAGS contains flags that control the operating mode and state of the processor such as TF (Trap, to debug) / IF (Interrupt enable) / DF (Direction of string instructions) / and so on. Note that if an instruction is executed without enough privilege, no exception is generated, but EFLAGS are not changed either.
- LAR, LSL, VERR, VERW. LAR loads access rights from a segment descriptor into a GPR. LSL instruction loads the unscrambled segment limit from the segment descriptor into a GPR. VERR and VERW verify whether or not a code or data segment is readable or writable from the current privilege level.
- POP / PUSH (A process that thinks it is running in CPL 0 pushes the CS register to the stack. It then examines the contents of the CS register on the stack to check its CPL. Upon finding that its CPL is not 0, the process may halt.)
- CALL, JMP, INT n, RET/IRET/IRETD. Task switches and far calls to different privilege levels cause problems because they involve the CPL, DPL and RPL. A task uses a different stack for every privilege level. Therefore, when a far call is made to another privilege level, the processor switches to a stack corresponding to the new privilege level of the called procedure. Since VM normally operates at user level (CPL 3), these checks will not work correctly when a VMOS tries to access call gates or task gates at CPL 0.
- STR (This instruction prevents virtualization because it allows a task to examine its requested privilege level (RPL). This is a problem because a VM does not execute at the highest CPL or RPL (RPL = 0), but at RPL = 3. However, most operating systems assume that they are operating at the highest privilege level and that they can access any segment descriptor. Therefore, if a VM, running at a CPL and RPL of 3, uses STR to store the contents of the task register and then examines the information, it will find that it is not running at the expected privilege level.)
- MOVE (CS and SS both contain CPL in bits 0 and 1, thus a task could store the cs or ss in GPR and could examine the content of that register to find that it is not operating at the expected privileged level.

Remark

In 64 bits:

- a GP is raised in 64bits if CR4.UMIP = 1 and CPL > 0 when performing SGDT SIDT SLDT
- *GDT entries are in virtual memory. Thus, MMU may be configured to raise faults on illegal access.

Consequently x86 is not virtualizable in the sense of Goldberg and Popek.

2.2.1.3 ARM virtualizability

In this section, we do not consider the ARM virtualization extensions brought by ARMv8. This topic will be discussed in section 2.2.3.2 (p 38).

The ARM instruction set is not virtualizable. Indeed, several instructions are control or configuration sensitive, but not privileged. For instance, the following instruction (Fig. 2.6) will read the CPSR. This instruction is behaviour sensitive, but will not trap in user mode.

Code

```
mrs r0, cpsr
```

Figure 2.6: Reading CPSR into R0 in user mode won't trap.

In [28, 29, 30], the authors reference 60 instructions which break Popek and Goldbergs requirements.

- instructions accessing coprocessor's registers (reading or writing);
- instructions modifying the processor mode;
- instructions modifying the processor state (CPSR);
- instructions modifying the program counter.

Sometimes, those instructions are both control-sensitive and configuration-sensitive. This shows that ARM instruction set is not virtualizable.

2.2.2 Software approach

We have seen that both ARM and x86 instruction-set are not virtualizable out of the box. To address this issue, developers have created software which is able to analyse programs, and detect problematic instructions. Several approaches are possible, each of them having advantages and disavantages. In this section, we describe classical solutions to address software virtualization.

2.2.2.1 Software interpretation

A software interpreter is a program that reads an instruction of the source architecture one at a time, performing each operation in turn on a software-maintained version of that architecture state. Figure 2.7 depicts a pseudo code of a minimal CPU emulator. Basically, a program is stored in `Memory`. A program counter `PC` is used to track the progression. For

each instruction (`OpCode`), the emulator will perform an operation to emulate the expected behaviour. This snippet also takes a countdown before interrupt events. Such events can arise between each instructions.

First generation interpreters would simply interpret each source instruction as needed. These tend to exhibit poor performance due to the interpretation overhead.

The second generation interpreters dynamically translate source instructions into target instructions one at a time, caching the translations for later use.

The third generation interpreters, improved upon the performance of second generation, dynamically translate entire blocks of source instruction at a time [31].

2.2.2.2 Static recompilation

A binary recompiler is a software system that takes executable binaries as input, analyzes their structure, applies transformations and optimizations, and outputs new optimized executable binaries. Translated code reproduces faithfully the calling standard, implicit state, instruction side effects, branching flow, and other artifacts of the old machine. Translators can be classified as follows [32]:

Bounded translation systems: all the instructions of the old program must exist at translation time and must be found and translated to new instructions.

Open-ended translation systems: part of codes may be discovered, created or modified at execution time.

Sometimes binary translation is not just simply replacing op-codes and adjusting the order of the operands. Some factors must be considered when doing binary translation [33]:

Distinguish code and data

In assembly program, one can insert arbitrary data. In order to disassemble effectively, we have to be able to detect whether byte chunks represent code or data. Figure 2.8 depicts a real-life sample which fools the decompiler. The latter makes no difference between code and data, which leads to decode `andeq` and `undefined` instructions instead of plain data. [34] proposes an algorithm to tackle this issue:

1. taint each byte as data chunks
2. replace the data taint with code taint for each known reference (Interrupt table for instance)
3. calculate the address referenced for each previously untainted entry
4. mark that entry as data
5. based on the instruction, mark recursively other locations as instructions:
   BL (absolute) + function return jump target and next addr as instruction

(mind the SP alteration)

B (absolute) jump target as instruction

Indirect branch do nothing

Otherwise (not modifying control flow) mark the next address as an instruction

Branching The number of instructions in the target might differ from the one in the source binary. This means that the location of the routines in the translated code could be at different addresses than the original code. Because of this, it will likely be necessary to adjust the target address of some branch instructions. This is fairly easy for statically determined branches, but it can become tedious for dynamically determined branches. In the latter case, the target of the branch cannot generally be determined at translation time.

Pipelining Pipelining creates data dependencies. Code produced for the target machine must not violate these constraints even though the order or the number of instructions may be changed. This also causes issues when branches destinations are PC relative, or for any instruction breaking the sequential execution of the code.

Self modifying code Self-modifying code is usually specific to the machine for which the program was targeted. This makes it difficult to write a binary translator which can handle it. Finding self-modifying code section is also difficult.

2.2.2.3 Dynamic recompilation

The alternative to static recompilation is dynamic recompilation. Dynamic recompilation is theoretically slower than static recompilation for two main reasons:

1. translation must be performed at runtime. Translation must be as fast as possible.
2. Because of the previous reason, only a limited time can be used for optimization.

Dynamic recompilation has advantages over the static approach. In particular, it can emulate all the code in a given machine. It works like an interpreter emulating the code, and only decodes instructions that are actually executed. This helps to handle indirect jumps but also self-modifying code. Fig. 2.9 presents an implementation for dynamic recompilation:

- for efficiency, a cache is built indexed by addresses;
- for each instruction, the cache is consulted: if it misses, the cache is filed with a program supposed to emulate the current instruction;
- this program is executed.

2.2.2.4 Summary

This section gave an overview of the software technique available to perform software virtualization. We have described interpreters, static recompilers and dynamic recompilers.

The following table summarizes the advantages and disavantages of each approach:

- interpreters are platform independent and can faithfully reproduce the behavior of self-modifying programs, or programs branching to data, or using relative branches (such as `jmp r4`).
- Static recompilation provides good performances but may not be feasible.
- Dynamic recompilation is a trade-off between interpreters and static translators.

The next section will describe hardware facilities to write efficient hypervisors.

Criterion	Interpreter	Static recompilation	Dynamic recompilation
Required computation	high	negligible (offcard)	high
Overhead at runtime	high	limited	high
Repetitive executions	high	good	good (cache)
Platform independence	good	requires a new backend	limited
Possible optimizations	few	highly optimizable	slightly optimizable

Table 2.1: Summary of technique and performances applied to software analysis.

Code

```
Counter=InterruptPeriod;
PC=InitialPC;

void emulate(instr_t opcode)
{
  switch(opcode)
  {
    case OpCode1:
      handleOpCode1(opcode);
      break;
    case OpCode2:
    ...
  }
}

...

for(;;)
{
  OpCode=Memory[PC++];
  Counter-=Cycles[OpCode];

  emulate(OpCode);

  if(Counter<=0)
  {
    /* Check for interrupts and do other */
    /* cyclic tasks here                 */
    ...
    Counter+=InterruptPeriod;
    if(ExitRequired) break;
  }
}
```

Figure 2.7: Minimal CPU emulator : http://fms.komkon.org/EMUL8/HOWTO.html

Code

```
00018280 <user_task>:
   18280:       e59f301c        ldr     r3, [pc, #28]   ; 182a4
   18284:       e59f201c        ldr     r2, [pc, #28]   ; 182a8
   18288:       e5832000        str     r2, [r3]
   1828c:       ef000000        svc     0x00000000
   18290:       e59f2014        ldr     r2, [pc, #20]   ; 182ac
   18294:       e5832000        str     r2, [r3]
   18298:       ef000000        svc     0x00000000
   1829c:       ef000003        svc     0x00000003
   182a0:       e12fff1e        bx      lr
   182a4:       000183b0        ; <UNDEFINED> instruction: 0x000183b0
   182a8:       00018340        andeq   r8, r1, r0, asr #6
   182ac:       00018360        andeq   r8, r1, r0, ror #6
```

Figure 2.8: A real-life code sample: "instructions" located at `0x182a4` `0x182a8` `0x182ac` are *not* actual instrucitons, but data used for (resp.) **ldr** at `0x18280` `0x18284` `0x18290`.

Code

```
while (!end)
{
  translatedCode = ReadTranslationCache(PC);
  if (translatedCode == NULL)
      translatedCode = translate(PC)
  (*translatedCode)();
}
```

Figure 2.9: Pure dynamic binary translation algorithm

2.2.3 Hardware approach

To help writing efficient hypervisors, hardware designers have extended their architectures to make them "virtualization aware". In this section, we consider x86_64 and ARMv7 with virtualization extensions [35].

2.2.3.1 x86 architecture

Intel and AMD implement the same features, but are called differently. Basically, the required features are the following:

Provide a "hypervisor" privilege level: Before hardware virtualization support, guest operating systems were launched deprivileged, which means having a privilege level greater than 0, so that the most privileged instructions were not executable by the guest. As we have described, some instructions were not properly handled by the hardware. Although they are mutually incompatible, both Intel VT-x (codenamed "Vanderpool") and AMD-V (codenamed "Pacifica") create a new "Ring -1" so that a guest operating system can run Ring 0 operations natively without affecting other guests or the host OS.

Hardware Page Table Virtualization: provides a hardware assist to memory virtualization, which includes the partitioning and allocation of physical memory among VMs. Memory virtualization causes VMs to see a contiguous address space, which is not actually contiguous within the underlying physical memory. The guest OS stores the mapping between virtual and physical memory addresses in page tables.
Because the guest OSs do not have native direct access to physical system memory, the VMM must perform another level of memory virtualization in order to accommodate multiple VMs simultaneously. That is, mapping must be performed within the VMM, between the physical memory and the page tables in the guest OSs. In order to accelerate this additional layer of memory virtualization, both Intel and AMD have announced technologies to provide a hardware assist. Intel's is called Extended Page Tables (EPT), and AMD's is called Nested Page Tables (NPT). These two technologies are very similar at the conceptual level.

Interrupt Virtualization: the IA-32 architecture provides a mechanism for masking external interrupts, preventing their delivery when the OS is not ready for them. A VMM will likely manage external interrupts and deny the guest operating system the ability to control interrupt masking. This will lead to frequent mask/unmask interrupts. Moreover, intercepting every guest attempt could significantly affect system performance. Even though, challenge remains when a VMM has a "virtual interrupt" to deliver to a guest. Intel VT-d Posted-Interrupts and AMD Advanced Virtual Interrupt Controller (AVIC) provide an effective hardware mechanism which causes no overhead for interrupt handling.

2.2.3.2 The ARM architecture

The ARM architecture exists in multiple revision. The latest revision is version 8. Fig. 2.10 illustrates the privileges layers of this new architecture. The architecture is separated in three profiles (where v8 stands for version 8):

the ARMv8-A "Application profile": suitable for high performance markets;
the ARMv8-R "Real-time profile": suitable for embedded applications;
the ARMv8-M "Microcontroller profile": suitable for embedded and IoT applications.

These architectures are used by different processors; for example the Cortex-A7 processor implements the ARMv7-A architecture, and the Cortex-A35 provides a full ARMv8-A support. The different within a CPU families are features, such as frequency, or number of cores.

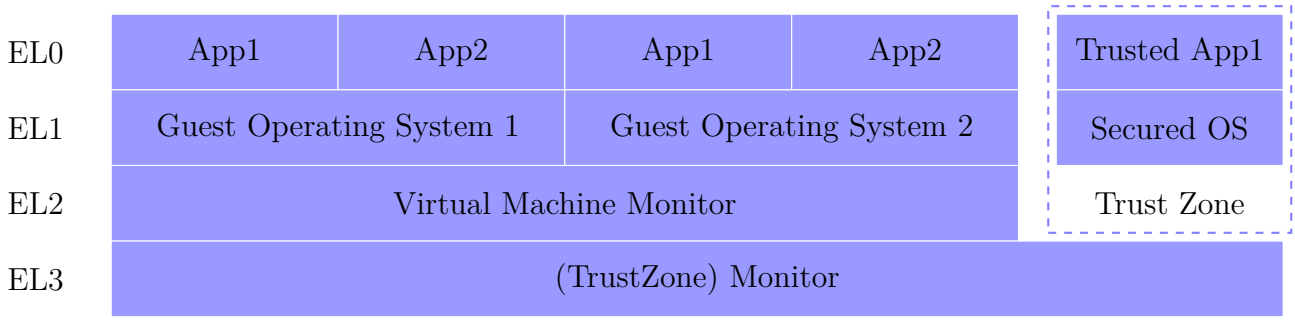


Figure 2.10: The ARM architecture's privileges levels with virtualization and trust zone features.

ARM processors may include TrustZone, a hardware facility which provides a dedicated execution context for a secure operating system (secure OS) next to a normal operating system. The instruction Secure Monitor Call (SMC) bridges the secure and normal modes. TrustZone enables protection of memory and peripherals. Applications running in secured world can access non-secure world, whereas the opposite is impossible. It offers a secure and easy-to-implement trusted computing solution for device manufacturers.

ARM processors may also include virtualization features. These features provide a new processor mode, and several features to improve performances [36, 37]:

CPU virtualization: a new processor mode (*HYP* mode) was introduced, dedicated for a VMM. Hence, this mode is more privileged than the kernel mode. To reduce the virtualization overhead, the ARM architecture allows traps to be configured in order to trap directly into a VM's kernel mode instead of going through Hyp mode.

Memory virtualization: ARM provides a hardware support for virtualization. A guest now manages Intermediate Physical Addresses (IPA, also known as guest physical addresses) which need to be translated into physical addresses (PA, or host physical addresses) by the hypervisor. TLB tags are also implemented: the TLB is now associated with a *VMID* so that TLB operations can be performed per VM instead of globally. With this new facility, shadow page table is no longer required.

Interrupts virtualization: the ARM architecture defines a GIC (Generic Interrupt Controller), which routes interrupts from devices to CPUs. CPUs use the GIC in return, to get the source of an interrupt. The GIC is separated in two parts: the distributor, and the CPU interfaces. Interrupts can be configured to trap in Hyp or kernel mode. The VMM can generate *virtual interrupts* to a guest, which will be handled transparently, as if the interrupt came from a genuine device. A trade-of must be found between trapping all the interrupts to kernel (high speed, not applicable in virtualized environment) or to the VMM (expensive solution, but the VMM retains control).

2.2.3.3 Discussion

Both ARM and x86 architecture have evolved to provide virtualization extensions to make it easier to write efficient virtual machine monitors. Despite some implementation specificities (such as guest context saving to be done by the hypervisor in ARM, whereas it is done in hardware on x86), the features are equivalent:

- a new processor mode was created. On ARM, this is a dedicated CPU execution mode, whereas on x86 it is a root/non-root mode.
- A new layer of memory virtualization was added, which lets the VMM operate an additional layer of translation. This makes shadow page-tables useless.
- Interrupts are virtualized. The VMM can inject interrupts which are handled by the guest the same way as hardware interrupts. Without this feature, the VMM must guess the interrupt entry-points, and branch at that address in the guest context.

Both architectures still require IOMMU to isolate devices (which operate on physical memory), and protect against DMA. This feature is called SMMU (System MMU) on the ARM architecture.

Currently, almost all the hypervisors rely on those hardware mechanisms. In particular, kvm and Xen do use these features on ARM and x86 architectures.

2.2.4 Hybrid approaches

Hybrid approaches can be considered when virtualization extensions are not available, or to improve performances (avoiding trap to the VMM and back). KVM [30] used to patch the Linux kernel (automatically) to replace sensitive instructions, and encode their operand. `SWI` instruction was used to trap back to the hypervisor. But because `SWI` only has 24 bits for its operand (which is not enough to encode all the parameters), a trick was used: coprocessors zero through seven are not defined by the ARM architecture, but trap regardless of their operands. That way, 24 additional bits could be used to encode additional parameters. Nowaday, KVM uses hardware virtualization features.

2.2.4.1 Paravirtualization

"Para-" is of greek origin that means "beside", "with" or "alongside". Paravirtualization (also known as OS assisted virtualization) refers to a cooperation between the guest OS and the hypervisor to improve performance and efficiency [38]. Cooperation does not mean that the guest has to be trusted, but only that it is aware not to run on bare-metal but on top of an hypervisor. This technique consists of modifying the guest source to remove non-virtualizable instructions and replace them with "hypercalls" which will delegate the privileged tasks to the hypervisor. Paravirtualization does not require any changes to the ABI (Application Binary Interface). Hence, no modification is required for guest applications.

The Xen project [39] is an example of a hypervisor which relies on paravirtualization to virtualize processor and memory using a modified kernel, and a custom protocol to virtualize I/O. The latter uses a ring buffer located in the shared memory to communicate between virtual machines. It was successfully ported on both x86 [20] and on ARM [40]. Basically, Xen must expose 3 interfaces:

Memory management: the guest OS will use hypercalls to manage virtual memory and TLB caches.

CPU: the guest OS will run in a lower privilege level than Xen, register exception handlers to Xen, implement an event system in place of interrupts, and manage a "real" and "virtual" time.

Device I/O: guest OS will use asynchronous I/O rings to transfer data toward disk or network devices.

Paravirtualization reduces the virtualization overhead because no traps back and forth to the hypervisor are required since those calls are made explicit. Its major drawback is that it cannot be used with unmodified guests. Nevertheless, paravirtualization can also be used for subsystems, such as device drivers like VirtIO [41]. VirtIO is an alternative to

Xen paravirtualized drivers. There are currently three drivers built on top of an efficient zero-copy ring buffer: a block driver, a network driver and a pci driver.

Porting an operating system on top of a paravirtualization engine:

Improves security: Because guests are deprivileged, they cannot perform critical tasks directly. Even if an unprivileged guest gets corrupted, the risk will be limited [42];

Eases writing of operating system: Exposing a high(er) level of abstraction of the machine reduces the need of low level language (formerly required for low-level operations). There exists several unikernels targeting Xen [43]: MirageOS, HalVM, ErlangOnXen, Osv, GUK ...

Still requires modification on guests: Today the total number of lines required to port Linux on Xen is about three thousands lines of code. This is estimated to be less than two percent of the x86 code base.

Removes the need of hardware extensions: At the early days of the porting of KVM for ARM, no hardware virtualization was supported. *Lightweight* paravirtualization was introduced: it is a script-based method to automatically modify the source code of the guest operating system kernel to issue calls to KVM instead of issuing sensitive instructions [30]. It is architecture specific, but operating system independent. As mentionned in 2.2.3.2 (p 38), this has been abandoned in favour of hardware mechanisms.

2.2.4.2 Trap and emulate

Trap-and-emulate was introduced by VMWare in [44] to subvert x86 virtualization issues. Suzuki and Oikawa [29] have presented their implementation on an ARMv6 CPU. Trap and emulate is somehow related to dynamic recompilation (see 2.2.2.3 (p 33)), for which the source instruction set is the same as the targeted one. Hence, only sensitive instructions must be replaced by traps to the hypervisor, which will emulate the effect of those specific instructions.

2.2.4.3 Other implementations

There are some other trends that have not been covered here, that I think worth to present.

Apple hypervisor framework: The Hypervisor framework [45] provides C APIs for interacting with virtualization technologies in user-space, without the need for writing kernel extensions. Hardware-facilitated virtual machines (VMs) and virtual processors (vCPUs) can be created and controlled by an entitled sandboxed user space process, the hypervisor client. The Hypervisor framework abstracts virtual machines

as tasks and virtual processors as threads. The framework requires x86 virtualization extensions.

BSD Jail: is a superset of chroot(2) system call. In the case of the chroot(2) call, a process' visibility of the file system name-space is limited to a single subtree. Jail facility provides a strong partitioning solution. Processes in a jail are provided full access to the files that they may manipulate, processes they may influence, and network services they can make use of. Any access to files, processes or network services outside their partition is impossible [46].

Linux containers: mainly built on top of Linux cgroup [47], a mechanism to organize processes hierarchically and distribute system resources along the hierarchy in a controlled and configurable manner. It features isolation and resource control (CPU, memory, disk I/O, network etc...) for a collection of processes, associated to namespaces. Like jails, processes confined in a namespace are bounded to their allocated resources, and cannot require or access other.

Windows Subsystem for Linux (WSL): WSL is a collection of components that enables native Linux ELF64 binaries to run on Windows. It contains both user mode and kernel mode components [48, 49]. It is composed of:

1. user mode session manager that handles the Linux instance life cycle;
2. pico provider drivers that emulate a Linux kernel by translating Linux syscalls;
3. pico processes that host the unmodified user mode Linux (*e.g.* /bin/bash).

2.2.5 Summary

As previously discussed, guests have a limited access to the resources. There are several ways for the VMM to intercept unwanted operations.

It can be transparent: the VMM interprets the guest os' intent and provides its own mechanism to meet that intent. In that case, the interception can be made by binary analysis, or assisted by hardware facilities, which makes this trap more efficient.

The guest may cooperate with the VMM, by calling the hypervisor instead of performing privileged operations. That requires source-code modification, but is suitable for performances. This principle is called paravirtualization. It is used for a full kernel (think Xen) or for dedicated drivers (think VirtIO).

2.3 Certified systems and Common Criteria

2.3.1 Landscape

Assessing information system' security is a hard task. The increase size and complexity of applications and services make it harder to stay secured and updated. To ease the evaluation of security, security standards have been writen by government agencies (for instance TCSEC (Trusted Computer System Evaluation Criteria) for United States, BS7799 for Great-Britain) or by private councils, such as PCI-DSS. PCI-DSS was founded in 2006 by American Express, Discover, JCB International, MasterCard and Visa Inc and is focused on security related to payment card processing. Some of these documents have been merged under ISO standards. In particular, ISO/IEC 27000-series focus on information security standard. It defines best practices and recommendations on information security management, and risk analysis. The information security management system preserves the confidentiality, integrity and availability. To achieve this, risk management processes are applied. It gives confidence to interested parties that risks are adequately managed.

Some of the most important items of the 270xx serie are:

ISO/IEC 27000 Information security management systems — Overview and vocabulary
ISO/IEC 27001 Information technology - Security Techniques - Information security management systems requirements.
ISO/IEC 27002 Code of practice for information security management
ISO/IEC 27003 Information security management system implementation guidance
ISO/IEC 27004 Information security management — Measurement
ISO/IEC 27005 Information security risk management
ISO/IEC 27006 Requirements for bodies providing audit and certification of information security management systems
ISO/IEC 27007 Guidelines for information security management systems auditing (focused on the management system)
ISO/IEC TR 27008 Guidance for auditors on ISMS controls (focused on the information security controls)
ISO/IEC 27010 Information security management for inter-sector and inter-organizational communications
ISO/IEC 27011 Information security management guidelines for telecommunications organizations based on ISO/IEC 27002
ISO/IEC 27013 Guideline on the integrated implementation of ISO/IEC 27001 and ISO/IEC 20000-1
ISO/IEC 27014 Information security governance.[9] Mahncke assessed this standard in the context of Australian e-health.[10]
ISO/IEC TR 27015 Information security management guidelines for financial services
ISO/IEC 27017 Code of practice for information security controls based on ISO/IEC

27002 for cloud services

ISO/IEC 27018 Code of practice for protection of personally identifiable information (PII) in public clouds acting as PII processors

ISO/IEC 27031 Guidelines for information and communication technology readiness for business continuity

ISO/IEC 27032 Guideline for cybersecurity

ISO/IEC 27033-x Network security - Part 1: Overview and concepts

ISO/IEC 27033-2 Network security - Part 2: Guidelines for the design and implementation of network security

ISO/IEC 27033-3 Network security - Part 3: Reference networking scenarios - Threats, design techniques and control issues

ISO/IEC 27033-5 Network security - Part 5: Securing communications across networks using Virtual Private Networks (VPNs)

ISO/IEC 27034 Application security - Part 1: Guideline for application security

ISO/IEC 27035 Information security incident management

ISO/IEC 27036 Information security for supplier relationships

ISO/IEC 27037 Guidelines for identification, collection, acquisition and preservation of digital evidence

ISO 27799 Information security management in health using ISO/IEC 27002. The purpose of ISO 27799 is to provide guidance to health organizations and other holders of personal health information on how to protect such information via implementation of ISO/IEC 27002.

These standards tend to define a shared terminology and methodologies among countries. This has led to the creation of ISO/IEC 15408, used as the basis for evaluation of security properties of IT products. ISO/IEC 15408 is also known as Common Criteria. In [50], Beckers et al. propose a method to evaluate security standards, applied on ISO 27001, Common Criteria and IT-Grundschutz standards. In the rest of the document, we focus on Common Criteria, because of their wide acceptance world-wide.

2.3.2 Common Criteria

2.3.2.1 General presentation

The Common Criteria (CC) is a global standard against which security products are evaluated. The Common Criteria, an internationally approved set of security standards, provides a clear and reliable evaluation of the security capabilities of Information Technology products. By providing an independent assessment of a product's ability to meet security standards, the Common Criteria gives customers more confidence in the security of Information Technology products and leads to more informed decisions. CC product certifications are mutually recognized by 25 nations, thus an evaluation that is conducted in one

country is recognized by the other countries as well [51]. The authorizing nations are: Australia, Canada, France, Germany, India, Italy, Japan, Malaysia, Netherlands, New Zealand, Norway, Republic of Korea, Spain, Sweden, Turkey, United Kingdom, United States and the consuming nations are: Austria, Czech Republic, Denmark, Finland, Greece, Hungary, Israel, Pakistan as consuming members. In France, the contact is the *National Agency for Information System's Security* (ANSSI, *Agence Nationale de la Sécurité des Systèmes d'Information*).

Despite the ISO standardization, the foundation documents are freely available on the common criteria portal [52]. There are three main documents:

Part1, introduction and general model: [53] is the introduction to the CC. It defines the general concepts and principles of IT security evaluation and presents a general model of evaluation.

Part2, security functionnal components: [54] establishes a set of functional components that serve as standard templates upon which functional requirements for TOEs (Target Of Evaluation) should be based. CC Part 2 catalogues the set of functional components and organises them in families and classes.

Part 3, security assurance components: [55] establishes a set of assurance components that serve as standard templates upon which to base assurance requirements for TOEs. CC Part 3 catalogues the set of assurance components and organises them into families and classes. CC Part 3 also defines evaluation criteria for PPs (Protection Profiles) and STs (Security targets) and presents seven pre-defined assurance packages which are called the Evaluation Assurance Levels (EALs).

These documents should be read by the interested party, which are:

consumers:
Consumers can use the results of evaluations to help decide whether a TOE fulfils their security needs. These security needs are typically identified as a result of both risk analysis and policy direction. Consumers can also use the evaluation results to compare different TOEs. The CC gives consumers, an implementation-independent structure, (Protection Profile) in which to express their security requirements in an unambiguous manner.

Developers and products vendors:
The CC is intended to support developers for both preparing and assisting the evaluation of their TOE, but also to identify the security requirements to be satisfied. Those security requirements are contained in an implementation-dependent construct named **Security Target** (ST). The security target can be based on one or more **Protection Profile** (PP) to establish that it conforms to the security requirements associated to that protection profile. The CC can be used to determine the evidence to be provided in order to support the evaluation against those requirements, but also the content and presentation of that evidence.

Evaluators and certifiers:
The CC contains criteria to be used by evaluators when forming judgements about the conformance of TOEs to their security requirements. The CC describes the set of general actions the evaluator is to carry out. The CC might specify procedures to be followed, but it is not mandatory.

To submit a product for certification, the vendor must first specify a Security Target (ST). This includes an overview of the product, security threats, detailed information on the implementation of all security features included, and a claim of conformity against protection profile (PP) at specified Evaluation Assurance Level (EAL). The number and strictness of the assurance requirements to be fulfilled depends on the Evaluation Assurance Level (EAL) [56]. Afterward, the vendor must submit the ST to an accredited testing laboratory for evaluation. The end of a successsful evaluation includes an official certification of the product against a specific protection profile at specified Evaluation Assurance Level.

2.3.2.2 Definitions

This section provides definitions, taken from the CC references (Part1 and Part3).

Definition 3.1: Target of evaluation (TOE)

The target of evaluation is the subject of an evaluation. Is a set of software, firmware and/or hardware possibly accompanied by user and administrator guidance documentation. Examples of TOEs include: software application, operating system, a software application in combination with an operating system, a cryptographic co-processor of a smart card integrated circuit, a LAN including all terminals, serveurs, network equipment and software etc... A TOE can occur in several representations: a list of files, a compiled copy, a ready to be shipped product, or an installed and operational version. Since there might exist several configuration for a TOE (in the case of an operating system, it could be the type of users, number of users, options enabled or disabled), it is often the case that the guidance part of the TOE strongly constraints the possible configuration: the guidance of the TOE may be differente from the general guidance of the product. TOE evaluation is concerned primarily with ensuring that a defined set of **security functional requirements** (SFRs) is enforced over the TOE resources.

Definition 3.2: Security Target

The security target is a set of implementation-dependent security requirements for a category of products. It is the document on which the evaluation is based, always associated to a specific TOE. It describes the assets and their associated threats. Afterward, contermeasures are described (in form of Security Objectives) and a demonstration that these contermeasures are sufficient to counter the threats is provided. The countermeasures are divided in two groups:

- Security objectives for the TOE which describe the countermeasure(s) for which correctness will be determined in the evaluation;
- Security objectives for the Operational Environment which describe the countermeasures for which correctness will not be determined in the evaluation.

To sum-up, (i) the security target demonstrates that the SFRs (Security Functional Requirements) meets the security objectives for the TOE, (ii) Which security objectives are associated to the TOE and to the environment, and (iii) the SFR and the security objectives for the operational environment counter the threats.

Definition 3.3: Security Functional Requirements

The Security Functional Requirements are a translation of the security objectives for the TOE. They are usually at a more detailed level of abstraction, but they have to be a complete translation (the security objectives must be completely addressed) and be independent of any specific technical solution (implementation). The SFRs define the rules by which the TOE governs access to and use of its resources, and thus information and services controlled by the TOE. There are eleven classes of SFR: security audit, communication, cryptographic support, user data protection, identification & authentication, security management, privacy, protection of the TOE security functions, resource utilization, TOE access, trusted path channels. Description of those classes is provided in Part 2. It can define multiple **Security Function Policies** (SFPs) to represent the rules that the TOE must enforce.

Definition 3.4: Security Function Policy

A security Function Policy is a set of rules describing specific security behaviour enforced by the TSF and expressible as a set of SFRs.

Definition 3.5: TOE Security Functionality (TSF)

All hardware, software and firmware that is necessary for the Security Functionality of the TOE.

Definition 3.6: Security Objective

Statement of an intent to counter identified threats and/or satisfy identified organisation security policies and/or assumptions.

Definition 3.7: Security Assurance Requirements

Security assurance requirements establish a set of assurance components as a standard to express the TOE assurance requirements. Consumer, developers and evaluators use the assurance requirements as guidance and for reference when determining assurance levels and requirements, assurance techniques and evaluation criteria. SAR are described in Part3, which contains nine assurance categories from which assurance requirements for a TOE can be chosen: Configuration management, delivery and operation, development, guidance documents, life cycle support, tests, vulnerability assessment, protection profile evaluation and security target evaluation.

To express security needs and facilitate writing Security Targets (ST), the CC provides two special constructs: packages and Protection Profiles (PP).

Definition 3.8: Protection Profile

A protection profile defines an implementation-independent set of security requirements and objectives for a category of TOEs, which meet similar consumer needs for IT security. Protection Profiles may be used for many different Security Targets in different evaluations.

For example, there is a protection profile for operating systems, whose goal is to describe the security functionality of operating systems in terms of CC, and to define functionnal and assurance requirements for such products.

If a protection profile is available for the product to certify, then a large part of the job is done already. Otherwise, Security Functional Requirements have to be defined to qualify the scope of the evaluation.

2.3.2.3 Evaluation Assurance Levels (EAL)

The common Criteria provides seven predefined assurance packages known as Evaluation Assurance Levels (EAL). These EALs provide an increasing scale that balances the level of assurance obtained with the cost and feasibility of acquiring that degree of assurance. The following description is based on the Part3 document.

EAL1, functionally tested: EAL1 is applicable where some confidence in correct operation is required, but the threats to security are not viewed as serious. EAL1 requires only a limited security target. It is sufficient to simply state the SFRs that the TOE must meet, rather than deriving them from threats, OSPs and assumptions through security objectives. It is intended that an EAL1 evaluation could be successfully conducted without assistance from the developer of the TOE, and for minimal outlay.

EAL2, structurally tested: EAL2 requires the co-operation of the developer in terms of the delivery of design information and test results, but should not demand more effort on the part of the developer than is consistent with good commercial practise. As such it should not require a substantially increased investment of cost or time. EAL2 is therefore applicable in those circumstances where developers or users require a low to mode rate level of independently assured security in the absence of ready availability of the complete development record.

This EAL represents a meaningful increase in assurance from EAL1 by requiring developer testing, a vulnerability analysis (in addition to the search of the public domain), and independent testing based upon more detailed TOE specifications.

EAL3, methodically tested and checked: EAL3 permits a conscientious developer to gain maximum assurance from positive security engineering at the design stage without substantial alteration of existing sound development practises. EAL3 is applicable in those circumstances where developers or users require a moderate level of independently assured security, and require a thorough investigation of the TOE and its development without substantial re-engineering.

This EAL represents a meaningful increase in assurance from EAL2 by requiring more complete testing coverage of the security functionality and mechanisms and/or procedures that provide some confidence that the TOE will not be tampered with during development.

EAL4, methodically designed, tested and reviewed: EAL4 permits a developer to gain maximum assurance from positive security engineering based on good commercial development practises which, though rigorous, do not require substantial specialist knowledge, skills, and other resources. EAL4 is the highest level at which it is likely to be economically feasible to retrofit to an existing product line. EAL4 is therefore applicable in those circumstances where developers or users require a moderate to high level of independently assured security in conventional commodity TOEs and are prepared to incur additional security-specific engineering costs.

This EAL represents a meaningful increase in assurance from EAL3 by requiring more design description, the implementation representation for the entire TSF, and improved mechanisms and/or procedures that provide confidence that the TOE will not be tampered with during development.

EAL5, semiformally designed and tested: EAL5 permits a developer to gain maximum assurance from security engineering based upon rigorous commercial development practises supported by moderate application of specialist security engineering techniques. Such a TOE will probably be designed and developed with the intent of achieving EAL5 assurance. It is likely that the additional costs attributable to the EAL5 requirements, relative to rigorous development without the application of specialised techniques, will not be large. EAL5 is therefore applicable in those circumstances where developers or users require a high level of independently assured security in a planned development and require a rigorous de velopment approach without incurring unreasonable costs attributable to specialist security engineering techniques.

This EAL represents a meaningful increase in assurance from EAL4 by requiring semiformal design descriptions, a more structured (and hence analysable) architecture, and improved mechanisms and/or procedures that provide confidence that the TOE will not be tampered with during development.

EAL6, semiformally verified design and tested: EAL6 permits developers to gain high assurance from application of security engineering techniques to a rigorous development environment in order to produce a premium TOE for protecting high value assets against significant risks. EAL6 is therefore applicable to the development of security TOEs for application in high risk situations where the value of the protected assets justifies the additional costs.

This EAL represents a meaningful increase in assurance from EAL5 by requiring more comprehensive analysis, a structured representation of the implementation, more architectural structure (*e.g.* layering), more comprehensive independent vulnerability analysis, and improved configuration management and development environment controls.

EAL7, formally verified design and tested: EAL7 is applicable to the development of security TOEs for application in extremely high risk situations and/or where the high value of the assets justifies the higher costs. Practical application of EAL7 is currently limited to TOEs with tightly focused security functionality that is amenable to extensive formal analysis.

This EAL represents a meaningful increase in assurance from EAL6 by requiring more comprehensive analysis using formal representations and formal correspondence, and comprehensive testing.

The Table 2.11 depicts the required levels for assurance families for each EAL. The highests level of certification require formal methods to establish the correctness of the model.

Assurance class	Assurance Family	Assurance Components by Evaluation Assurance Level						
		EAL1	EAL2	EAL3	EAL4	EAL5	EAL6	EAL7
Development	ADV_ARC		1	1	1	1	1	1
	ADV_FSP	1	2	3	4	5	5	6
	ADV_IMP				1	1	2	2
	ADV_INT					2	3	3
	ADV_SPM						1	1
	ADV_TDS		1	2	3	4	5	6
Guidance documents	AGD_OPE	1	1	1	1	1	1	1
	AGD_PRE	1	1	1	1	1	1	1
Life-cycle support	ALC_CMC	1	2	3	4	4	5	5
	ALC_CMS	1	2	3	4	5	5	5
	ALC_DEL		1	1	1	1	1	1
	ALC_DVS			1	1	1	2	2
	ALC_FLR							
	ALC_LCD			1	1	1	1	2
	ALC_TAT				1	2	3	3
Security Target Evaluation	ASE_CCL	1	1	1	1	1	1	1
	ASE_ECD	1	1	1	1	1	1	1
	ASE_INT	1	1	1	1	1	1	1
	ASE_OBJ	1	2	2	2	2	2	2
	ASE_REQ	1	2	2	2	2	2	2
	ASE_SPD		1	1	1	1	1	1
	ASE_TSS	1	1	1	1	1	1	1
Test	ATE_COV		1	2	2	2	3	3
	ATE_DPT			1	1	3	3	4
	ATE_FUN		1	1	1	1	2	2
	ATE_IND	1	2	2	2	2	2	3
Vulnerability assessment	AVA_VAN	1	2	2	3	4	5	5

Figure 2.11: Evaluation assurance level summary. Aiming highest levels brings highest requirements in term of assurance components.

2.3.2.4 Example

In this section, we compare the effort required for the different EAL levels for a given vulnerability assessment class. The purpose of the vulnerability assessment activity is to determine the exploitability of flaws or weaknesses in the TOE in the operational environment. It is based upon anlysis of the evaluation evidence and a search of publicly available material by the evaluator and is supported by evaluator penetration testing.

More specifically, we consider the vulnerability analysis activity (AVA_VAN). Vulnerability analysis is an assessment to determine whether potential vulnerabilities identified, during the evaluation of the development and anticipated operation of the TOE could allow attackers to violate the SFRs.

Leveling is based on an increasing rigour of vulnerability analysis by the evaluator and increased levels of attack potential required by an attacker to identify and exploit the potential vulnerabilities. There exist the following sub-activities (described in the CEM [57], the evaluation methodology used by the evaluator):

AVA_VAN.1 Vulnerability survey: The objective is to determine whether the TOE, in its operational environment, has easily identifiable exploitable vulnerabilities.

AVA_VAN.2 Vulnerability analysis: The objective is to determine whether the TOE, in its operational environment, has vulnerabilities exploitable easily by attackers possessing basic attack capability.

AVA_VAN.3 Focused vulnerability analysis: The objective is to determine whether the TOE, in its operational environment, has vulnerabilities exploitable by attackers possessing enhanced basic attack potential.

AVA_VAN.4 Methodical vulnerability analysis: The objective is to determine whether the TOE, in its operational environment, has vulnerabilities exploitable by attackers possessing moderate attack potential.

AVA_VAN.5 Advanced methodical vulnerability analysis: This one has no general guidance.

The dependance graph of the required levels for AVA_VAN for EAL 1, 3, 5 and 7 is illustrated in Figures 2.12, 2.13, 2.14 and 2.15, respectively.

2.3.2.5 Discussion

While the ISO/IEC 27k serie targets organisations, the Common Criteria provides certificates for products. These certificates provide confidence to a buyer, that the manufacturer has taken care of the security of the product. The confidence can be evaluated with the EAL. Above level 4, the evaluation is performed in white box, meaning that the evaluator

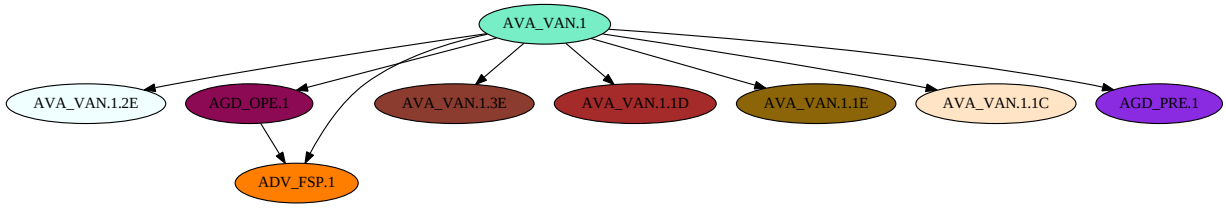


Figure 2.12: EAL 1: dependency graph.

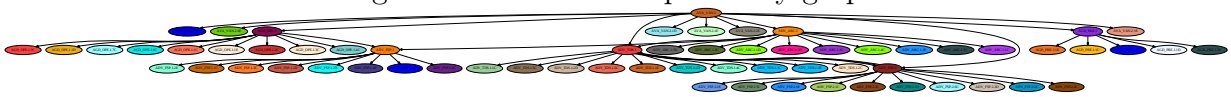


Figure 2.13: EAL3: dependency graph.

Figure 2.14: EAL5: dependency graph.

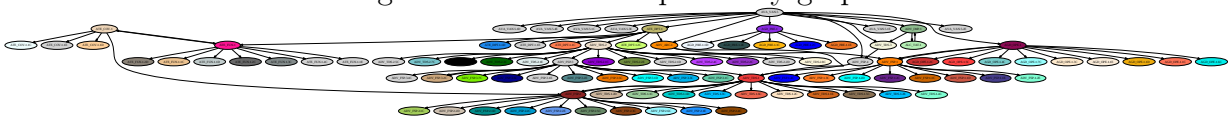


Figure 2.15: EAL7: dependency graph.

must have access to the code, and developpers may be involved. According to TÜViT[3] (an accredited laboratory to perform CC evaluations), an audit for EAL 1 takes two months, EAL 4 from five to 9 months and more than 9 months for EAL 6 and above.

Reading the Common Methodology for Information Technology Security Evaluation (targeted for the evaluator), few hints are given for the vulnerability assessment. Instead, it remains generic. From AVA_VAN.4 (which should be methodical) "This method requires the evaluator to specify the structure and form of the analysis will take". No methods are provided, so every method would be acceptable. This leads to a (potentially) poor vulnerability analysis. In contrast, MITRE (a non-for-profit organization which manages several american research centers in security) provides CAPEC, for Common Attack Pattern Enumeration and Classification. Even if CAPEC is not the panacea, the Common Criteria should have its own attack pattern collection.

In France in 2008, the ANSSI (*National Agency for Information System's Security*) has developed a first level security certification for information technology products called CSPN[4]. It is said to be an alternative for Common Criteria evaluation, when cost or evaluation duration can be an issue, and the targeted level of security is less important.

[3] https://www.tuvit.de/cps/rde/xbcr/SID-81880875-27A695D4/tuevit_de/common-criteria.pdf
[4] http://www.ssi.gouv.fr/administration/produits-certifies/cspn/

There are currently 56 products evaluated. I think it is a good thing to provide an easier evaluation, so that more actors could try and get one, which should provide more confidence in these solutions.

CHAPTER 3

Problem statement

> It is not about whether we trust the hardware or not (like if it is malicious or not) – it is whether we know how to include it in the formal model or not.
>
> *Joanna Rutkowska*

Contents

3.1 Hypervisors and security 56
3.2 Software security and certified products 59
3.3 Certification's impact on design 61
3.4 A new architecture for embedded hypervisors 68
3.5 Discussion 73

3.1 Hypervisors and security

3.1.1 Introduction

Virtualization was born in the 60's with pioneer companies such as General Electric, Bell-Labs and IBM. At that time, the only available computers were mainframes. They could only do one job at a time. So tasks had to be run in batches. The first publicly available virtualization software was CP/CMS (Control Program, Console Monitor System): a single-user operating system designed to be interactive. CP was in charge of creating virtual machines, with which user could interact.

Afterward, personal computers became affordable for the masses, and virtualization was neglected, until 1998 when a company called VMWare was created. In 1999, they began selling "VMWare workstation". VMWare Workstation lets users run operating system inside an existing one (at the beginning, only Windows was supported as a host). Virtualization was born in desktop market. Later on in 2001, they released the first version of ESX, which targets the server market. In 2003, the first releases of Xen were launched, and in 2006 Amazon launched AWS, a cloud computing service. The 2000's are definitely the decade of virtualization in the wild: in the server market, and with the rise of cloud computing. Nowaday, everybody uses virtualization on a daily basis.

The next step for virtualization is embedded systems. With the forthcoming Internet Of Things (IOT) and its needs for security will certainly lead to a secured, trustworthy hypervisor which would provide security for secrets, and isolation between partitions. This approach to enforce security from the bottom layers is easier than verifying the security of each flashed software. For example, in 2014, 12 million devices were remotely exploitable because they use an insecured version of RomPager, an embedded webserver used in network gateways [58]. Because manufacturers shipping firmware to vendors, who customize them, and because of the time to market, a lot of on-the-shelf devices are still vulnerable. Moreover, most people don't patch their internet gateway anyway.

3.1.2 How does virtualization instructions enable security?

Qubes OS is a security-oriented operating system, which takes an approach called *security by compartmentalization*. Several security domains are created (by default: work, personnal and untrusted), and applications are bound to one security domain. Each security domain is running on top of a virtual machine, so that the isolation between each context is enforced by virtualization technologies. In the project's architecture document [59], Joanna Rutkowska (Qubes OS project leader) defends Qubes' design.

"Qubes' OS architecture"

"Virtualization allows to create isolated containers, the Virtual Machines (VM). VMs can be much better isolated between each other than standard processes in monolithic kernels of popular OSes like Windows or Linux. This is because the interface between the VM and the hypervisor can be much simpler than in case of a traditional OS, and also the hypervisor itself can be much simpler (*e.g.* hypervisors, unlike typical OS kernels, do not provide many services like filesystem, networking, etc). Additionally modern computers have hardware support for virtualization (*e.g.* Intel VT-x and VT-d technology), which allows for further simplification of the hypervisor code, as well as for creating more robust system configurations, *e.g.* using so called driver domains – special hardware-isolated containers for hosting *e.g.* networking code that is normally prone to compromise."

Indeed, virtualization provides benefits in isolation capabilities [60]: In automotive systems, each application (like cruise control, brake control, etc...) is mapped onto a different Electronic Control Unit (ECU) [61]. Newest vehicules consist of up to 100 ECUs. Today's processing cores scale horizontally (increasing number of cores) rather than vertically (increasing speed). Migrating and merging several functions into a shared hardware resource decreases unnecessary redundancy, wiring, maintenance effort and reduces costs. As opposed to traditional systems which isolate real-time components onto separated processors, hypervisors can be used to mix critical systems, that is provide strong isolation (both spatial and temporal) between components executing on a single processor. Virtualization can also be suitable in domains having an existing critical (sometimes certified) code-base. Virtualization can be used to increase flexibility, interoperability and backward-compatibility. Legacy software wouldn't need to be modified, and the virtualization layer would perform critical operations to make the new hardware like an already-supported one.

Hardware designers also add security mechanisms in their processors. On x86, Intel has added SGX instructions [62, 63, 64] which enhance security features allowing developers to create secured enclaves, a dedicated area where code and data are protected against disclosure or modification. ARM provides TrustZone extension [65], which provides security features as a service, like encryption, rights management or secure storage. But this requires drivers in the client operating-system. While TrustZone can be seen as a CPU which has two halves (secured, and non-secured), SGX only has one CPU which can have many secure enclaves. Virtualization can be used to provide security (in terms of isolation of guests for instance) by interposition. It means that no cooperation from the guest is required to enforce those policies. Such solution is not achievable with hardware mechanism; hence a software approach (similar to the one defended in this thesis) should be used.

But hardware is not bug-free. In the past, Intel [66] has released a bogus CPU which was missing five entries in a lookup table used for division algorithm. Sometimes, these bugs can be exploited to breach the system's security [67]. Moreover, most of the time, hardware

is not designed with security in mind. The intrinsic design of processors make some classes of attack possible, such as side channels attacks [68]. In 2005, Bernstein [69] and Osvik et al [70] have demonstrated that processors' cache could leak memory accesses, which can be abused to recover private data, such as encryption keys (AES keys in those papers, but also part of RSA keys in [71]). Despite the isolation claimed by hypervisors, similar attacks are achievable across virtual machines running on top of the same host [72]. There are other classes of side channels attacks, for example timing attacks, branch prediction abuse, power analysis, acoustic channels, or electromagnetic attacks.

This class of attacks is tedious, but obviously achievable. We do not claim to prevent this kind of attacks with our solution. In their paper, Osvik et al. note that access control mechanisms, used to provide isolated partitionning (such as user space/kernel space, memory isolation, filesystem permissions) rely on a model of the underlying machine, which is often idealized and does not reflect many intricacies of the actual implementation.

3.1.3 How do formal methods provide additional security?

They do not. On the invisible things lab's blog (a security research company, involved in Qubes OS), Joanna Rutkowska (Qubes OS project lead) claims that formally verified microkernel (such as SeL4) do not provide end-to-end security guarantees, and would even be vulnerable to remote attacks, such as the one presented by Duflot et al [73]. She also critizises that the compiler and a large part of the hardware remains absent from the specification. Finally, such microkernel wouldn't be ready for production use, because they do not support important features such as SMP or IOMMU. Gerwin Klein (L4.verified project leader) agrees on most points, but still points out that there are two separated targets for SeL4: one verified for a restricted ARM platform targetting embedded devices (providing trusted boot, and fixed configuration), and an x86 port, where further work is being done to implement aforementioned features. Anyhow, Klein says "We also don't claim to have proven security. In fact, if you look at the slides of any of my presentations I usually make a point that we do *not* prove security, we just prove that the kernel does what is specified. If that is secure for your purposes is another story."

3.1.4 Existing hypervisors with security focus

[74] presents various hypervisors designed for real-time embedded systems. Safety is more addressed than security, but if we consider availability as a security criterion, it makes sense to mention it. In particular, ARINC 653 is a standard which addresses software architecture for spacial and temporal paritioning for safety-critical integrated modular avionics. In this context, hypervisors such as Xtratum [75] are used to enforce isolation. The same holds for automative [76], where isolation is required between entertainment and safety-critical

systems, which tend to use the same hardware for cost effectiveness.

In computers, virtualization is sometimes used to enforce security. For instance, QubeOS a "reasonably secure operating system". QubesOS is based on the Xen hypervisor, which claims to have security sensitive approach [42]. In practice, vulnerabilities on Xen do exist[1], but are publicly available.

NOVA [77] is a microhypervisor. It is a microkernel, that minimizes the trust computing base of virtual machines, by one order of magnitude. Indeed, Nova consists of the micro-hypervisor (9 KLOC), a user-level environment (7 KLOC) and the VMM (20 KLOC). In comparison with Xen that requires the VMM, a full linux kernel for dom0, and some parts of Qemu. This layer of code is evaluated to more than 400 KLOC. Our approach is comparable to Nova, except for the fact that no modern virtualization features are used. Hence, there should be no performance gain in using the latest processors available.

Finally, SeL4 [78] is a micro-kernel based on the work achieved by L4 [79]. According to their brochure[2], (SeL4) "is the only operating system that has undergone formal verification, proving bug-free implementation, and enforcement of spatial isolation (data confidentiality and integrity)." An impressive work is done on SeL4, but several points need to be mentionned:

- two versions are available (x86 and ARM), but only the latter is verified;
- a modification of the guest is required to make it run on top of SeL4 (paravirtualization);
- the hardware is trusted;
- the system initialization is assumed correct;
- the correctness of compiler, assembly code, hardware and boot code are assumed.

Our implementation is far less advanced, and not even proven. But the approach is different: we do not require any modification of the guest code, and do not assume the hardware correctness: we only used certifiable hardware features (roughly ALU and MMU).

3.2 Software security and certified products

3.2.1 A market requirement

According to Apple Inc.[3], security-conscious customers, such as the U.S. Federal Government, are increasingly requiring Common Criteria certification as a determining factor

[1] https://xenbits.xen.org/xsa/
[2] http://sel4.systems/Info/Docs/seL4-brochure.pdf
[3] https://www.apple.com/support/security/commoncriteria/CC_Whitepaper_31605.pdf

in purchasing decisions. Since the requirements for certification are clearly established, vendors can target very specific security needs while providing broad product offerings.

3.2.2 Certified products

Certifying a product is a long and expensive task. The higher the EAL is, the more expensive. The following table presents the number of certified products by categories.

Category	Certified products
Access Control Devices and Systems	67
Biometric Systems and Devices	3
Boundary Protection Devices and Systems	84
Data Protection	69
Databases	29
Detection Devices and Systems	15
ICs, Smart Cards and Smart Card-Related Devices and Systems	944
Key Management Systems	27
Multi-Function Devices	141
Network and Network-Related Devices and Systems	229
Operating Systems	96
Other Devices and Systems	274
Products for Digital Signatures	91
Trusted Computing	10

The following table presents the number of certified products by EAL, and their rates (number of products of this category over the total number of products).

EAL	1	2	3	4	5	6	7
Certified products	74	335	392	975	676	61	6
Percentage	2.94	13.30	15.56	38.71	26.84	2.42	0.24

Fort Fox Hardware Data Diode: this target is a hardware-only device that allows data to travel only in one direction. The intention of is to let information be transferred optically from a low security classified network (Low Security Level) to a higher security classified network (High Security Level), without compromising the confidentiality of the information on the High Security Level. Once manufactured, there is no way to alter the function of the TOE.

Virtual Machine of Multos M3: this target is a smart-card conceived to let several applications be loaded and executed in a secured way. The security features provided are:

- handling memory with regard to the life-cycle (opening, loading, creation, selection, deselection, exit, erasing);

- interpreting primitives and instructions called by loaded applications;
- handling interactions between loaded applications.

Only the Application Memory Mamager (MM) which handles memory belonging to applications, and providing services for loading, executing and erasing applications, and the Application Abstract Machine (AM) which interprets the instructions of the applications handled by the MM are considered.

Virtual Machine of ID Motion V1 (two of them): This virtual machine has the same security functionalities than the previous one.

Memory Management Unit for SAMSUNG micro-controllers: This target concerns three micro-controllers (S3FT9KF, S3FT9KT and S3FT9KS) which use the same MMU. The security services provided are the following:

- handling memory accesses (read/write/execute) on the memory areas of the micro-controllers;
- triggering alarms as interrupts, whenever unauthorized access arise.

Secure Optical Switch: this target is a hardware based tamper evident fibre-optic switching device that connects a single common port to any one of four selectable ports while maintaining isolation between the selectable ports within the body of the switch. There are two variants of the SOS; the local operation option has a selector dial incorporated onto the unit; while the SOS remote operation option has connectors to allow remote selection and optical feedback of the selected switch position. The security services provided are the following:

- the common port can be connected to only one selectable port at any one time;
- the selectable ports can never be connected to each other via the TOE;
- indication is provided by the TOE unequivocally confirming to which selectable port the common port is connected;
- tamper evidence of the physical case of the TOE.

3.3 Certification's impact on design

As described in 2.3, certification leverages efforts in producing high level models, tasks descriptions, documentation and extensive tests. The development becomes a rigourous and systematic process, which tends to reduce human errors and bad practices. At some point, the certification enforces traceability for each task. This implies ticketing, source control, code reviews and so on. Highest levels (above 5) also require formal methods to demonstrate the equivalence between a high level model and the actual implementation. This can become complicated with large software: if we consider the Linux code-base, there is very few (up to date) documentation available. As long as the userspace is not impacted, kernel developers don't mind refactoring API. It makes it hard for new-comers to get to know the code, and even harder for an auditor, which is not an expert in low-level programming.

3.3.1 How to gain confidence in software?

Several points of views should be considered. Informed consumer will probably read reviews on specialized press or websites. Known vulnerability database (CVE) can also be consulted. Companies managers may rely on certifications or warranties. Developers will likely read architectural documents, documentations and code (if available). Even though, such docmuents do not provide warranty on the software itself. In contrast, tests or formal verification can provide additional confidence: it's easier to read the specification or the tests cases than the whole source-code.

3.3.1.1 Testing software

In this section, we will take SQLite [80] as a case study. "Sqlite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed database engine in the world". SQLite is a heavily tested software [81]: 100% branch test coverage, millions of test cases, out-of-memory tests, fuzzing, boundary value test, regression tests and valgrind analysis. All releases of SQLite are tested with the TH3 (an aviation-grade) test harness test suite. These tests are written in C, and provide 100% branch test coverage and 100% modified condition/decision coverage (MC/DC) on the SQLite library. These tests come from avionic software guidance DO-178B and DO-178C to ensure adequate testing of critical software (critical in sense of being able to provide safe flight and landing). The following points are enforced:

- every entry points and exit in the program have been invoked at least once;
- every conditions in the program have taken all possible outcomes at least once;
- every conditions of a decision are shown to have an effect on the outcome independently.

But all those tests come at the price of an extensive codebase: SQLite consists of 120k lines of code (approximately), but the project has 91600k lines of code. That makes roughly 765 times more code. Despite the impressive results achieved by the tests, John Regehr and people from TrustInsoft managed to find issues in the codebase using a nearly automated tool: tis-interpreter [82]. Tis-interpreter is a software analysis tool based on Frama-C, designed to find subtle bugs in C programs. It works by interpreting C programs statement by statement, verifying whether each statement invokes any undefined behaviour. With that tool, several bugs were found [83] (and are now corrected). Those bugs are mainly undefined behaviour in the C standard [84] which happen to work "most of the time" (since tests are valid).

Dangling pointers: consists of using the value of a pointer after the lifetime of the pointed object.

Uses of uinitialized storage: for instance allocating a variable on the stack without initialization, and accessing it. There are no warranties on the initial value of such variable.

Out-of-bounds pointers: Some SQLite structs use 1-based array indexing. When allocating memory, the pointer tracking that memory region is decremented to make it transparent for developer to access 1-based whereas C is 0-based.

Illegal arguments: mostly on *memset* and *memcpy* which do not work properly with *NULL* pointers arguments.

Comparison of pointers to unrelated objects: comparison operators (> >= <= <) can raise undefined behaviour (see §6.5.8 of the C standard). SQLite has used some corner case comparisons which led to undefined behaviour.

This real-life example, gives some more credits to Edsger W. Dijkstra's famous quote: "testing only shows the presence of bugs, not their absence". Despite the extensive work done by SQLite developers, testing software is not sufficient to reach a high level of confidence. Formal methods are a candidate to achieve this task. Using them used to be expensive, because they were not industrialized. Companies such as Prove&Run[4] or TrustInSoft[5] were created to ease the adoption of formal methods in the industry.

3.3.1.2 Using formal methods

Formal methods consist of applying a rigourous mathematic logic on models, representing an implementation. Afterward, tools are used (automated or interractive) to produce a computer-verifiable proof of correctness, of theorems made on these models. Such methods have been successfuly used to produce operating systems, such as SeL4 [78] or ProvenCore by Prove&Run [85]. They have also been used to verify properties on hardware after the Pentium bug. But formal methods are not limited to computing scope, for instance Lille's subway has been verified using B-method.

3.3.1.2.1 Model Checking Model checking belongs to the familly of properties checking. Property checking is used for verification instead of equivalence checking. An important class of model checking methods have been developed for checking models of hardware and software designs where the specification is given by a temporal logic formula [86, 87]. Pioneering work in the model checking of temporal logic formulae was done by E. M. Clarke and E. A. Emerson and by J. P. Queille and J. Sifakis.

The main principle is to describe a model (an automaton) and a possible execution (a path in the graph). Then model checker will try to validate user's properties on the execu-

[4] http://www.provenrun.com/
[5] http://trust-in-soft.com/

tion. It may respond "true", that is to say "the property is verified", "false" which means "the property is wrong, here is an example why", or "I don't know because the execution is out of bound". This may apply when the model is large, because the model checker is constrained by finite memory and a given execution time. Several approaches exist [88], but generally, SAT solving is used. Model checking is then a NP-hard problem. The number of states of a model can be enormous. For example, consider a system composed of n processes, each of them having m states. Then, the asynchronous composition of these processes may have m^n states. Similarly, in a n-bit counter, the number of states of the counter is exponential in the number of bits, ie 2^n. In model checking this problem is referred to as the state explosion problem [89].

However, several work tend to demonstrate that model checking is achieveable in real-life applications.

Shwartz et al. performed a software model checking for security properties on large scale: a full Linux distribution (839 packages, and 60 million lines of code) [90].

Nasa researchers K. Havelund and T. Pressburger used model checking for Java programs using PathFinder. In [91], they state that even though today's model checkers cannot handle real sized programs, and consequently cannot handle real sized Java programs, there are aspects that make the effort worthwhile:

- providing an abstraction workbench makes it possible to cut down the state space;
- model checking can be applied for unit testing, and thus focused on a few classes only.

3.3.1.2.2 Abstract interpretation Abstract interpretation was formalized by Patrick and Radhia Cousot in the late 1970s [92]. Abstract interpretation is about creating an abstract model of a concrete one, and stating that every property verified in the abstract model will also be verified in the concrete model. An example used by Cousot is the interpretation of an arithmetic expression $-1515 * 17$ (concrete model) applied to the abstract model $\{(+), (-), (\pm)\}$ where the semantic of arithmetic operators is defined by the rule of signs. The abstract execution of $-1515 * 17 \Rightarrow -(+) * (+) \Rightarrow (-) * (+) \Rightarrow (-)$ proves that $-1515 * 17$ is a negative number. One can also represent abstract interpretation in a n-dimension space, where each dimension is bound to a variable, and several "forbidden zones" are to be avoided. Abstract interpretation will approximate the values of each variables using (more or less precise) heuristics, to show that there is no intersection between "forbidden zones" and possible valuations of the analyzed expression. The use of heuristics is compulsory, because general program verification is undecidable.

Abstract interpretation is used in the industry. The following examples have been successfully used in the industry, by AbsInt (Absint.com):

Astrée Astrée (http://www.astree.ens.fr/) is a static program analyzer aiming at proving the absence of Run Time Errors (RTE) in programs written in the C programming language. Astrée was successfuly used in industry to verify electric flight control [93, 94] or space vessels maneuvers [95].

aiT aiT WCET Analyzers statically compute tight bounds for the worst-case execution time (WCET) of tasks in real-time systems [96].

Frama-C Frama-C is Open Source software, which allows to verify that the source code complies with a provided formal specification. For instance, the list of global variables that a function is supposed to read from or write to is a formal specification. Frama-C can compute this information automatically from the source code of the function, allowing you to verify that the code satisfies this part of the design document. Tis-interpreter was built on top of Frama-C.

Polyspace Polyspace Code Prover™ proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code. It uses formal methods-based abstract interpretation to prove code correctness.

3.3.1.2.3 Formal proof Formal proof is for a proof, what a source code is for an algorithm. It translates the proof in an actual proof object, which is usable by a computer. For instance, Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together. But Coq is not dedicated to software prooving. It was used for the fomalization of mathematics (*e.g.* the full formalization of the 4 color theorem or constructive mathematics at Nijmegen).

Formal proofs can use a Hoare-like axiomatic method [97]. Hoare logic is a 3-upplet of values $P\{Q\}R$, where P is called a pre-condition, Q a program, and R a post-condition. Hoare logic states that "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion".

To help reasoning, one uses hypothesis (or axioms) which are assumed to be true, and tries to prove theorems (or lemmas). The output is either a success, and a certificate is given, or a failure, which does not mean that the reasoning is false, but that a proof was not found. The easiest way to prove something wrong is to find a counter-example (like model-checking approaches).

There exist two kinds of tools [98]:

(Interactive) Proof assistant Proof assistants are computer systems that allow a user to do mathematics on a computer, but not so much the computing (numerical or symbolical) aspect of mathematics but the aspects of proving and defining. So a user can set up a mathematical theory, define properties and do logical reasoning with them. In many proof assistants, users can also define functions and perform actual computation with them.

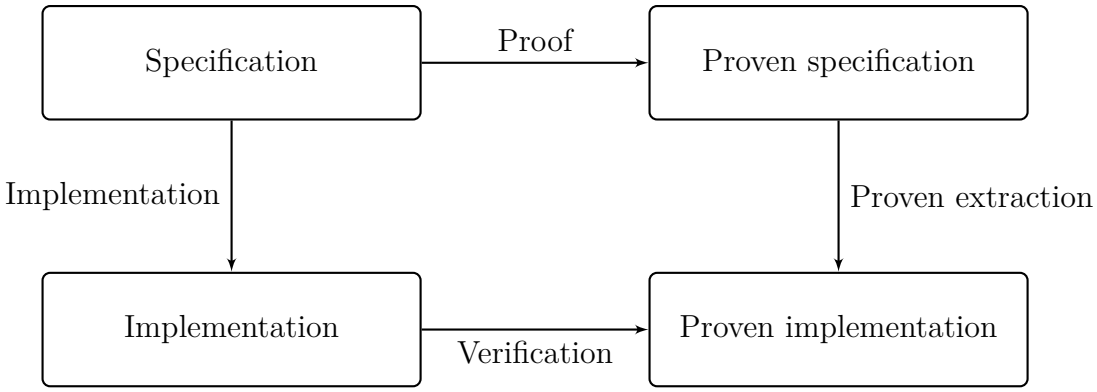


Figure 3.1: Two approaches are possible: (i) writing code, writing a model, prove the correctness of the model and proof the equivalence between the code and the model. (ii) Writing and prove a model, then extract the code from the model with a verified tool.

(automated) Theorem prover Theorem prover are systems consisting of a set of well chosen decision procedures that allow formulas of a specific restricted format to be proved automatically. Automated theorem provers are powerful, but have limited expressivity, so there is no way to set up a generic mathematical theory in such a system.

Formal proof can also be applied to software verification, for instance, Why3 is a platform for deductive program verification. It provides a rich language for specification and programming, called WhyML, and relies on external theorem provers, such as Coq, to prove correctness.

3.3.2 Issues with end to end proof

3.3.2.1 Prove the code's correctness

First of all, one should prove that implementation is correct with regard to its specification. That means that the semantic expressed by the high level model is successfully translated into source code. To gain confidence one can choose several approaches, illustrated in Fig 3.1:

- writing a model, writing an implementation, and proving the functionnal equivalence between those implementations.
- Writing a model and extracting the code from that model. Notice that it leverages a trust dependency on the extractor.

3.3.2.2 Prove the compiler's correctness

The verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

In a future where formal methods are routinely applied to source programs, the compiler could appear as a weak link in the chain that goes from specifications to executables. The safety-critical software industry is aware of these issues and uses a variety of techniques to alleviate them, such as conducting manual code reviews of the generated assembly code after having turned all compiler optimizations off. These techniques do not fully address the issues, and are costly in terms of development time and program performance. An obviously better approach is to apply formal methods to the compiler itself in order to gain assurance that it preserves the semantics of the source programs. CompCERT [99] is such a compiler. It is shiped with a proof of the semantic continuation among the different intermediate languages.

3.3.2.3 Prove the underlying hardware

Once the implementation and the compilation phases are trusted, we can now try to execute the software on actual hardware. In order to maintain the chain of trust, the hardware should also be proven. Such hardware does not exist on the shelf. Worse, the state of art shows that most hardware are not reliable:

Hardware design issues Some bugs have been shown in the wild: Intel Pentium microprocessor occasionally makes errors in floating point division due to five missing entries from a lookup table [66]. More recently Matt Dillon, a DragonFly BSD developer, found out that some AMD processor could incorrectly update the stack pointer [100, 101].

Abuse genuine hardware features

- DMA stands for Direct Memory Access. It is a way to access **physical** memory from hardware without CPU conscent (and thus, without memory translation). It is available over FireWire, Thunderbolt, ExpressCard, PC Card and any PCI/PCI-express interfaces. This lets an attacker read memory at arbitrary location, which could let him grab the screen content, scan possible key material, or extract information from memory layout. DMA can also be used to write data to arbitrary location, which could change the screen content, change UID/GID of certain process, inject code into a process etc ...
- CPU Caches are small banks of memory that store the contents of recently accessed

memory locations. They are much faster than main memory, and have an important impact on system's speed. Modern CPUs have several levels of cache hierarchy. CPU caches have been abused to extract cryptographic secrets [102]

- Intel Active Management Technology (AMT) is a hardware technology for out-of-band management. It is built on top of a secondary service processor located on the motherboard; its memory is separated from the host, and it has a dedicated link to the network card. The AMT is active even if the computer is put into sleep mode. This mechanism was abused by Alexander Tereshkin and Rafal Wojtczuk from Invisible Things Lab. Their work was presented at BlackHat 2009 [103].

Non-temper resistant implementation A temper resistance can occur in safety and security. The environment can be hostile, such as space where hardware has to be hardened against important temperature ranges, and ionizing radiations. Because this thesis is focused on security rather, this part excludes safety concerns.

As described before, most hardware have not been built with security in mind. The SmartCard industy has tried to improve the tamper resistance of its hardware. A smartcard is a tamper-resistant computer, which can securely store and process information. We use them on daily basis: in our SIM cards, credit cards, pay tv.

Even though, successful attacks were conducted using passive measurements on physical elements. In [104], Kocher has demonstrated how timing attacks could lead to cyptographic leakage on common cryptographic algorithms. In [105], similar attacks are achieved by analysing power consumption. Finally, in [106], Quisquater and Samyde noticed that radiation is directly connected to the current consumption of the processor. Hence, it could be used to perform the same attacks in a non-intrusive way.

Those attacks on very specific system illustrate that there is no silver bullet, but protections can be used. It also remind us that we should be suspicious against hardware security features. Most of the time, these are just black boxes with very few information. This is why this thesis takes the side of avoiding to rely on hardware virtualization features.

3.4 A new architecture for embedded hypervisors

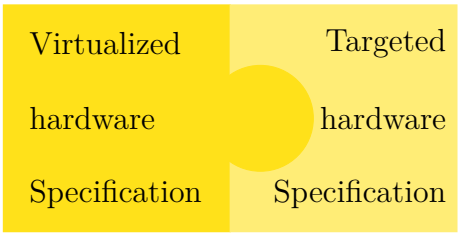
3.4.1 What is to be formalized?

To provide an end-to-end chain of trust on the VMM, the VMM itself must be specified, implemented and proven correct.

3.4.1.1 VMM software

The VMM must be fully specified. Considering the snippet given in Fig 2.7, where each instruction from the **source** ISA is emulated:

- The source instruction set must be fully described. This description should include an operational semantic, corner cases, accepted values for each arguments, raised errors, side effects, etc.
- The analyser should be proven correct. That means that each instruction from the source ISA will be successfully decoded, and interpreted, with regard to the **destination** ISA. Otherwize, an error should be raised.

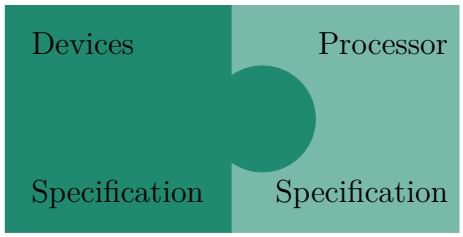


3.4.1.2 Compiler software

Assuming a correct VMM source-code, the compiler must provide guarantee that the generated program will exhibit the same semantic than the source-code.

3.4.1.3 Underlying hardware

When the VMM is correctly compiled, a binary-form program is produced. This program is loaded in memory and executed by the CPU. Each instruction from that binary should be correctly decoded by the hardware, and no different behaviour from the specification should arise.



3.4.2 Factorizing formalization: confound ISA and hardware specification

Having three layers of models / development / proof / validation is tedious. We claim that the VMM and the Hardware should share a large part of the specification. The following schema illustrates the link established by each components: the software, the compiler and the hardware. At the beginning, each layer is independant, and no specification are shared.

I suggest a bottom-up approach, in which lower level specifications are used on the upper layers. In practice, the underlying CPU specification should be reused by both the compiler, and virtual machine monitor's code. In addition, the latter must also integrate the formalisation done on the devices.

Since the instruction set may not be virtualizable (with regard to Popek&Goldberg's definition), we should distinguish the two cases.

3.4.2.1 Popek and Golderg compliant ISA

In that case, all the instructions from the **source** instruction set will be executed directly on the processor. The whole security contracts for those instruction relies on the hardware proof. The latter will raise signals to the VMM. The correct behaviour requires a correctness proof of the handling of those signals.

3.4.2.2 Non Popek and Goldberg compliant ISA

When the instruction set is not virtualizable, a large part of the instruction will still be executed directly on the processor. For the same reason than in the previous section, the hardware proof is suficient to guarantee the correct execution. But the non virtualizable instructions will have to be properly detected by the VMM. Thus, the scanning part of the VMM must also be proven.

3.4.3 Factorizing the interpreter by executing code directly on the CPU

The second axis to reduce the work to be done on both the formalization and the implementation of the interpreter, is to rely on the underlying hardware, formally described and modeled. Hence, non-sensitive instructions do not need to be emulated anymore, but can be run directly on the processor. The native execution of the source-code leads us to modify our basic CPU interpreter (Fig 2.7) into an optimized version (Fig 3.2). In the newer version, only the sensitive instructions are emulated. Hence, the second property (efficiency) formulated by Popek&Goldberg is satisfied: "All the innocuous instructions are executed by the hardware directly , with no intervention at all on the part of the VMM".

Definition 4.1: Basic block

According to the IDA Pro book [107]:

A basic block is a maximal sequence of instructions that executes, without branching, from beginning to end. Each basic block therefore has a single entry point (the first instruction in the block) and a single exit point (the last instruction in the block). The first instruction in a basic block is often the target of a branching instruction, while the last instruction in a basic block is often a branch instruction.

The branch instructions include function calls, jumps, but also conditional statements. In the latter case, the conditional jump generates two possible destinations: the *yes* edge (if the condition is satisfied) and the *no* edge otherwise.

Code

```
Counter=InterruptPeriod;
PC=InitialPC;

for(;;)
{
  OpCode=Memory[PC++];
  Counter-=Cycles[OpCode];

  if (!is_sensitive(OpCode))
     exec_on_cpu(OpCode);
  else
     emulate(OpCode);

  if(Counter<=0)
  {
    /* Check for interrupts and do other */
    /* cyclic tasks here                 */
    ...
    Counter+=InterruptPeriod;
    if(ExitRequired) break;
  }
}
```

Figure 3.2: Modified CPU emulator : non-sensitive instructions now run directly on the CPU

Definition 4.2: Hyperblock

By analogy with the basic blocks, I introduce the concept of hyperblock as follows:

A hyperblock is a maximal sequence of innocuous instructions. Hence, no intervention from the VMM is required. It is safe to let these instructions execute directly on the hardware.

In an emulator (without JIT), the largest hyperblock is one instruction long. With this optimisation, a hyperblock remains smaller than a basic block.

3.5 Discussion

In this chapter, we have described the security features provided by hypervisors, and how hardware can support such features. We have shown that hardwware could not be trusted because of its design and potential bugs. We have mentionned that highest levels of certification require formal verification, which have limited expressivity. Moreover, the formal verification can only proof that a model is valid with respect to a specification. Hence, formal verification itself does not prove security features. Finally, we have illustrated that a correct source-code could only achieve a correct behaviour on verified hardware, if the compiler is also proven to maintain the semantic from the source-code down to the binary.

Based on these elements, we have proposed a design which tries to reuse the largest models from one layer to the other among source-code, compiler and target of execution. This would make a TCB (Trusted Computing Base) as restricted as possible. Without such factorization, proving a VMM would require a proof of the underlying CPU, and a proof of the targeted CPU. The first one would ensure a correct execution of instructions, whereas the latter would ensure that the emulated semantic would be correct. In our approach, only the proof of the underlying hardware is required, because most instruction will map directly on the underlying instructions, and run on the physical CPU. Considering the hardware available on the shelf, (only a proven MMU was certified at EAL7), it was decided to restrict the instruction-set required to run the hypervisor; no virtualization extension should be used. Such instructions are actually micro-instructions, that are firmware programs whose proofs are not available (if they even exist); hence, not included in the models of the processor.

Finally, we have introduced the concept of hyperblocks. Larger hyperblocks avoid context switches to the hypervisor, which reduces the overhead and increases the system's performance.

CHAPTER 4

Contribution

Contents

4.1 Prototype 76
4.2 Performance overview 86
4.3 Hypervising cost – Micro benchmarks overview 90
4.4 Conclusion 92

4.1 Prototype

4.1.1 Foreword: general design of the VMM

The main goal of the hypervisor is to provide an isolated context for guest virtual machines. An isolated context contains memory and states (general purpose registers, configuration registers). To provide isolation, we need integrity and confidentiality. Integrity states that the data and state of that context can only be modified by a running instruction associated with this guest. This instruction can also be run by the VMM, executing the instruction on behalf of the guest. Confidentiality is the dual of integrity. While integrity addresses the write permission, confidentiality addresses the read permission.

The hypervisor acts as a bootloader: it is in charge of starting the board, setting up interruptions, memory protection, and running the guest kernel. Only one guest is supported at the moment, because the main aspect of this thesis is to evaluate the feasibility and overhead of such an approach.

Its second role is to schedule guest(s). Because some instructions do not raise faults in user-mode, the guest will run at the same level of privilege as the hypervisor. Thus, our main concern is to keep control on the execution of the guest code, even when it performs privileged operations. It is assumed that the guest OS will protect itself from userland code. As a consequence, only the code running in privileged mode is analyzed.

Basically, the hypervisor is divided into two parts: an instruction scanner, and an arbitration engine. The former is in charge of analysing raw data, decoding instructions and infering behaviors. Using those information, the latter implements callbacks to decide whether the instruction should run unmodified. The hypervisor's implementation is detailed in the chapter 5 (p 93). This chapter provides a high-level description of the hypervisor's design, and evaluate its performance.

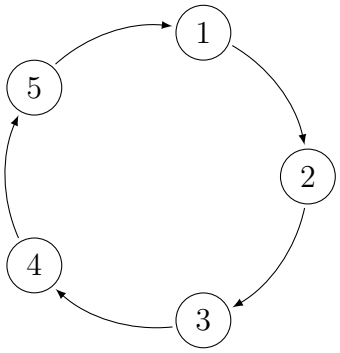
4.1.2 Design

The hypervisor was conceived with platform independence in mind, but currently only ARM architecture has been implemented. Two platforms were targeted: Versatile-Express and Raspberry Pi (the latter is more advanced though). Tests have been performed on Qemu and on a real Raspberry Pi. The hypervisor and the guest both run in privileged mode. The hypervisor can be seen more in a filtering mechanism than an interposition one.

Our main concern is to keep control on the execution of the guest code, even when it performs privileged operations. Thus, no privileged guest code is executed if it has not

previously been analysed by the hypervisor. This ensures that the guest cannot escape the hypervisor analysis. To achieve this, the hypervisor will scan the guest code to identify the next *sensitive instruction*. Sensitive instructions are instruction that read or write processor configuration registers (*e.g.* `CPSR/SPSR`), coprocessors (*e.g.* `MRS/MCR`), when trapping (*e.g.* `SWI, SVC, SMC, HVC, WFE, WFI`) or when they access hardware devices. There are two way to access the hardware: using coprocessors, or accessing memory-mapped registers. The first case is handled by the code analysis (presented in the following section), and the latter by the memory isolation primitives (such as MMU).

When a sensitive instruction is detected, it is saved in the guest context, and replaced by a trap to the arbitration engine of the hypervisor. Then the guest context is restored, and the execution is restarted. The guest code runs natively on the CPU, and eventually a trap will arise and the control will be given back to the hypervisor, in the arbitration engine. Given the exact context of the guest, it will decide whether to let the instruction execute, or to skip it. Some emulation can be performed to update the guest context. This mechanism is illustrated in Figure 4.1.




```
   next_trap <- GUEST_CODE_ADDR
   while KERNEL_MODE; do
1.    next_trap <- find_next_sensitive(next_trap)
2.    patch (next_trap)
3.    exec_guest ()
4.    perform_arbitration ()
5.    unpatch (next_trap)
   done
```

Figure 4.1: Main loop performed by the analyser on a guest code

4.1.3 Analysing guest code

This analysis is performed statically by matching instructions against patterns and sometimes dynamically, crafting an instruction which has the same side effects, such as testing condition flags.

4.1.3.1 Static analysis

This part of the analysis can be compared to the static analysis presented before. In our case, the concrete model would be the guest program (*i.e* a sequence of instructions), and the abstract model would be whether an instruction in the sequence is sensitive or not.

The hypervisor will match each instruction against a matching table in the same manner a disassembler would do. Because of the complex encoding of the ARM instruction set, a simple "switch-case" could not be done. The whole ARM, Thumb and Thumb32 instruction sets were encoded in `Python` tables. Those tables are used to generate `C` code. Some caveats had to be handled:

Mixing instructions and data ARM instruction set allows mixing code and data inside a `code` section. This can lead to false positive detection, as illustrated in Fig 4.2.

Switching to Thumb, and back Most ARM processors have two running states: ARM and Thumb mode. Thumb is a different encoding scheme for instructions, providing a better code density. Switching from ARM to Thumb can be done in four ways:

Branching an odd address: Since all ARM instructions will align themselves on either a 32 or 16-bit boundary, the LSB of the address is not used in the branch directly. However, if the LSB is 1 when branching from the ARM state, the processor switches to the Thumb state before it begins executing from the new address; if 0 when branching from Thumb state, back to ARM state it goes. This is done with explicit branch, but also with several operations on `PC`;

Using "exchange" instruction: Some instructions have the side effect of switching mode from ARM to Thumb and back. Such instructions are usually suffixed by **X**, for instance the `BLX` instruction.

On exception: The same vector table is used for both Thumb-state and ARM-state exceptions. An initial step that switches to the ARM state is added to the exception handling procedure. This means that when an exception occurs, the processor automatically begins executing in the ARM state.

Setting the `SPSR.T` bit: This is used to restore the Thumb state after an interrupt handling. a `RFE` or `SUBS PC, LR, #imm8` will restore the `CPSR` with the `SPSR` . So if the later was set in Thumb, the processor will switch to Thumb.

As a consequence, the hypervisor has to detect all instructions which break the control flow explicitly (*e.g.* `B 0x12345678`) or implicitly (*e.g.* `SUBS PC, LR, #imm8`).

Some instructions cannot be matched statically, for instance register to register operations, or conditional execution. To address this issue, dynamic analysis is performed.

4.1.3.2 Dynamic analysis

Some cases cannot easily be handled by the static analysis. For instance, register-to-register operation would have to be interpreted and emulated in order to track the actual value of

```c
int doit(void)
{
  unsigned int a, b;
  a = 4043440144;
  b = 1;
  return a + b;
}
```

```
00000000 <doit>:
   0:   e52db004        push    {fp}        ; (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
   8:   e24dd014        sub     sp, sp, #20
   c:   e59f3024        ldr     r3, [pc, #36]   ; 38 <doit+0x38>
  10:   e50b3008        str     r3, [fp, #-8]
  14:   e3a03001        mov     r3, #1
  18:   e50b300c        str     r3, [fp, #-12]
  1c:   e51b2008        ldr     r2, [fp, #-8]
  20:   e51b300c        ldr     r3, [fp, #-12]
  24:   e0823003        add     r3, r2, r3
  28:   e50b3010        str     r3, [fp, #-16]
  2c:   e24bd000        sub     sp, fp, #0
  30:   e49db004        pop     {fp}        ; (ldr fp, [sp], #4)
  34:   e12fff1e        bx      lr
  38:   f1020010        cps     #16
```

Figure 4.2: On the top of this figure, is a piece of C code which performs an arithmetic operation. This example shows that disassembling each instruction one after another can lead to false results. In this case, the first operand's value is a numeric value, but it can also be decoded as `CPS`. Replacing it by a trap to the hypervisor would change the program behavior.

those registers. This is unsuitable because:

- It would require a lot of code (basically writing an emulator for the instruction set);
- This code would have to be proven correct;
- This would break the efficiency properties of Popek&Goldberg, stating that most of the instructions should be executed without hypervisor intervention.

To keep a hypervisor as small as possible, with the highest level of fidelity to the hardware, a dynamic approach was used. It is used to tackle several issues:

4.1.3.2.1 Conditional execution ARM supports conditional execution. The easiest way to determine whether a condition is fulfilled is to execute a dummy instruction which shares the same condition flags. That instruction could be a simple conditional move of a given value in a given register; reading this register's value gives the state of the flag. An example is given in Fig 4.3.

```
1  @ initial instruction
2  @ bne 0x00508000
3
4  movne r0, #1
5  cmp   r0, #1
6  beq   condition_true
7  b     condition_false
```

Figure 4.3: The original instruction was a conditional branch. Given the guest context (general registers and CPSR), we can craft an instruction which will conditionally be executed. Knowing the side effect (assigning 1 to r0), we can detect whether the condition was verified or not.

4.1.3.2.2 Dynamic assignment Assignment instructions can be used to perform branch, for example: `mov pc, r3` or `add pc, pc, #4`. There are some implicit rules (documented though) which need to be known. For instance, when executing an ARM instruction, `PC` reads as the address of the current instruction plus 8. But the instruction may also be much more complex, like `ldrls pc, [pc, r2, lsl #2]`.

In order to preserve the highest fidelity to the underlying processor, every instruction which has `PC` as a destination register is handled using self-modifying code. Given a guest context, the destination register (`PC`) is replaced by another register (for instance `r0`), and the instruction is executed on the processor. Afterward, we get the actual destination in `r0` accurately.

4.1.4 Interrupt handling

4.1.4.1 Keeping track of the guest interrupt vector

The hypervisor is the recipient for interrupts. It must handle its own interrupt handler, and sometimes forward the interrupt to the guest. Thus, the hypervisor must keep track of the interrupt vector. On ARM, there are two ways to setup an interrupt vector:

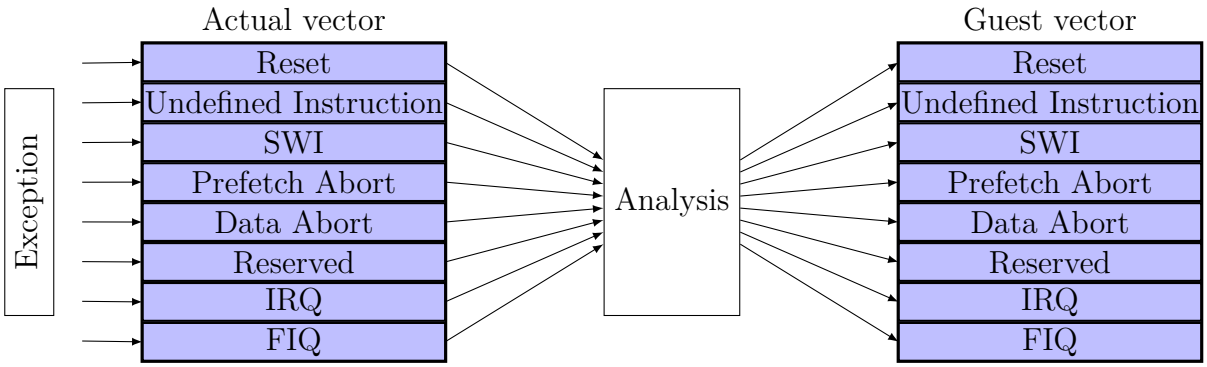
Writing at `0x00000000` or `0xFFFF0000`: The ARM interrupt vector can be located either in low memory or in high memory. When a guest writes between this base address to base address + table size, it is setting the interrupt handlers.

Setting `VBAR`: VBAR is the Vector Base Address Register. Modifying its value has side effect to actually set news interrupt handlers.

The hypervisor keeps track of the guest interrupt handlers updates, but prevents the hardware from being updated. The hypervisor must always receive all the interrupts.

4.1.4.2 Handling interrupts

Because it controls the exception vector, the control flow is restored to the hypervisor whenever an exception or an interrupt arises. If the interrupt was targeted to the guest, the hypervisor wraps the guest interrupt-handlers with the analysis mechanism described in 4.1.2 (p 76). That way, even interrupt code is handled by the hypervisor, and no sensitive operation can be performed without prior validation by the hypervisor.



4.1.4.3 Reset

Reset is invoked on power up or a warm reset. The exception model treats reset as a special form of exception. When a reset is received, the hypervisor performs the board

initialization, sets-up memory protection (to prevent the guest from accessing or configuring hardware), sets-up a guest context, loads the guest at the specified address, and starts analysing the guest from its start address.

4.1.5 Tracking privileges changes

The trap-and-emulate approach is known to be expensive in terms of context-switches from one privilege level to the other. Moreover, ARM processors only have two execution modes: privileged and unprivileged. Because some sensitive instructions are context dependent, and do not trap in unprivileged mode (`mrs r0, cpsr` for instance), it was decided that both the guest and the hypervisor will run in privileged mode. This way, no tricks must be setup to lure the guest on its actual privilege level.

Despite some instructions will not trap in unprivileged mode, it cannot have a significant impact on the system's setup, except accessing specific memory regions. This part of the isolation of the hypervisor is handled by the MMU. Moreover, the guest-OS is supposed to protect itself from userspace programs. The hypervisor only provides isolation, and is not supposed to provide additional security on the guest. Thus, unprivileged code (userspace code) is not analyzed. The hypervisor tracks privilege changes in the guest so that when user mode is triggered, the hypervisor stops hypervising, and is rescheduled in privileged mode (via interrupts handlers). This way, usermode code runs directly on the CPU without performance degradation.

4.1.6 Metrics

Linux currently ships two hypervisors[1]: Xen and lguest. They both require guest OS to be modified[2], that is called paravirtualization.

Lguest is roughly 6,000 lines of code, including userspace code. [77] states that Xen hypervisor is 100,000 lines of code, 200,000 lines for the dom0 (management virtual machine required to perform the administration operations), and some 130,000 lines for QEMU. In version 3.0.2 of Xen, the source-tree is separated in $\sim$ 650 files containing 9,000 lines of assembly and 110,000 lines of C, whereas version 4.7.0 is composed of $\sim$ 1200 files, 7,000 lines of assembly, and 265,000 lines of C[3]. Xen also requires some adaptation in the guest code. [20] mentions a total of 2995 lines to modify on Linux to support Xen. Nova [77] is a micro-hypervisor composed of 9,000 lines of code, 7,000 user-level code and 20,000 lines of code for the VMM.

In comparison, our hypervisor is composed of 4,400 lines plus 12,000 generated lines.

4.1.6.1 CPU description

We have used a pseudo-dedicated language to encode the instruction set. The chosen language was `Python` because it is easily readable, and has interesting built-in features such as `map` or `filter` on lists. Using these two functions made it easy to isolate instructions in the global instruction table, to generate matching functions for several behaviours. Both ARM and Thumb instructions were encoded. Some metrics are provided in Fig 4.4.

CPU mode	Instructions	Entries in table	Overhead
ARM	299	976	226.4%
Thumb	276	1111	302.5%

Figure 4.4: Number of entries in the `Python` table for ARM and Thumb instruction set.

Because we wanted to extract multiple information from the same instructions, we have used "behaviors". When recognized, the information associated with an instruction is bound to several behaviors. These behaviors provide insights for the arbitration engine. The behavior list is the following:

Behavior_OddBranch: Targets `PC`, needs a dynamic execution to resolve the address;
Behavior_Link: Will update `LR`;

[1] It also provides Cgroup API, which is used for containers, but this is considered out of scope here

[2] Considering the first Xen releases, HVM was not available

[3] Those metrics come from the Xen source-tree, using `cloc(1)` to count lines of code. Cloc splits lines of code, blanks and comments. This metrics are thus, code only.

Behavior_ReadsCopro: Reads coprocessor;
Behavior_WritesCopro: Writes coprocessor;
Behavior_WritesPsr: Writes Process Status Register;
Behavior_ReadsPsr: Reads the Process Status Register;
Behavior_HardwareAccess: Accesses hardware ;
Behavior_Unpredictable: Has an unpredictable behavior;
Behavior_Branches: Changes the control flow;
Behavior_CanChangeState: Changes state (from ARM to Thumb, or Thumb to ARM);
Behavior_RaisesException: Raises exception.

This part of the project is written in `Python` and produces `C` code. The metrics are provided in Fig 4.5.

Language	Files	Code
C (gen)	1	11800
Python	4	2300

Figure 4.5: Metrics in lines of code to encode the ARM processor specification.

4.1.6.2 Hypervisor

Because most of the instructions are performed on the CPU, almost no emulation is done. Thus, the code size of the hypervisor is small (4,200 lines of C). We have tried to reduce the amount of assembly to the minimum, because it is known to be hard to audit, hence to formally verify. From that verification point of view, it is commonly admitted that projects with less than 10,000 lines of code are good candidates to work on.

Language	Files	Code
C	33	4,200
C (gen)	1	11,800
Python	4	2,300
Assembly	6	400
Total	234	18,700

Figure 4.6: Synthesis of the code size for the hypervisor.

4.1.6.3 Discussion

Considering Nova -which is one of the smallest hypervisor in the literature, the core of our implementation is roughly two to five times smaller.

Implementing the dynamic analysis mechanism was tricky, because of some instruction's side effects, but it is quite easy to isolate. Once written, these features did not need additional work.

In contrast, implementing the analysis was much more troublesome. The ARM instruction set specification is lengthy and intricate. Exploiting this documentation is hard. First of all, we had to extract the binary encoding from printed documentation. Afterward, an internal representation had to be found. That piece of work was complex enough for us to decide to write a generator. That generator was written in `Python`. It is composed of instruction description (90%) and code processing this data to produce C code (10% remaining).

These 2,300 lines of Python code (2,000 describing the processor, 300 to generate code) produce $\sim 12,000$ lines of C code. This represents the hard part of this work. To gain sufficient confidence in the implementation, this part should be extracted from either a formalized version of the ARM instruction set; or the evaluation candidate should provide a proof of correction of that table. This is the real challenge to achieve a genuine, formally proven implementation of an hypervisor, because at the moment, even SeL4 (which is a reference in this domain) assumes correctness of hand-written assembly code, boot code, management of caches, and the hardware [108].

4.2 Performance overview

4.2.1 Benchmarks

This section presents the benchmarks used to evaluate the performances of the hypervisor. Four benchmarks were used to validate the chosen design:

Dhrystone: a synthetic benchmark program designed to measure integer computation performances [109],
Puts: a simple function that performs an output using a hardware UART,
Fibonacci: the classical Fibonacci sequence. This benchmark is a recursive function that makes heavy use of the stack,
AES: this task performs the self test function of the mbed TLS [110] (formerly PolarSSL) AES implementation for ECB, CBC and CTR ciphering mode. This function performs encryption and decryption of data buffers with different cipher modes.

4.2.2 Evaluation

The benchmarks are run in two distinct configurations: first, on top of an operating system, as in existing systems; secondly, on top of the same operating system, running on top of our hypervisor.

The first case evaluates the performance which is achievable on the actual system in nominal operating mode. The second one evaluates the overhead induced by the hypervisor. Only the privileged code is run under hypervision. Unprivileged code is executed directly on the processor, achieving native performances. Since this is one of the main traits of the proposed design, we expect a limited performance overhead (tenth of percents).

4.2.3 Results

This section provides results of the benchmarks in various context. Unless otherwise stated, the results are given in milli-seconds.

Figure 4.7: On the left side: The benchmarks are run on top of an operating system. On the right side: the underlying operating system is run on top of our hypervisor.

4.2.3.1 On top of an operating system

This configuration is the traditional case of benchmarks running on top of an operating system. In our case, the operating system is a homemade preemptive, multi-task operating system, implementing a round-robin scheduler. Five tasks are created, one containing the benchmarks and four doing random work (homogeneous though). At the beginning, one task per benchmark was created. Because all the benchmarks don't have the same execution time, some tasks were terminated before others. As a side effect, the quantum associated for each task was modified: since execution are terminating, there are fewer task to schedule. Hence, the time spent by the system in the scheduler was modified. Using one unique task for the benchmarks and infinite tasks to feed the scheduling queue solved this issue.

Bench	Bench Result	# Run	Execution Time (ms)
NOP SYSCALL	101	100	**1.01**
PUTS SYSCALL	3670998	100	**36709.98**
Fibonacci	5959942	1	**5959942**
Dhrystone	13852709	200	**69263.545**
AES	103767253	5	**20753450.6**

4.2.3.2 On top of an hypervised operating system

This configuration is the same as above, except that the operating system does not directly run on the hardware anymore. Instead, it is started in our hypervisor. The results are also given in milliseconds.

BENCH	Bench Result	# Run	Execution Time (ms)
NOP SYSCALL	1854375	100	**18543.75**
PUTS SYSCALL	10161195	100	**101611.95**
Fibonacci	6002561	1	**6002561**
Dhrystone	14454528	200	**72272.64**
AES	108536430	5	**21707286**

4.2.3.3 Performance overhead

This section compares the results, and exibits the overhead induced by the hypervisor.

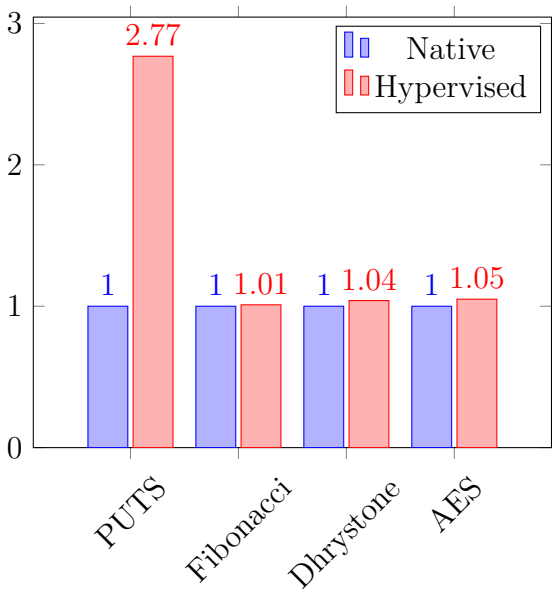
BENCH	Native	Hypervised	Overhead
NOP SYSCALL	1.01	18543.75	18360.148
PUTS SYSCALL	36709.98	101611.95	2.768
Fibonacci	5959942	6002561	1.007
Dhrystone	69263.545	72272.64	1.043
AES	20753450.6	21707286	1.046

4.2.4 Analysis

As described in 4.2.1 (p 86), these benchmarks do not perform operating system like instructions; in particular, they do not access hardware (except the UART, which is allowed) nor coprocessors. Because of their nature, these benchmarks are well suited for userspace execution. Where does the overhead come from? As explained in 4.2.3.1 (p 87), each task is run in userspace, and is preempted by a timer (set to 10ms). Thus, at every tick, the hypervisor takes back control, and performs analysis on system code, responsible for scheduling. In order to quantify the overhead in kernel time, the same benchmarks are run in privileged mode (hence, under hypervision). The corresponding results are presented in the next section.

The `NOP` benchmark is equivalent to a userspace program for which a scheduling policy would be cast before each instruction. A software interrupt is generated, which transfers control to the hypervisor (because the hypervisor handles all the interrupts). A new analysis for the guest's interrupt handler is performed. The guest handler simply performs a "return from interrupt". As a side effect, the processor is switched back to unprivileged mode. Hence, the analysis terminates. The overhead is important, because of:

1. the hypervisor interposes between the guest and the interrupts;
2. the cost induced by the context-switches starting and stoping the hypervisor (hence backuping states);
3. the actual hypervision of the guest code.



The same reasons hold for the `PUTS` benchmark. In that case, more time is spent in kernel mode handling the busy-wait loop on the UART hardware, so (1) and (2) are small(er) compared to (3). Also, the overhead of analysing the guest code (third item) overlaps with the time spent by the kernel waiting for the hardware to be ready: under hypervision, fewer iterations are made waiting for the FIFO to be ready.

In the general case, the hypervisor cannot be compared to hardware-assisted products. However, in specialised applications applied to embedded systems, our approach has shown a reasonable overhead (10% average). Our proposal might be naive compared to the aggressive strategies used by modern hypervision techniques. Nevertheless the performance impact is not critical in light of those benchmarks. This is mainly due to the fact that only the privileged code is hypervised, not the applications.

Those results are good essentially because the MMU does its job. It ensures that userspace will behave correctly: it ensures a proper isolation between userspace and kernelspace, so that the userspace cannot compromise the security of the kernel, nor the hypervisor's. This reinforces the need of a high level of hardware certification. Such hardware exist, in 2013 Samsung Eletronics Co. got an EAL7 certification for a Memory Management Unit for RISC microcontroller [111, 112].

The worse benchmark result is `PUTS`, which has 276% cost. In 2007, a report showed that a paravirtualized linux running on top of Xen has an overhead of 310% in reception, and 350% in emission [113]. Also, we have measured the same overhead in the micro-benchmarks presented in the following section.

4.3 Hypervising cost – Micro benchmarks overview

4.3.1 Benchmarks

This section discusses the total cost of hypervision. As opposed to the previous section which qualified the overhead on a system, this one provides an analysis on a bare-metal system. The whole system plus benchmarks will be evaluated under hypervision. In that case, the hypervisor basically does nothing, because no hypervision is needed for those benchmarks to work. Even-though the benchmarks do not access hardware, their control flow is much larger. We expect an important overhead.

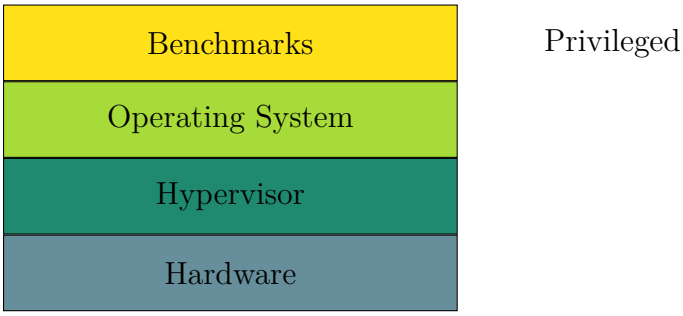


Figure 4.8: As opposed to the previous ones, benchmarks are run in privileged mode. Hence, they are executed under hypervision.

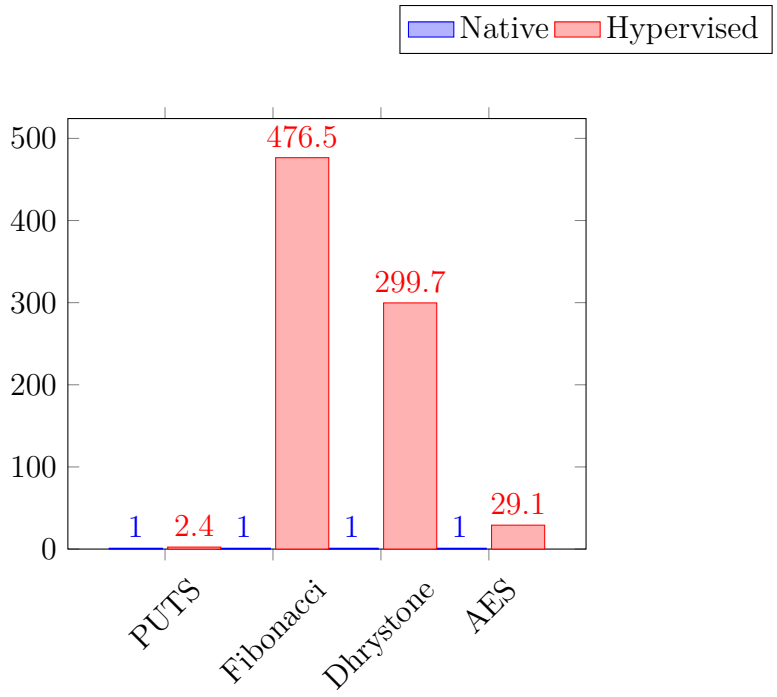
4.3.2 Results

Bench	Time hypervised	Time Native	# Runs	Overhead
PUTS	140019	57200	10	2.448
Fibonacci	21410805	44931	10	476.526
Dhrystone	36175704	120697	10	299.723
AES	21633224	743360	4	29.102

4.3.3 Discussion

Those micro-benchmarks present much worse results than macro-benchmarks. The multiple context-switches between guest and hypervisor become hot-spots.

Full virtualization (as opposed to system virtualization from the previous section) is not realistic. The kernel code should be limited as much as possible. This reflects the



micro-kernel design, which reduces the kernel code to the minimum in favor of userspace. That is the credo followed by Minix [1], and also in some embedded systems (Erika [114] in automotive industries, Camille [115] for Smart Cards), but not on general purpose system, where linux is number one.

Hypervisors are candidates to provide security by isolating security vulnerabilities from bare-metal software such as micro-controllers firmwares. For this kind of software, no distinction is made between userspace and kernel-space. All the code is run in privileged mode. For such applications, our approach is not feasible: two orders of magnitude in performance degradation is prohibitive. Thereby reducing that hypervision cost is advisable.

The analysis performed in this work is the simplest, in order to ease the certification work. A cache mechanism was introduced, to avoid analysing several times the same instructions. If the targeted guest is a monolytic kernel, or bare-metal privileged application, it is necessary to extend the analysis. The context-switch cost is expressed as follow:

$$C = S * N$$

where C is the total cost, S the context-switch cost, and N the number of context-switch. Because $N >> S$, the number of context switch should be addressed first. The analyser would need to be smarter in the detection of sensitive instructions, or in the control flow tracking. That way, it would allow more code to be executed without context-switches; hence, increasing overall performances.

4.4 Conclusion

This chapter presents a new design for a "certifiable" hypervisor. For that purpose, several axes are presented:

Limit the size of the formal model: which implies to re-use specification from one layer to another (hardware spec in both compiler and software layer),

Rely only on provable elements: this is why no virtualization features are used. We argue that using MMU is fair, because it has already been certified at level 7 for SmartCard.

Reduce the codebase: and avoid writing specific code. Most of the job should be done on the CPU, even if it seems simpler to emulate the instruction.

With this design in mind, a proof of concept was written in C and assembly. Python code was used to generate C tables. The code base is quite small: less than 5,000 lines of code, excluding the generation part. Despite extensive tests, we cannot ensure that the Python description is error-free. But because the hypervisor uses a large table, the verification job can be done separately on the table (or generator) and on the hypervisor. Currently, the `Python` table does this job, but it is not sufficient. This should be done either by a formal specification provided by the manufacturer, or provided by the certification candidate.

In favorable cases, the performances are decent ($< 15\%$) and encourage a micro-kernel style architecture for embedded systems. This design is suitable because of the responsibilities isolation between kernel (privileged code) and actual tasks (like sensing, processing and communicating). There are two main axis to improve performances: either reduce the cost of a context switch (from guest to hypervisor), or reduce the number of context switch.

Security is said to be a trade-of between usability and security: having a very secured system will likely be a burden for users (think of password patterns, frequent changes...), whereas having an easy to use system will probably have less security mechanism. The same kind of comparison can be done between performance improvements and certifiability. The underlying theory to express formal properties is a limiting factor in this case. For instance, expressing concurrent access is a hard problem. The current implementation is (in my opinion) a fair share between tricky mechanism and understandability.

CHAPTER 5

In depth implementation details

> You are absolutely deluded, if not stupid, if you think that a worldwide collection of software engineers who can't write operating systems or applications without security holes, can then turn around and suddenly write virtualization layers without security holes.
>
> *Theo de Raadt*

Contents

5.1 General concepts about the ARM architecture and instruction set 94
5.2 Analyser's implementation: the instruction scanner 100
5.3 Arbitration's implementation 104
5.4 Example on a simple guest 107
5.5 Summary 107

5.1 General concepts about the ARM architecture and instruction set

This section describes technical details of the hypervisor. First, an overview of the ARM architecture is given. Next, the instructions encoding schemes are presented. Then, the instruction scanning layer of the hypervisor is detailed. Finally, the dynamic solutions for the issues presented in the previous chapter are explained.

5.1.1 The ARM architecture overview

The ARM core uses a RISC (Reduced Instruction Set Computing) architecture. While RISC emphasizes compiler complexity, CISC (Complex Instruction Set Computing) architecture emphasizes hardware complexity. Each platform is composed of a processor and peripherals, which includes coprocessors. Coprocessors are dedicated hardware mechanism which perform specific tasks such as virtual memory handling for example. Coprocessors are addressed using specific instructions (`MCR` and `MRC`) while more classical peripherals (such as network card) are memory-mapped. The ARM processor uses a load-store architecture. Two kinds of instructions exist to transfer data in and out of the processor: load and store. There are no data processing instructions that directly manipulate data in memory. All data processing is carried out in registers. General purpose registers hold either data or an address, and program status register contains information on the current execution state (such as processor mode, interrupt state, and condition flags). There are 16 general purpose registers (from r0 to r15), but some of them have a dedicated role:

- `r13 (or sp)` holds the stack pointer;
- `r14 (or lr)` holds the return address in a subroutine;
- `r15 (or pc)` holds the program counter.

Depending on the generation used, ARM processor supports several processor modes, which are either privileged or non-privileged. Switching from one mode to the other exposes new registers (called banked registers). The processor keeps a safe copy of them; and will restore them upon returning in the previous mode. The number of banked registers depends on the target mode. Finaly, is is important to recognize the difference between a processor's family name and the Instruction Set Architecture (ISA) version it implements. For example, the first Raspberry Pi includes a 700 MHz ARM1176JZF-S processor. This means that it belongs to the 11th family, and implements cache, MMU with physical address tagging, with Jazelle support (J flag), TrustZone extension (Z flag), VFP instructions (F flag) and synthesizable (S flag).

But according to [116], the ARM core is not a pure RISC architecture:

Variable cycle execution: not every instruction takes the same number of cycles to execute. In [116], the appendix D is dedicated to the instruction cycle timings. For example, ALU operations take one cycle, whereas branch instructions take 3. Accessing coprocessors takes between one to three cycles, plus a number of busy-wait cycles issued by the coprocessor.

Inline barrel shifter: the barrel shifter is a hardware component that preprocesses one register before the instruction is executed. This leads to more complex instructions.

Thumb instruction set: some ARM processors can execute in ARM mode, or in Thumb mode. Further details are provided in the following sections.

Conditional execution: an instruction is only executed if a specific condition has been met.

Enhanced instructions: some specific instructions were added to the standard ARM instruction set, such as DSP, which support fast 16x16bits multiplier operations and saturations.

Despite the break with the RISC traditional architecture, those features make the ARM instruction set suitable for efficient embedded applications.

5.1.2 The ARM instruction set families

The ARM instruction set is composed of five instruction classes:

- Data processing instructions;
  - Move instructions;
  - Arithmetic instructions;
  - Logical instructions;
  - Comparison instructions;
  - Multiply instructions;
- Branch instructions;
- Load-store instructions;
  - Single-register transfer instructions;
  - Multiple-register transfer instructions;
  - Swap instructions;
- Software interrupt instructions;
- Program status register instructions;

This constructor-defined classes can be summarized in the following:

Register to register instructions: all the instructions that perform operations on general purpose registers such as arithmetic operations, move operations, shift operations.

Memory access instructions: instructions that load or store data from (resp. to) mem-

ory.

Control-flow instructions: all the instructions whose role is to change the program counter.

Trapping instructions: which are used (mainly) to perform system calls, and lead to (basically) switching to privileged mode.

Hardware configuration instructions: all the instructions which access (reading or writing) to hardware configuration, coprocessors, or status-register.

In practice, some instructions can exhibit unexpected behaviours ; for example, arithmetic instructions can be used to produce a `branch` if the destination register is `PC` .

5.1.3 The ARM instruction set encoding

The ARM instruction set is actually separated into several sub-instruction set. The following subsection provides an overview of those features. The first subsections (5.1.3.1 and 5.1.3.2) are relevant, and handled by the hypervisor, whereas the other one are not: they should not be required in kernel code, and the hypervisor could trap on them to prevent their execution.

5.1.3.1 ARM

The ARM instruction set encodes instructions on 32 bits, as illustrated in Fig. 5.1.

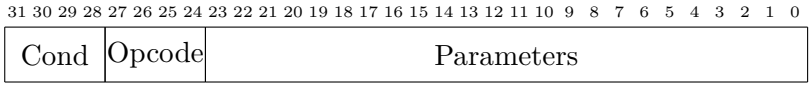


Figure 5.1: An ARM instruction encoding. First part is the condition bits, then comes the opcode, and finally the parameters for the instruction.

The ARM instruction set encoding seems regular and straightforward, thus easy to analyze. But the reality is different. The "Opcode" field (illustrated in Fig. 5.1) is four bits encoded. Hence, it can store up to 16 distinct values. But instruction belonging to the "branch instructions" class aforementioned, such as branch with link, and chance CPU mode (`BLX`) has an opcode value of 1 (%0001) whereas the immedia branch (`B`) as an opcode value of 10 (%1010). This misuse of the opcode qualifier is also present among different classes: the opcode 1 (%0001) is also used to encode load-store class instructions, such as `LDRD`. There also exist unconditionnal instructions (such as `MCR2`), that do not have a proper conditionnal value and use 15 (%1111), which is different from the **execute always** condition flag 14 (%1110). These cases are illustrated in Fig. 5.2.

These very simple examples illustrate that decoding ARM instruction is not as simple

Instruction	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	cond				1	0	1	0	signed 24-bit branch offset																							
BLX	cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	Rm			
LDRD	cond				0	0	0	1	U	0	W	0	Rn				Rd				0	0	0	0	1	1	0	1	Rm			
MCR2	1	1	1	1	1	1	1	0	op1			0	Cn				Cd				copro				op2			1	Cm			

Figure 5.2: Examples of inconsistend usage of the opcode part of the encoding scheme.

as one would think. Hence, using a simple "switch/case" on the opcode value is not longer suitable.

5.1.3.2 Thumb - Thumb32 and ThumbEE

ARMv4T and later define a 16-bit instruction set called Thumb. Most of the functionality of the 32-bit ARM instruction set is available, but some operations require more instructions. The 16-bit Thumb instruction set provides better code density, at the expense of performance.

ARMv6T2 introduces a major enhancement of the 16-bit Thumb instruction set by providing 32-bit Thumb instructions. The 32-bit and 16-bit Thumb instructions together provide almost exactly the same functionality as the ARM instruction set. The 32-bit Thumb instruction set achieves the high performance of ARM code and better code density like 16-bit Thumb code.

ARMv7 defines the Thumb Execution Environment (ThumbEE). The ThumbEE instruction set is based on Thumb, with some changes and additions to make it a better target for dynamically generated code, that is, code compiled on the device either shortly before or during execution.

Wether you start in ARM or Thumb mode depends on your board. The CPU mode can be infered by reading the `CPSR.J` and `CPSR.T` flags. The instruction set is given in the Fig. 5.3. There is a (funny) chicken/egg problem here: how does the compiler know how to encode the instructions to read the CPU mode, without knowing the CPU state, hence which encoding to chose?

J	T	Instruction set state
0	0	ARM
0	1	Thumb
1	0	Jazelle
1	1	ThumbEE

Figure 5.3: Instruction set state values given `CPSR.J` and `CPSR.T`.

The tricky part of Thumb instruction set is that it has a **variable** instruction size. Those instruction can either be 16 or 32 bits. Also, even if they are 32 bits in size, Thumb

instruction will be 16 bits aligned. The CPU does not detect automatically whether the data pointed by PC is ARM or Thumb code. Switch from one mode to the other are tracked into `CPSR.T`, but one cannot change this bit in the CPSR to switch mode. Basically, almost all branches, function call, return, exception entry or return can switch mode.

Branching an odd address: Since all ARM instructions will align themselves on either a 32 or 16-bit boundary, the LSB of the address is not used in the branch directly. However, if the LSB is 1 when branching from ARM state, the processor switches to Thumb state before it begins executing from the new address; if 0 when branching from Thumb state, back to ARM state it goes. This is done with explicit branch, but also with several operations on `PC`;

Using "exchange" instruction: Some instruction have the side effect to switch mode from ARM to thumb and back. Those instruction are usually suffixed by **X**, for instance *BLX* instruction.

On exception: The same vector table is used for both Thumb-state and ARM-state exceptions. An initial step that switches to ARM state is added to the exception handling procedure. This means that when an exception occurs, the processor automatically begins executing in ARM state.

Setting the `SPSR.T` bit: This is used to restore Thumb state after an interrupt handling. a `RFE` or `SUBS PC, LR, #imm8` will restore the `CPSR` with the `SPSR` . So if the later was set in Thumb, the processor will switch to Thumb.

5.1.3.3 Neon and VFP

NEON is a SIMD accelerator processor part of the ARM core. It means that during the execution of one instruction, the same operation will occur on every data sets in parallel. VFP stands for Vector Floating Point. It often comes with NEON on ARM processors. As its name stands, VFP is floating point capable, whereas NEON works on integers. From now on, we consider NEON and VFP as a single computation mode.
There are several ways to use such features:

Instructing the compiler: directs the compiler to auto-vectorize, and produce NEON code.

Using NEON intrinsics: those compiling macro provide low access to NEON operations.

Writting assembly code: provides a *de facto* access to NEON mnemonics.

This unit comes up disabled at power on reset. User willing to use that unit has to manually start it using coprocessor dialect. This computation mode is not handled by the hypervisor. This limitation can easilly be explained; such instruction are not required for an operating system's kernel. Moreover, we could argue that we would detect probe / enabling of NEON features thanks to the copro access which are catched by the hypervisor.

It would be easy to return an error code to prevent the guest from using Neon while it's not properly handled by the hypervisor.

5.1.3.4 Jazelle

Quoting ARM Architecture Manual, Jazelle is an execution mode in ARM architecture which "provides architectural support for hardware acceleration of bytecode execution by a Java Virtual Machine (JVM)". It came from the famous Java adage "Write once, run anywhere". BXJ is used to enter Jazelle mode. Jazelle has it own configuration register (JMCR). Jazelle mode has to be enabled in the CPSR. For the same reason that we do not handle NEON/VFP, we decided not to handle Jazelle in the hypervisor.

5.1.3.5 Security extentions (TrustZone)

As described in 2.2.3.2 (p 38), TrustZone is an architectural extension targeting security. Because operating systems become more and more complex, it becomes difficult to verify the security and the correctness in the software. The ARM solution is to add a new operating state, where a small, verifiable kernel will run. This kernel will provide services to the "rich-operating system" using a message passing mechanism, built on top of the `SMC` instruction.

The security extensions define two security states, Secure state and non-secure state. Each security state operates in its own virtual memory address space, with its own translation regime [117]. Using security extensions provides more security than systems using the split between the different levels of execution privilege:

- The memory system prevents the non-secure state from accessing regions of the physical memory designated as secured;
- System controls that apply from the secure state are not accessible from the non-secure state;
- Entry/exit from the secure state is provided by a small number of mechanisms;
- Many exceptions can be handled without changing security state.

The security extensions are not adressed by this thesis.

5.2 Analyser's implementation: the instruction scanner

5.2.1 Instruction behaviors

As explained in 5.1.2 (p 95), the large-grained families are not sufficient. For example, arithmetic instruction applied to `pc` changes the control flow. In that case, branching instruction would better qualify the actual behaviour of the instruction. Hence, we have identified the following fine(r)-grained families:

Family description	Register to register	Memory access	Control flow	Trapping	Hardware config
Reads memory	✗	✓	✗	✗	✗
Writes memory	✗	✓	✗	✗	✗
Explicit branch	✓	✗	✓	✗	✗
Branch by an odd way	✓	✗	✓	✗	✗
Subrouting call	✓	✗	✓	✗	✗
Changes the instruction set	✗	✗	✗	✗	✓
Can raise an exception	✓	✓	✓	✓	✓
Begins an IT block	✗	✗	✓	✗	✗
Reads a coprocessor	✗	✗	✗	✓	✓
Writes a coprocessor	✗	✗	✗	✓	✓
Reads a *PSR	✗	✗	✗	✓	✓
Writes a *PSR	✗	✗	✗	✓	✓
Unpredictable instructions	✗	✗	✗	✗	✗
Unsupported instructions	✗	✗	✗	✗	✗
Accesses hardware	✗	✗	✗	✓	✓

These newly defined behaviour are more precise and remove some ambiguities. Notice that the first families (read/write memory) are not used in practice because of the performance overhead. Instead, a hardware memory protection facility is used.

5.2.2 Instruction matching

5.2.2.1 The match table

Because the opcode part of the encoding was not suitable to discriminate the instructions, we have created a matching table which associates a mask-value couple to associated behaviours. This table is generated in C, by a Python program. Instructions encoding have

been extracted using "human OCR" from the ARM reference documentation [117]. A proper specification would have been much faster, easier and less error-prone (because OCR may have false positive, but as humans, we are even more subject to misreading or miswriting). Figure 5.4 illustrates a fragment of the generated C table. This mechanism is similar to routing algorithm: the outgoing route is determined by looking-up entries in the routing table(s), in which each destination is associated to a netmask.

Mask	Value	Instruction
fff0000	c4f0000	MCRR
fe000000	fa000000	BLX (immediate)
fff8000	8bd8000	POP (ARM)
f000000	a000000	B
fe00010	800000	ADD (register, ARM)

Figure 5.4: Binary-encoded instruction is matched against the *mask*, expecting *value*.

5.2.2.2 Example: decoding the instruction `e0823003`

Consider the following binary-encoded instruction: `e0823003`. How does the hypervisor performs the instruction decoding? The relevant bits of the instruction are extracted using a binary **and**, then compared to their expected value. If the computed value matches the stored one, then it is a match, and the line entry is returned. Otherwise, the same computation is performed until a match is found, or the last entry is reached.

```
# Input : instr : binary-encoded instruction
# Output: The index of the first matching line
idx := -1
for each (mask,value,name) in match_table {
  idx++;
  if ((instr & mask) == value)
  return idx
}
```

Fig. 5.5 illustrates a factice lookup in the matching table.

The instruction `e0823003` is then, a ADD instruction. The table browsing algorithm is linear. It may seem inappropriate, but the following section explains why this is necessary.

instruction	mask	instruction & mask	expected value	match?	Instruction name
e0823003	fff0000	820000	c4f0000	✗	MCRR
e0823003	fe000000	e0000000	fa000000	✗	BLX (immediate)
e0823003	fff8000	820000	8bd8000	✗	POP (ARM)
e0823003	f000000	0	a000000	✗	B
e0823003	fe00010	800000	800000	✓	ADD (register, ARM)

Figure 5.5: The matching table is browsed linearly, until a match is found.

Instruction Set	Instruction count	# lines w.o UB	Overhead	# lines w/ UB	Overhead
ARM	307	976	317.92 %	2175	708.47 %
Thumb 16	78	98	125.64 %		
Thumb 32	279	1013	363.08 %		
Total (Thumb)	357	1111	311.20 %	1756	491.88 %

Table 5.1: Instruction set comparison between entry count in the specification and in the match table.

5.2.2.3 Scanner metrics

Because a matching-table is used, we provide some metrics to compare the size of the tables with regard to the size of the instruction set. The ARM documentation includes implementation defined and unpredictible instructions. Those categories are gathered into a larger "undefined behaviour" one. The following table (5.1) presents those numbers with and without the undefined behaviours (UB) entries:

Without undefined behaviour instructions, there are about three times more entries in the table than existing instructions. The actual table includes those unpredictable behaviour. In that case, the table is even larger: about seven times more entries for the ARM encoding, and almost five times for the Thumb encoding. Because the table is browsed linearly, it should be ordered by most frequent instructions. But this has not been done, because there might exist relations between several lines in the table: for instance instructions having `PC` as a target register are matched first. Hence, this order should be preserved. With the used encoding scheme, there is no easy way to implement this sort. Instead, a cache was used in the analyser, which holds the corresponding line entry in cache. At the beginning, no entries are filed. Upon execution, the cache contains the last destination address known for a source address. The cache entries are clean when the memory is writen. We have measured the cache hit to more than 90% in our benchmarks.

The next table (5.2) associates the instruction families previously described, with the number of instruction for each one.

Family	Entries	Rate (w/ UB)	Rate (w.o UB)
IT block start	1	0.06 %	0.24 %
Subrouting invocation	6	0.34 %	1.43 %
Changes CPU state	10	0.57 %	2.38 %
Raises exception	12	0.57 %	2.38 %
Reads *PSR	12	0.62 %	2.61 %
Unsupported	14	0.80 %	3.33 %
Explicit branch	26	1.48 %	6.18 %
Reads coprocessor	26	1.48 %	6.18 %
Writes coprocessor	26	1.53 %	6.41 %
Writes *PSR	29	1.59 %	6.65 %
Accesses hardware	30	1.70 %	7.13 %
Branches an odd way	53	3.01 %	12.59 %
Writes memory	62	3.52 %	14.73 %
Reads memory	117	6.65 %	27.79 %
Unpredictable	1339	76.08 %	
Total	1760		
Total without unpreditable	421		

Table 5.2: Instruction families count, and their associated rates. The second rate is computed without the unpredictable family.

The largest family is the unpredictable one. Without those entries, there would be much less instructions to match, but unpredictable behaviour could arise.

5.2.2.4 Finding the next sensitive instruction

As explained in the state of the art, because the ARM instruction set mixes code and data, scanning the whole binary by 32bits chunk is not realistic. Hence, the control flow is followed. Finding the next sensitive instruction is easy with the match-table and the control flow. It consists of applying the analysis of each instruction, and returning if a sensitive behavior is present in the table. But because the table is large, the analyser keeps a cache entry:

- At the beginning, the cache is empty, and the entry point is analysed (named *current_addr*);
- Following the control flow, the following instruction is scanned;
  - If the instruction is not sensitive, another iteration is performed;
  - otherwise, the current entry is returned, and a cache entry is append containing the current entry, associated with *current_addr*.

Care must be taken to flush the cache entries when a write access is performed on the concerned memory region.

5.3 Arbitration's implementation

This section provides further details on the dynamic part of the hypervisor. After statically matching an instruction, a trap is inserted in the guest code to switch back to the hypervisor. Afterward, a dynamic arbitration is performed, to decide whether to modify the current instruction or not.

5.3.1 Trap to the hypervisor: the callback mechanism

To switch back to the hypervisor, a trap mechanism must be implemented. The ARM architecture provides several way to do this:

Emit an hypercall (`HVC`): in a processor that implements the Virtualization Extensions, the HVC instruction causes a Hypervisor Call exception. This means that the processor mode changes to Hyp, the CPSR is saved to the SPSR_hyp, and execution branches to the HVC vector.

Emit a supervisor call (`SVC`): the SVC instruction causes the SVC exception.

Emit a breakpoint (`BKPT`): the BKPT instruction causes the processor to enter Debug state.

Craft an invalid instruction: this technique was used in the earliest version of KVM (presented in [30]).

Branching to the hypervisor (`B`): this instruction causes a branch to the provided address.

Loading an address into `PC`: same as branching, but oversteps the branch range.

Because the implementation does not require virtualization nor debug extensions, the `HVC` and `BPKT` cannot be used. `SVC` would cause an exception, and save `r13,r14`. The same holds when issuing an invalid instruction (which traps to the "undefined" entry in the interrupt vector). Because both the guest kernel and the hypervisor run in privileged mode, the current implementation uses a simple branch instruction to perform the hypercall. Since the hypervisor is further than 32Mb (the maximum value encodable in the `B` instruction) from the guest (see Fig. 5.8), the implementation uses the last item: loading a value into `PC`.

The trap mechanism (illustrated in Fig. 5.7) is implemented as follow (implementation in Fig. 5.6):

- Given an address *addr*, and a CPU mode (Thumb or ARM):
- The instruction stored at *addr* an *addr + instr_size* (depending on the CPU mode) are stored in the guest context;
- A branch to `PC-4` is crafted through an `LDR` instruction.

Uppon the trap, the hypervisor will restore the guest code with the stored instructions (Fig. 5.6).

Code

```
void arm_platform_patch_arm_8(addr_t at, addr_t dest) {
  uint32_t *ptr = (uint32_t *) at;
  *ptr++ = 0xE51FF004; // LDR PC, [PC, #-4]
  *ptr = dest;
}
```

Figure 5.6: The hypercall is implemented through a LDR instruction.

One can notice the *-4* offset in the crafted instruction. This is simply due to the pipeline stage. Also, an additional instruction is required to store the destination address.

A more graphical representation is given by Fig.5.7. As we can see, a single instruction is replaced by two instructions (actually, a load instruction and its corresponding value). Those two instructions are saved in the guest context, and restored before scheduling the guest on the processor.

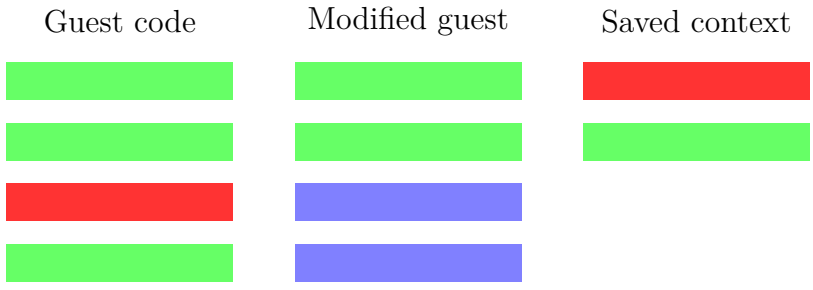


Figure 5.7: On the left side, the guest code before analysis. The red box is a sensitive instruction. After analysis, this instruction is replaced by a trap into the hypervisor (blue box). Two instructions are required because of the PC-relative encoding. That's why two instructions are saved.

5.3.2 Tracking indirect branches

The instructions that perform branch by an odd way are typically instructions which assign `PC` a dynamically computed value (using another register for instance). Trivial cases could have been handled with static analysis, for instance:

```
mov r0, #0x10EF3E4
mov pc, r0
```

Instead, because of the processor's complexity, we decided to let the actual processor perform the task. Hence, when an "odd-branch" instruction is reached, the destination register (`PC`) is replaced by another register (not used within the instruction), and that instruction is executed on the CPU, with the current guest's context. Upon execution, the hypervisor has the actual destination address into that register.

5.3.3 Condition validation

The same kind of instruction crafting is also performed to determine whether a conditional instruction will execute. A dummy instruction is created, and its conditional bits are set accordingly.

```
mov r0, #0
add<<COND>> r0, r0, #1
```

Upon executing, the conditional behaviour is determined by reading `r0`'s value: if `0` is read, the condition was not met, if `1` is read, then the condition was met.

5.3.4 Conditional blocks

IT (If-Then) instruction is a thumb instruction which makes up to four following instructions conditional (IT block). Otherwise, thumb mode does not have conditional execution. The hypervisor performs a complex analysis to handle this instruction:

1. Determine the size of the IT block;
2. For each instruction, store the conditional outcome;
3. Sets a trap, if needed, accordingly.

This instruction is defined as follows: `IT{x{y{z}}} {cond}`, where `x y` and `z` can be either `T` (then) or `E` (else) of `cond`. In addition, the assembler may warn about unpredictable instructions in an IT block like `B,BL` and `CPS` but also `BX,CBZ` and `RFE`.

5.3.5 Summary

The ARM instruction set is no longer RISC, and introduce interesting features to achieve performances on embedded system. But those features are often hard to catch and to analyse in software. The lack of specification also makes it hard to have confidence in the matching table, because it is human made. Releasing a computer-usable specification of the instruction-set would be a huge plus for this kind of work.

5.4 Example on a simple guest

5.4.1 description

Several guests were writen to evaluate the hypervisor. One of them was a user-space hello world: *test_user_mode_swi.c*. It is composed of 105 lignes of C code, and 35 lines of assembly. This guest does the following actions:

1. Setup a proper context
   - Switch to system mode
   - Setup a stack for user mode
   - Switchs back to supervisor mode
2. Setups interrupt vector (to handle SWI)
3. Switchs to user mode
4. Performs a home-made system call to trigger a print on UART

Down to assembly code, only 30 distinct instructions are used.

The following table exposes the number of entries for each instruction families aforementioned:

5.5 Summary

Contrary to what might be assumed, the Thumb encoding is not easier to handle. First, changes in the encoding must be tracked (basically ..X instructions). Then, the IT instruc-

Instruction	Count	Branches
add	1	0
bhi	1	0
bic	1	0
ldrls	1	1
lsr	1	0
mrs	1	0
orr	1	0
subs	1	0
uxtb	1	0
beq	2	2
cmp	2	0
cps	2	0
strb	2	0
sub	2	0
umull	2	0
ldrb	3	0
udf	3	0
ldm	4	4
pop	4	2
svc	4	0
bx	5	5
push	5	0
tst	6	0
bne	7	7
b	8	8
str	9	0
bl	11	11
mov	11	0
.word	22	0
ldr	30	2
Total	153	42

Family	entries
Subrouting invocation	11
Changes CPU state	10
Raises exception	12
Reads *PSR	12
Unsupported	14
Explicit branch	26
Reads coprocessor	26
Writes coprocessor	26
Writes *PSR	29
Accesses hardware	30
Branches an odd way	53
Writes memory	62
Reads memory	117
Unpredictable	1339
Total branching	42

tion makes it harder to follow the control flow. Finally, Thumb32 must also be encoded an analysed. As show in the metrics, there is as many overhead in the entrys for ARM than for Thumb instructions.

I could only get a PDF description of the instruction set. Hence, quite some time was needed to encode those instructions. Also, in practice, founder can implement features, but also add bugs. Hence, the ARM documentation is not enough, but the peripheral documentation was also needed. Sometimes, errata must also be consulted to avoid known error. Again, it would be suitable to have an electronic released form of these information, so that automatic extracting tools could be used. That way, altering features and publishing erratas could easily and reliably be included into such implementations.

On a blog of Linaro[1], Christoffer Dall states that "A primary goal in designing both hypervisor software and hardware support for virtualization is to reduce the frequency and cost of transitions as much as possible." Because of all the cases to be handled (tracking branches, undefined behaviours), a lot of traps are set, which makes the hypervisor rather slow. We have estimated that without smart improvements, a trap is set every three instructions. Hyperblocks should be as large as possible.

[1]urlhttp://www.linaro.org/blog/core-dump/on-the-performance-of-arm-virtualization/

Memory map

For information purpose, the memory map of our hypervisor is given here.

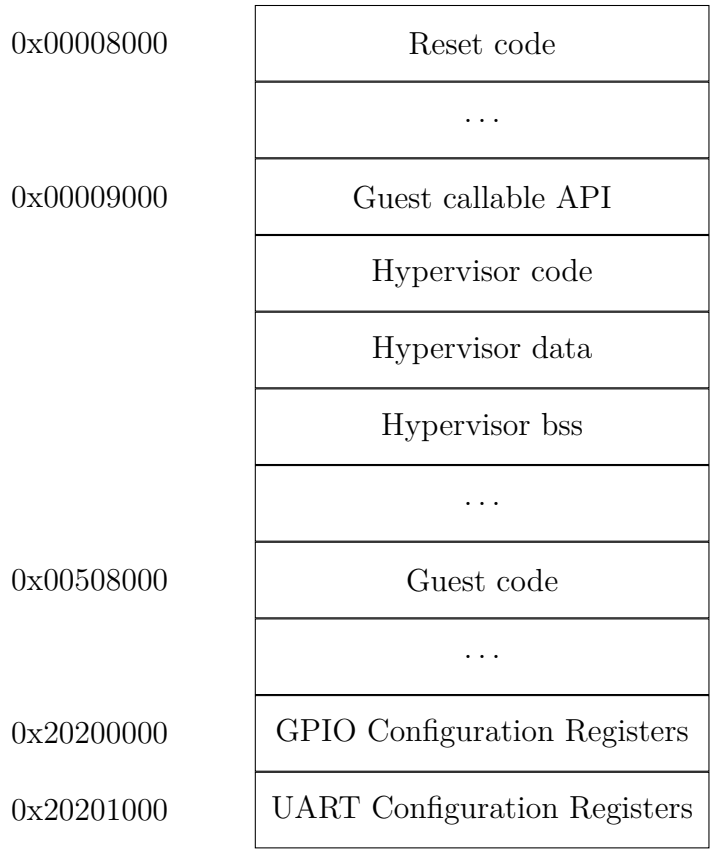


Figure 5.8: The hypervisor's memory map.

CHAPTER 6

Conclusion and future work

> And on the seventh day God finished his work that he had done, and he rested on the seventh day from all his work that he had done.
>
> *Genesis 2:2, English standard version*

Contents

6.1 Synthesis **112**
6.2 Future work **113**
6.3 Personal feedback **118**

6.1 Synthesis

This thesis examines the security guarantees that can be provided by certification. The most frequently used architectures such as ARM and x86 are not natively virtualizable. That is why current solutions tend to rely on hardware mechanisms which make these architectures virtualizable, by providing a dedicated privilege level for the hypervisor. This hypervisor will be invoked whenever the hardware requires it. This places a heavy responsibility on the proper functioning of these newly introduced hardware features ; assuming developers have correctly apprehended the extensive documentation, and specific erratas, which may or may not be freely available.

Based on the principle that this dependency on the hardware's feature is too strong, a new design was proposed to achieve virtualization with the minimal hardware requirements. It is believed that a more precise formalisation can be done by expressing all the instructions' specifications and behavior. The prototype is split in two layers:

- The first one analyses each guest's instruction keeping track of the execution mode. Either that instruction is benign, in which case the same analysis will be performed on the rest of the code, or it is juged to be sensitive, and a trap will be setup so that the second layer of the hypervisor will be invoked in lieu of that instruction being executed.
- The second one consists of deciding whether to execute an instruction or not, given the actual execution context of a guest. This allows the hypervisor to put itself in interposition with the underlying hardware, preventing the guest from doing uncontrolled operations.

In order to provide decent performance, the hypervisor will only hypervise privileged code. Assuming a correct protection of memory-mapped hardware, hardware can only be reconfigured in privileged mode. Letting unprivileged code execute unsupervised does not leverage security issues for the guest kernel and hence the hypervisor. Under those conditions, the measured overhead on actual hardware was around 10% in most cases, which is reasonable considering the simple design and the limited level of optimization applied on the prototype. This approch looks reasonable because most of the intensive tasks could be performed in user-space, by delegating hardware access to kernel tasks, as general purpose systems do.

Another evaluation was performed by evaluating the whole system; benchmarks where running in privileged mode. This use-case corresponds to more specific systems, such as firmware, in which no privilege separation is achieved. This scenario is much less effective. We have a two order of magnitude overhead, which is not practicable. Considering that the bottleneck is the frequent context-switch between guest and hypervisor, the architecture could be improved in order to gain performances. This can be achieved by two ways: the cost and the frequency of such context-switches. And because the frequency is a bigger

factor, that seems to be the first point to tackle.

To conclude, this thesis demonstrates that writing a tiny hypervisor is achievable. Having the smallest code should reduce the code complexity, which eases the certification process, at the expense of performances. Instead of optimizing the code of the hypervisor, we decided to rely on the kernel/user-space splitting principle, which tends to offload the kernel from the most computational intensive tasks. Security (integrity and confidentiality) can be enforced by a certified MMU.

This work has highlighted the challenges to virtualize an ARM processor, and resulted in 2 academic publications [118, 119] and several communication material (lab presentation and poster session).

6.2 Future work

This section presents the future work. Three perspectives are described here: increasing performances, discussing the expectations and expectations of the formal verification and finally, discuss how the security of the hypervisor should be challenged.

6.2.1 Increasing the hypervisor performances

To achieve that, I claim that hyperblocks should be as large as possible: currently they are a subset of basic blocks, but they should be a superset. The following sub-sections present some insights to accomplish that.

6.2.2 Reducing the context-switch cost

6.2.2.1 Reducing the cost of a single context-switch

Currently when a guest is switched, all its registers, cpsr, spsr are saved.

Operation	Memory load	Memory store	Copro load	Copro store
Saving context	0	15 + 2	2	0
Restoring context	15 + 2	0	0	2
Total	17	17	2	2

This makes 34 memory access instead of a single instruction. Assuming every instruction has the same cycle consumption, that makes a 34x performance overhead.

ARM IRQ bank 7 registers plus the SPSR. FIQ (Fast IrQ) only bank 2 registers plus the SPSR. Adding constraints on the register used by the hypervisor could let us reduce the number of register to save, thereby reducing the switch context cost.

Another solution, would be to use an alternative interrupt mechanism, such as SWI or crafting an undefined instruction. This would produce an exception, caught by the hypervisor. The hardware would have saved the guest context, and the hypervisor would just have to check whether that exception is an interruption or a call to the hypervisor.

Finally, some overhead might be expected on the MMU. Currently, no operation on the MMU is performed, because at this stage of development, the guest is assumed benevolent. Otherwise more overhead should be expected due to the TLB flush and MMU updates. But some MMUs provide an execute only permission, which would be the best case scenario.

6.2.2.2 Reducing the number of context-switches

The second axis to reduce the overhead of the hypervision mechanism is to avoid trapping. For instance, in the following code, the loop performs 43 iterations. The hypervisor will not let the branch take place, instead it will set the trap back to the hypervisor in order to track the next executed instruction. In that case, the body of the loop is harmless. The hypervisor could simply skip the loop, and perform the analysis after `0x30`.

Another idea would be to perform some computation on the guest binary off-card, or before starting the guest itself. Basically, that would consist in feeding the cache with initial data. Of course, great care must be taken regarding self modifying code. The guest code should be marked read-only and memory fault on writing should invalidate associated cache entries.

6.2.3 Expectations and foreseeable issues with a formal verification

There are two main components in the hypervisor: the instruction scanner, and the hypervisor itself. Given an address, the former determines whether the instruction located there is sensitive or not. Given a context (including trap's location) allows or forbids the instruction from being executed (with or without being modified).

The formal verification establishes a link between a model and an implementation. The model verifies some properties. The role of the verification process is to prove that the same properties also holds in the implementation. Those properties can be related to security or correctness. Hence, we expect two models: one for each main components. In both

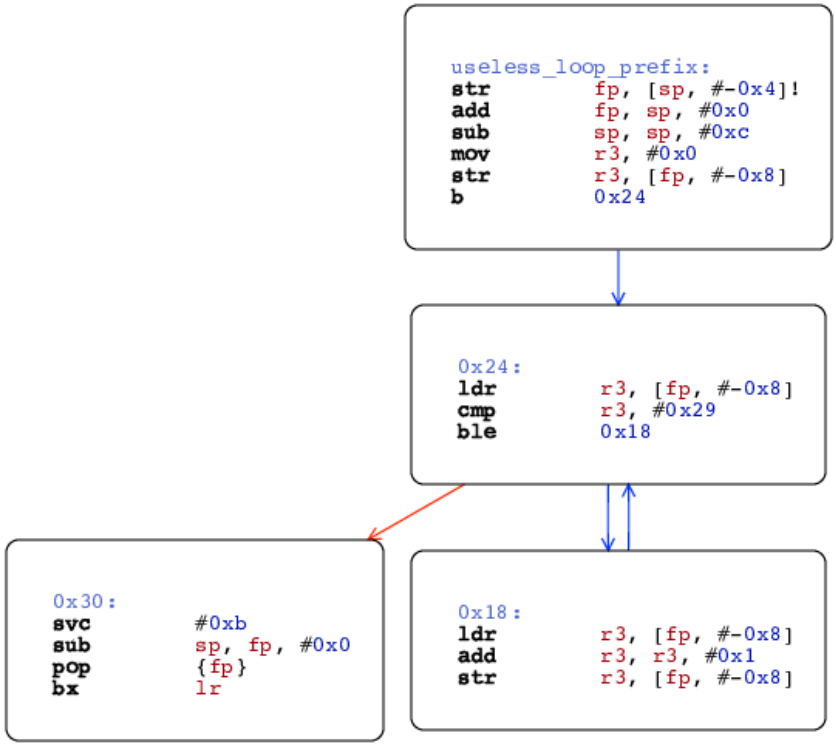


Figure 6.1: Because the loop contains no sensitive instructions, the hyperblock could include basic blocks number `0x24` and `0x18`.

cases, a difference should be made between security and precision. For instance, a scanner matching all the instructions as sensitive would be correct from a security point of view (a disaster for performances though) but it wouldn't be precise, because of false positives.

6.2.3.1 Verifying the instruction scanner

The first model would be provided by ARM, while the verification process would demonstrate that the instruction scanner properly decodes chunks of memory. That is not suitable in practice. Indeed, formal methods are efficient with a simpler model, where properties are verified. In this work, the documentation (which can be seen as the model) is more than four thousand pages. The complexity of the model is in the same order of magnitude than the implementation. This makes it very unsuitable for such work.

The only remaining solution would be exaustive tests. I have performed this kind of

test to validate the instruction matcher. The principle was to generate all the possible instructions from 0 to $2^{32} - 1$, and analyse each of them, and match the result of the match, with regard to the one expected. The obvious problem with this approach is that I did not use another matching table at that time. Another "database" should be used, such as the one from Unicorn Engine[1]. This framework is based on QEMU , and provides bindings in several languages such as Python which makes it really handy to perform code analysis or experimenting on small pieces of code. It should be noted that this framework did not exist at that time.

6.2.3.2 Verifying the hypervisor

After an instruction is marked as sensitive, it is replaced by a trap to the hypervisor. The main issue to verify for this component, is the following: if an instruction is allowed to run (be it modified or not), its behaviour will not break any other assumption. That is, no other behaviour should raise in runtinme, that was not expected during the analysis. This implies a precision-related verification, rather than security. Also, because the hypervisor does not scan the guest while running in user-mode, the verification must ensure that the sandbox remains robust. Hence, properties should be verified against an execution model, which would ensure that all the cases that can be used by the guest to escape the sandbox are properly handled.

6.2.4 Security assessment

Security is often described as the *CIA* triad: Confidentiality, Integrity and Availability. The confidentiality basically states that private data should remain private; the integrity is concerned by the non alteration of the data while the availability ensures that a service will always be responsive.

6.2.4.1 Designing security

Because this hypervisor is designed as a proxy between a single guest operating system and the underlying hardware, only those two subjects are to be considered. Hence, the study should no be concerned by non-interference with multiple guests. The security assessment will focus on the guest beeing the attacker, whereas the hypervisor is the target. The attack scenario is a malicious guest trying to escape its sandbox, or outperform the hypervisor. No hardware-based attacks are considered.

[1] https://www.unicorn-engine.org

Because the confidentiality and the integrity are dual, they rely on the same principle. Enforcing the confidentiality consists of preventing the guest from **reading** the hypervisor's memory, while enforcing the integrity consists of preventing the guest from **writing** the hypervisor's memory. The sealing of the hypervisor relies on a hardware mechanism: a MMU when available, but a MPU could also be considered. A minimum part of the hypervisor should be mapped on the guest address space: this is required to perform the callback on sensitive instructions. This piece of code should be sufficient to switch into the hypervisor context, and restore the appropriate mappings to make the code actually reachable.

On the other hand, the availability consists in preventing the hypervisor from crashing, or being subverted. This imply two things:

from the analyser point of view: if an instruction is marked innocuous, then no sensitive behaviour must be achievable through it;

from the hypervisor point of view: the composition of innocuous instructions (*i.e* inside an hyperblock) must not exhibit any sensitive behaviour.

Also, care must be taken when handling data from the guest, which might have been maliciously crafted to subvert the hypervisor. For example, memory locations, coprocessor numbers or operations should be validated.

6.2.4.2 Challenging security

Because the guest runs at the same level of privilege as the hypervisor, an attacker should not expect to increase its privileges. Instead, it would rather try to escape the hypervisor. Several scenarios can be imagined:

Abusing the hardware configuration: if a guest manages to access some hardware without the hypervisor conscent, the guest may obtain additional privileges. An actual example in Xen is XSA-182 (CVE-2016-6258), in which a malicious guest can leverage fast-path in the MMU validation code, and escalate their privilege to that of the host.

Abusing the hypervisors' algorithm: the hypervisor traps on branches, or sensitives instructions. If a guest manages to craft a sequence of instructions that seems legitimate for the hypervisor, it manage to make the hypervisor lose track. Another actual example from Xen is XSA-106 (CVE-2014-7156), in which the emulator misses some privileges checks on the emulation of software interrupts.

Abusing race conditions: when the hypervisor performs an analysis, it assumes that the memory of the guest will remain the same (otherwise, a trap should arise). If the guest manages to exploit a race condition, or modify its state on the behalf of the hypervisor, the latter may be fooled. For instance, if a guest branches to an address

A, which contains an innocuous instruction, the hypervisor will start an analysis from A, and place a trap further away. When the context is given back to the guest, if it manages to change the instruction at the address of A, it can escape the hypervisor. Those vulnerabilities are also common. For example, XSA-155 (CVE-2015-8550) exploits a double fetch in a shared memory region, and abuses it by changing the content of the memory between those two fetches.

Exploiting traditionnal vulnerabilities: such as buffer overflows, unsanitized user-inputs and so on. This family is not specific to hypervisor's security. Hence, no further details are provided here.

To leverage those bugs, one could reverse engineer the hypervisor, or simply have a look on the code. But the compiler may have introduced some changes between the C code and the binary machine code. Hence, for specific details (such as the double-fetch previously described) both approaches can be considered. Finally, fuzzying seems to be suitable for such scenario. Altering genuine guests could help detect some corner cases which could not be properly handled by the hypervisor. In practice, such approache is very effective. Recently, lcamtuf's AFL (American Fuzzy Loop) was modified to support full-system fuzzing using Qemu[2]. This project has already found several bugs: CVE-2016-4997 and CVE-2016-4998 in the Linux kernel *setsockopt* implementation.

6.3 Personal feedback

For this thesis to succeed, two dual-objectives had to be handled. The first one was that I was funded by a company. Hence, the objectives may not be the same than a public research institute. The cooperation was easy, but I had to put some distance between the research part and the company, in order to avoid interferences, especially in publications material. It was nice to be autonomous, but at the same time frustrating not to be part of engineering tasks. The second objective, was to conciliate two different worlds: formal methods and low level programming. In the academic world, it seems hard for people from different communities to work together, because they don't speak the same language, and don't have the same objectives. At Prove&Run, everybody want to make it work, in order to have an actual product. Hence, things worked really good. So in one case, the need of an "immediate application" driven by the economy is suitable, because it makes people work together. On the other hand, it creates issues to share information, because of classified materials.

After quite some time spent experimenting on operating systems and bare-metal programs for x86, I was given a slightly variation of my objectives: work on an ARM based hypervisor, without virtualization extensions. It must be held that very few documentation

[2] https://github.com/nccgroup/TriforceAFL

was available on the subject, but ARM being a Reduced Instruction Set Computing with approximately 200 instructions, it shouldn't be too hard to decode each instructions. It was wrong. Or at least, not as easy as it should be. On older ARM encoding schemes, the first bits were dedicated to the condition encoding, the following to the instruction class (like arithmetic operations, memory based and so on), and the latest for the operands. That's no longer the case. Not in the general case anyway. There were a lot of particularities which forced me to abandon my switch-case in favor of a matching instruction table. All the instructions had to be encoded at once, rather than encoding only the ones assumed to be sensitive.

In the beginning, my ambition was to boot a Linux kernel on the VMM. Linux is a nice piece of (complex) software. The early stage of the kernel decompress itself, to another memory location, jumps at that location, and starts messing up with the MMU. Moreover, at that point I discoverd that Thumb code was actually used. This made me discover the If-Then instructions, which can conditionally execute the following one, two or three instruction if the condition is either true or false. After a lot of time spent on those issues, the kernel could decompress, but it took approximately 15 minutes on top of the hypervisor, for about 1sec on the CPU. It was decided that a smaller target should be considered. Nevertheless, this gave us confidence in our matching table: to dynamically verify the correctness of the operating performed on the guest context, I have written a concurent debugging platform, which execute concurently Linux on top of Qemu, and Linux on top of the hypervisor, itself on top of Qemu, both commanded by gdb. The native case was stepping on each instruction, whereas the virtualized one was stepped until the next instruction, in order to let the hypervisor do the work without. It helped us (Pierre, an intern) and I to solve numerous bugs on the VMM, and the match table, and validate the proper handling of IT blocks. Afterward, work was done on FreeRTOS. First conclusion: don't rush into something too large, things, and split the work on realistic steps. Maybe a PoC on MSP430 or MIPS and a tiny operating system would have been better to start with. Second conclusion: x86 is said to be bloated, from my point of view, so are armv6 and armv7. It seems that armv8 tends to come back to the roots, and simplify some design elements. I hope it will get a better acceptance than Intel Itanium architecture, which had the same goals.

I do have regrets though. I used to find the system community hard to please: you need a good idea, implement it, get benefits from the published results and try to get accepted by a community which is English driven, and where (often) most accepted papers are american. In their result, they state a couple years of work, with several active contributors. How should (let's say it) French PhD students, which are supposed to get a diploma within three years achieve a comparable amount of work?

To conclude on a more positive tone, this thesis gave me the opportunity to get an expertise on low-level programming, which was a daily challenge. I had to deal with a wide spectrum of software to get things done, which was very educational. The work, was

interesting and my colleagues were very nice. If I was in the same situation again, I would most probably enroll again!

Acronyms

CC Common Criteria.
CPL Current Privilege Level.
CPU Central Processor Unit.

DMA Direct Memory Access.
DPL Descriptor Privilege Level.

EAL Evaluation Assurance Level.
EL Execution Level.

GP General Protection (fault).
GPR General Purpose Registers.

ISA Instruction Set Architecture.

JVM Java Virtual Machine.

MMU Memory Management Unit.

OS Operating System.

PAAS Platform As A Service.
PP Protection Profile.

RPL Requested Privilege Level.

SAAS Software As A Service.
SFP Security Functional Policy.

SFR Security Functional Requirements.
SMC Secure Monitor Call.
ST Security Target.

TLB Translation Lookaside Buffer.
TOE Target Of Evaluation.

VFS Virtual File System.
VMM Virtual Machine Monitor.

Bibliography

[1] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005. 2.1.2.1, 2.1.3.2, 4.3.3

[2] "Bochs the open source ia-32 emulation project." http://bochs.sourceforge.net/. Accessed: 2016-05-13. 2.1.2.2

[3] "QEMU open source processor emulator." http://wiki.qemu.org/Main_Page. Accessed: 2016-05-13. 2.1.2.2

[4] "Valgrind's homepage." http://www.valgrind.org/. Accessed: 2016-05-13. 2.1.2.2

[5] "Em86's homepage." http://ftp.dreamtime.org/pub/linux/Linux-Alpha/em86/v0.4/docs/e Accessed: 2016-05-13. 2.1.2.2

[6] K. P. Lawton, "Bochs: A portable pc emulator for unix/x," *Linux Journal*, vol. 1996, no. 29es, p. 7, 1996. 2.1.2.2

[7] "Qemu internals." http://qemu.weilnetz.de/qemu-tech.html. Accessed: 2016-05-13. 2.1.2.2

[8] F. Bellard, "Qemu, a fast and portable dynamic translator.," 2005. 2.1.2.2, 2.2.1.1

[9] G. D. Knott, "A proposal for certain process management and intercommunication primitives," *SIGOPS Oper. Syst. Rev.*, vol. 8, pp. 7–44, Oct. 1974. 2.1.2.3

[10] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. 2.1.2.3

[11] Intel, Santa Clara, CA, USA, *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, June 2013. 2.1.2.3

[12] W. R. Stevens, *TCP/IP Illustrated (Vol. 1): The Protocols*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993. 2.1.3.1

[13] J. Spolsky, *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*, ch. The Law of Leaky Abstractions, pp. 197–202. Berkeley, CA: Apress, 2004. 2.1.3.1

[14] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, pp. 32–38, May 2005. 2.1.3.3

[15] T. Olausson and M. Johansson, "Java-past, current and future trends," 2006. 2.1.3.3

[16] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. 2.1.3.3

[17] B. Venners, *Inside the Java Virtual Machine*. New York, NY, USA: McGraw-Hill, Inc., 1996. 2.1.3.3

[18] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st ed., 2014. 2.1.3.3

[19] R. P. Goldberg, "Architecture of virtual machines," in *Proceedings of the Workshop on Virtual Computer Systems*, (New York, NY, USA), pp. 74–112, ACM, 1973. 2.2.1.1

[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 164–177, Oct. 2003. 2.2.1.1, 2.2.4.1, 4.1.6

[21] A. Velte and T. Velte, *Microsoft Virtualization with Hyper-V*. New York, NY, USA: McGraw-Hill, Inc., 1 ed., 2010. 2.2.1.1

[22] C. A. Waldspurger, "Memory resource management in vmware esx server," Dec. 2002. 2.2.1.1

[23] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, pp. 225–230, 2007. 2.2.1.1

[24] "bhyve - the bsd hypervisor." Accessed: 2016-05-13. 2.2.1.1

[25] I. VirtualBox, "The virtualbox architecture," 2008. 2.2.1.1

[26] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, pp. 412–421, July 1974. 2.2.1.1

[27] J. S. Robin and C. E. Irvine, "Analysis of the intel pentium's ability to support a secure virtual machine monitor," in *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9*, SSYM'00, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2000. 2.2.1.2

[28] N. Penneman, D. Kudinskas, A. Rawsthorne, B. De Sutter, and K. De Bosschere, "Formal virtualization requirements for the arm architecture," *J. Syst. Archit.*, vol. 59, pp. 144–154, Mar. 2013. 2.2.1.3

[29] A. Suzuki and S. Oikawa, "Implementing a simple trap and emulate vmm for the arm architecture," in *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, vol. 1, pp. 371–379, Aug 2011. 2.2.1.3, 2.2.4.2

[30] C. Dall and J. Nieh, "Kvm for arm," 2010. 2.2.1.3, 2.2.4, 2.2.4.1, 5.3.1

[31] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, pp. 97–113, June 2003. 2.2.2.1

[32] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," *Commun. ACM*, vol. 36, pp. 69–81, Feb. 1993. 2.2.2.2

[33] W. S. Juan Rubio, "Binary-to-binary translation literature survey," tech. rep., University of Texas at Austin, 1998. 2.2.2.2

[34] A. Kelley, "Statically recompiling nes games into native executables with llvm and go." http://andrewkelley.me/post/jamulator.html. Accessed: 2016-06-08. 2.2.2.2

[35] "Virtualization extensions - arm." http://www.arm.com/products/processors/technologies/v
Accessed: 2016-06-10. 2.2.3

[36] P. Varanasi and G. Heiser, "Hardware-supported virtualization on arm," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, (New York, NY, USA), pp. 11:1–11:5, ACM, 2011. 2.2.3.2

[37] C. Dall and J. Nieh, "Kvm/arm: the design and implementation of the linux arm hypervisor," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 333–348, 2014. 2.2.3.2

[38] "Understanding Full Virtualization, Paravirtualization, and Hardware Assist," *VMware White Paper*. 2.2.4.1

[39] "The xen project: the powerful open source industry standard for virtualization." http://www.xenproject.org/. Accessed: 2016-06-09. 2.2.4.1

[40] J.-Y. Hwang, S. bum Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim, "Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones," in *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pp. 257–261, Jan 2008. 2.2.4.1

[41] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 95–103, July 2008. 2.2.4.1

[42] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*. Upper Saddle River, NJ, USA: Prentice Hall Press, first ed., 2007. 2.2.4.1, 3.1.4

[43] M. Rosenblum, "The reincarnation of virtual machines," *Queue*, vol. 2, pp. 34–40, July 2004. 2.2.4.1

[44] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 2–13, ACM, 2006. 2.2.4.2

[45] A. Inc, "Hypervisor framework reference." https://developer.apple.com/library/mac/docume Accessed: 2016-06-10. 2.2.4.3

[46] P.-H. Kamp and R. N. Watson, "Jails: Confining the omnipotent root," in *Proceedings of the 2nd International SANE Conference*, vol. 43, p. 116, 2000. 2.2.4.3

[47] T. Heo, "Cgroups documentation in linux kernel." https://www.kernel.org/doc/Documentation/cgroup-v2.txt. Accessed: 2016-06-10. 2.2.4.3

[48] J. H. (Microsoft), "Windows subsystem for linux overview." https://blogs.msdn.microsoft.com/wsl/2016/04/22/windows-subsystem-for-linux-over Accessed: 2016-06-10. 2.2.4.3

[49] J. H. (Microsoft), "Windows subsystem for linux, the underlying technology enabling the windows subsystem for linux." https://blogs.msdn.microsoft.com/wsl/2016/06/08/wsl-system-calls/. Accessed: 2016-06-10. 2.2.4.3

[50] K. Beckers, I. Côté, S. Fenz, D. Hatebur, and M. Heisel, *A Structured Comparison of Security Standards*, pp. 1–34. Cham: Springer International Publishing, 2014. 2.3.1

[51] The Common Criteria Recognition Agreement Members, "Common Criteria for Information Technology Security Evaluation." http://www.commoncriteriaportal.org/ccra, Sept. 2006. 2.3.2.1

[52] "Common critieria: New cc portal." http://www.commoncriteriaportal.org/. Accessed: 2016-06-14. 2.3.2.1

[53] "Common Criteria for Information Technology Security Evaluation Part 1: Introduction and general model," tech. rep., July 2009. 2.3.2.1

[54] "Common Criteria for Information Technology Security Evaluation Part 2: Security functional components," tech. rep., July 2009. 2.3.2.1

[55] tech. rep. 2.3.2.1

[56] M. Vetterling, G. Wimmel, and A. Wisspeintner, "Secure systems development based on the common criteria: The palme project," *SIGSOFT Softw. Eng. Notes*, vol. 27, pp. 129–138, Nov. 2002. 2.3.2.1

[57] tech. rep. 2.3.2.4

[58] "Too many cooks - exploiting the internet-of-tr-069-things." https://events.ccc.de/congress/2014/Fahrplan/events/6166.html. 3.1.1

[59] J. Rutkowska and R. Wojtczuk, "Qubes os architecture," *Invisible Things Lab Tech Rep*, p. 54, 2010. 3.1.2

[60] J. Pelzl, M. Wolf, and T. Wollinger, "Virtualization technologies for cars," tech. rep., Technical Report 2008, escrypt GmbH-Embedded Security, 2008. 3.1.2

[61] M. Broy, S. Ramesh, M. Satpathy, and S. Resmerita, "Cross-layer analysis, testing and verification of automotive control software," in *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pp. 263–272, IEEE, 2011. 3.1.2

[62] I. inc, "Intel sgx homepage." https://software.intel.com/en-us/sgx, 2016. 3.1.2

[63] I. Anati, S. Gueron, S. Johnson, and V. 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*, vol. 13, 2013. 3.1.2

[64] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. 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, (New York, NY, USA), pp. 10:1–10:1, ACM, 2013. 3.1.2

[65] A. Ltd, "Trustzone arm homepage." http://www.arm.com/products/processors/technologies 3.1.2

[66] V. Pratt, "Anatomy of the pentium bug," in *TAPSOFT '95: Theory and Practice of Software Development* (P. Mosses, M. Nielsen, and M. Schwartzbach, eds.), vol. 915 of *Lecture Notes in Computer Science*, pp. 97–107, Springer Berlin Heidelberg, 1995. 3.1.2, 3.3.2.3

[67] L. Duflot, "Cpu bugs, cpu backdoors and consequences on security," *Journal in computer virology*, vol. 5, no. 2, pp. 91–104, 2009. 3.1.2

[68] A. Canteaut, C. Lauradoux, and A. Seznec, "Understanding cache attacks," Research Report RR-5881, INRIA, 2006. 3.1.2

[69] D. J. Bernstein, "Cache-timing attacks on aes," tech. rep., 2005. 3.1.2

[70] D. A. Osvik, A. Shamir, and E. Tromer, *Cache Attacks and Countermeasures: The Case of AES*, pp. 1–20. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. 3.1.2

[71] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: A timing attack on openssl constant time RSA," in *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pp. 346–367, 2016. 3.1.2

[72] M. S. I. B. G. G. I. T. E. B. Sunar, "Cache attacks enable bulk key recovery on the cloud." Cryptology ePrint Archive, Report 2016/596, 2016. http://eprint.iacr.org/2016/596. 3.1.2

[73] L. Duflot, Y.-A. Perez, and B. Morin, "What if you can't trust your network card?," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, (Berlin, Heidelberg), pp. 378–397, Springer-Verlag, 2011. 3.1.3

[74] Z. Gu and Q. Zhao, "A state-of-the-art survey on real-time issues in embedded systems virtualization," 2012. 3.1.4

[75] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," in *11th Real-Time Linux Workshop*, pp. 263–272, Citeseer, 2009. 3.1.4

[76] D. Reinhardt and G. Morgan, "An embedded hypervisor for safety-relevant automotive e/e-systems," in *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pp. 189–198, June 2014. 3.1.4

[77] U. Steinberg and B. Kauer, "Nova: A microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, (New York, NY, USA), pp. 209–222, ACM, 2010. 3.1.4, 4.1.6

[78] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an operating system kernel," *Communications of the ACM*, vol. 53, pp. 107–115, jun 2010. 3.1.4, 3.3.1.2

[79] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg, "The performance of micro-kernel-based systems," *SIGOPS Oper. Syst. Rev.*, vol. 31, pp. 66–77, Oct. 1997. 3.1.4

[80] "Sqlite home page." https://www.sqlite.org/. Accessed: 2016-06-23. 3.3.1.1

[81] "How sqlite is tested." https://www.sqlite.org/testing.html. Accessed: 2016-06-23. 3.3.1.1

[82] "Trust-in-soft: tis interpreter." http://trust-in-soft.com/tis-interpreter/. Accessed: 2016-06-23. 3.3.1.1

[83] 3.3.1.1

[84] ISO, "The ANSI C standard (C99)," Tech. Rep. WG14 N1124, ISO/IEC, 1999. 3.3.1.1

[85] S. Lescuyer, "Provencore: Towards a verified isolation micro-kernel," *HiPEAC 2015 (High Performance Embedded Architectures and Compilers)*, 2015. 3.3.1.2

[86] M. Browne, E. Clarke, D. Dill, and B. Mishra, "Automatic verification of sequential circuits using temporal logic," *Computers, IEEE Transactions on*, vol. C-35, no. 12, pp. 1035–1044, 1986. 3.3.1.2.1

[87] G. Bochmann, "Hardware specification with temporal logic: An example," *Computers, IEEE Transactions on*, vol. C-31, no. 3, pp. 223–231, 1982. 3.3.1.2.1

[88] K. McMillan, "Interpolation and sat-based model checking," in *Computer Aided Verification* (J. Hunt, WarrenA. and F. Somenzi, eds.), vol. 2725 of *Lecture Notes in Computer Science*, pp. 1–13, Springer Berlin Heidelberg, 2003. 3.3.1.2.1

[89] E. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *Tools for Practical Software Verification* (B. Meyer and M. Nordio, eds.), vol. 7682 of *Lecture Notes in Computer Science*, pp. 1–30, Springer Berlin Heidelberg, 2012. 3.3.1.2.1

[90] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West, "Model checking an entire linux distribution for security violations," in *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, (Washington, DC, USA), pp. 13–22, IEEE Computer Society, 2005. 3.3.1.2.1

[91] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000. 3.3.1.2.1

[92] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, (New York, NY, USA), pp. 238–252, ACM, 1977. 3.3.1.2.2

[93] J. Souyris and D. Delmas, "Experimental assessment of astrée on safety-critical avionics software," in *Computer Safety, Reliability, and Security* (F. Saglietti and N. Oster, eds.), vol. 4680 of *Lecture Notes in Computer Science*, pp. 479–490, Springer Berlin Heidelberg, 2007. 3.3.1.2.2

[94] D. Delmas and J. Souyris, "Astrée: From research to industry," in *Static Analysis* (H. Nielson and G. Filé, eds.), vol. 4634 of *Lecture Notes in Computer Science*, pp. 437–451, Springer Berlin Heidelberg, 2007. 3.3.1.2.2

[95] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, and M. Turin, "Space software validation using abstract interpretation," in *Proc. of the Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA 2009)*, vol. SP-669, (Istambul, Turkey), pp. 1–7, ESA, May 2009. 3.3.1.2.2

[96] C. Ferdinand and R. Heckmann, *aiT: Worst-Case Execution Time Prediction by Static Program Analysis*, pp. 377–383. Boston, MA: Springer US, 2004. 3.3.1.2.2

[97] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969. 3.3.1.2.3

[98] H. Geuvers, "Proof assistants: History, ideas and future," *Sadhana*, vol. 34, no. 1, pp. 3–25, 2009. 3.3.1.2.3

[99] X. Leroy, "Formal verification of a realistic compiler," *Communications- ACM*, vol. 52, pp. 107–115, July 2009. 3.3.2.2

[100] M. Dillon, "archive from kernel@crater.dragonflybsd.org: Buildworld loop segfault update – i believe it is hardware." http://bochs.sourceforge.net/. Accessed: 2016-06-27. 3.3.2.3

[101] M. Dillon, "archive from kernel@crater.dragonflybsd.org: Amd cpu bug update – amd confirms!." http://article.gmane.org/gmane.os.dragonfly-bsd.kernel/14518. Accessed: 2016-06-27. 3.3.2.3

[102] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: A timing attack on openssl constant time rsa." Cryptology ePrint Archive, Report 2016/224, 2016. http://eprint.iacr.org/2016/224. 3.3.2.3

[103] W. R. Tereshkin, A. 3.3.2.3

[104] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pp. 104–113, Springer-Verlag. 3.3.2.3

[105] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," pp. 388–397. 3.3.2.3

[106] J.-J. Quisquater and D. Samyde, "ElectroMagnetic analysis (EMA): Measures and counter-measures for smart cards," in *Smart Card Programming and Security* (I. Attali and T. Jensen, eds.), vol. 2140, pp. 200–210, Springer Berlin Heidelberg. 3.3.2.3

[107] C. Eagle, *The IDA Pro book*. No Starch Press, Inc., 2nd ed., 2011. 4.1

[108] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Trans. Comput. Syst.*, vol. 32, pp. 2:1–2:70, Feb. 2014. 4.1.6.3

[109] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Commun. ACM*, vol. 27, pp. 1013–1030, Oct. 1984. 4.2.1

[110] ARM, "mbed tls implementation." https://tls.mbed.org/. 4.2.1

[111] "Certification report of the memory management unit des microcontrollers samsung s3ft9kf/ s3ft9kt/ s3ft9ks, rev 1." https://www.commoncriteriaportal.org/files/epfiles/ANSSI-CC-2013_12_fr.pdf. Accessed: 2016-06-09. 4.2.4

[112] "Security target of the memory management unit des microcontrollers samsung s3ft9kf/ s3ft9kt/ s3ft9ks, rev 1." https://www.commoncriteriaportal.org/files/epfiles/anssi-cc-cible2013_12.pdf. Accessed: 2016-07-16. 4.2.4

[113] J. R. Santos, G. Janakiraman, and Y. Turner, "Xen network i/o performance analysis and opportunities for improvement," *HP Labs*, 2007. 4.2.4

[114] "Erika : Embedded real time kernel architecture." http://erika.tuxfamily.org/drupal/erika-educational.html. Accessed: 2016-07-16. 4.3.3

[115] D. Deville, A. Galland, G. Grimaud, and S. Jean, "Smart card operating systems: Past, present and future," in *In Proceedings of the 5 th NORDU/USENIX Conference*, 2003. 4.3.3

[116] A. Sloss, D. Symes, and C. Wright, *ARM System Developer's Guide: Designing and Optimizing System Software*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. 5.1.1

[117] ARM, *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*. ARM, 2012. 5.1.3.5, 5.2.2.1

[118] G. G. François Serman, Michael Hauspie, "Achieving virtualization trustworthiness using software mechanisms," 2016. 6.1

[119] G. G. François Serman, Michael Hauspie, "Hypervision logiciel et défiance matériel," 2016. 6.1