

# Towards Trustworthy Virtualisation: Improving the Trusted Virtual Infrastructure

Carl Gebhardt

Technical Report  
RHUL-MA-2011-10  
17 March 2011



Department of Mathematics  
Royal Holloway, University of London  
Egham, Surrey TW20 0EX, England

<http://www.rhul.ac.uk/mathematics/techreports>

# **Towards Trustworthy Virtualisation: Improving the Trusted Virtual Infrastructure**

Carl Gebhardt

Thesis submitted to the University of London  
for the degree of Doctor of Philosophy

Information Security Group  
Department of Mathematics  
Royal Holloway, University of London

October 2010

# Declaration

These doctoral studies were conducted under the supervision of Dr. Allan Tomlinson. The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Mathematics as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Carl Gebhardt  
October, 2010

To everyone who believed in me.

# Acknowledgements

My heartfelt thanks to my supervisor Dr. Allan Tomlinson for his guidance, encouragement, patience and dedication over the past three years. I found in you an excellent supervisor and also a good friend.

I would like to also thank my examiners, Professor Chris Mitchell and Professor Andrew Martin for taking the time to examine my thesis.

I gratefully acknowledge the financial support of the ISG and Professor Keith Martin in helping to organise funding. I would like to thank my fellow post-graduate students in the Information Security Group for many helpful discussions. I must also thank Professor Kenny Paterson for his support and guidance. I am also extremely grateful to Adam Davison and Laura Nequest for proof-reading my thesis.

Additionally, I would like to thank Chris I. Dalton, Richard Brown, Boris Balacheff and the rest of the System Security Lab for fruitful collaborations during and beyond my time at Hewlett-Packard's System Security Lab.

Finally, I would like to thank my family, and especially my parents, for their support and encouragement throughout my education. Thank you for allowing me to be a free spirit and giving me the freedom to pursue my own ideas and goals, even if they seemed dangerous at times. I would also like to thank my girlfriend Amanda Castillo for her patience and support during so many late nights.

# Abstract

Modern commodity platforms have become easy targets, which are increasingly plagued by malware exploiting legacy design weaknesses. Malware often abuses the large and feature-rich computing base, which forms the basis of modern commodity systems and inherently has to be trusted. In recent years, research has suggested employing machine virtualisation technology to provide isolation where commodity systems fail to do so. On one hand, hardware machine virtualisation support on commodity systems is a very recent technology, and with its novel technology allows for new creative security solutions. On the other hand, machine virtualisation changes many previous security assumptions about a platform and therefore creates new challenges itself.

*This thesis investigates machine virtualisation and trusted computing technology and outlines how those technologies could be utilised to move towards a more trustworthy virtualisation infrastructure.*

To achieve this, the thesis has been divided into three main parts. In the first part of this thesis, we describe how the hypervisor's Trusted Computing Base could be reduced and, with new hardware advances, could be further strengthened. To achieve this, we reassess the definition of the Trusted Computing Base and illustrate how segregation of different code blocks could be enforced by hardware protection mechanisms.

In the second part, we propose a novel scheme to protect the integrity and confidentiality of storage in a virtualised infrastructure. We discuss the implementation of a prototype for a secure, flexible and transparent virtual disk image. We base our concepts on trusted computing, utilising the Trusted Platform Module to efficiently deliver integrity assurance to virtual disk images, as well as enabling the owner to retain control over the disk image throughout its life-cycle.

In the third part, we present a flexible architecture that enables a platform user to benefit from the advantages of a fast booting system and a full-featured mainstream Operating System at the same time. The prototype builds on newly available machine virtualisation and trusted hardware features increasingly available on commodity systems. Moreover, this design enhances the concept of an instant-on system with secure, trustworthy and policy enforced compartments.

In this thesis, we find that a sensible trusted virtualisation layer requires more protection guarantees than simply the combination of Trusted Computing and virtualisation building blocks. We therefore start with the basic foundations to increase the trustworthiness of the lower hypervisor level; in the second part we build up on the previous layer to provide trusted storage in a virtualised environment. The final part embraces the preceding concepts and combines the latest hardware machine virtualisation and trust technologies to deliver a more robust virtualisation infrastructure.

# Contents

<b>1</b>	<b>Introduction</b>	<b>18</b>
1.1	Motivation . . . . .	18
1.2	Contribution . . . . .	20
1.3	Thesis Organisation . . . . .	21
1.4	List of Publications . . . . .	23
<b>I</b>	<b>Background</b>	<b>24</b>
<b>2</b>	<b>Virtualisation</b>	<b>25</b>
2.1	Introduction . . . . .	26
2.2	Historical Overview . . . . .	26
2.3	Taxonomy . . . . .	29
2.3.1	Emulation, Simulation and Virtualisation . . . . .	31
2.3.2	Formal Requirements . . . . .	32
2.3.3	Hypervisor and Virtual Machine Monitor . . . . .	33
2.4	Machine Virtualisation . . . . .	35
2.4.1	Intel x86 Architecture . . . . .	36
2.4.2	Hypervisor Implementations . . . . .	38
2.4.3	Memory Virtualisation . . . . .	40
2.4.4	Device and I/O Virtualisation . . . . .	41
2.4.5	Summary . . . . .	44
2.5	Security Discussion . . . . .	45
2.6	Summary . . . . .	48
<b>3</b>	<b>Trusted Platforms</b>	<b>49</b>
3.1	Introduction . . . . .	50
3.2	Trusted Computing . . . . .	50
3.2.1	Historical Overview . . . . .	51
3.2.2	Trusted Platform . . . . .	52

3.2.3	Trusted Computing Base . . . . .	54
3.2.4	Trusted Platform Module . . . . .	54
3.2.5	TPM Functional Overview . . . . .	61
3.2.6	Static Root of Trust for Measurement . . . . .	62
3.2.7	Dynamic Root of Trust for Measurement . . . . .	64
3.3	Discussion . . . . .	67
3.4	Summary . . . . .	71
<b>II</b>	<b>Problem Definition and Related Work</b>	<b>72</b>
<b>4</b>	<b>Problem Definition</b>	<b>73</b>
4.1	Trusted Virtualisation . . . . .	74
4.1.1	Motivation . . . . .	74
4.1.2	Requirements . . . . .	76
4.2	Challenges . . . . .	78
4.2.1	Isolation Issues . . . . .	78
4.2.2	Trusting the Hypervisor . . . . .	80
4.2.3	Management Domain . . . . .	81
4.2.4	Platform Limitations . . . . .	83
4.2.5	I/O Device Sharing . . . . .	87
4.2.6	Virtualising the TPM . . . . .	88
4.3	Discussion . . . . .	91
4.4	Related Work . . . . .	94
4.4.1	Virtualisation and Isolation . . . . .	94
4.4.2	Trusted Computing . . . . .	97
4.5	Summary . . . . .	99
<b>III</b>	<b>Improving the Trusted Virtual Infrastructure</b>	<b>100</b>
<b>5</b>	<b>Separating Trusted Computing Base with Hardware</b>	<b>101</b>
5.1	Introduction . . . . .	102
5.2	Motivation . . . . .	103
5.3	Background . . . . .	104
5.3.1	Trusted Computing Base . . . . .	104
5.3.2	Ring Protection . . . . .	105
5.4	Trusted Computing Base for HVMs . . . . .	108
5.4.1	HVM Protection Rings . . . . .	108
5.4.2	Code Separation . . . . .	109

5.5	Usage Scenarios . . . . .	111
5.5.1	Inter VM Communication . . . . .	112
5.5.2	Virtual Private Networks . . . . .	113
5.5.3	Legacy Virtualisation . . . . .	114
5.5.4	vTPM Implementation . . . . .	115
5.5.5	ACPI . . . . .	116
5.5.6	Policy Control . . . . .	117
5.6	Considerations . . . . .	118
5.6.1	Performance . . . . .	118
5.6.2	Development Effort . . . . .	118
5.6.3	Device Sharing . . . . .	119
5.7	Discussion . . . . .	119
5.8	Summary . . . . .	120
<b>6</b>	<b>Trusted Virtual Disk Images</b>	<b>121</b>
6.1	Introduction . . . . .	122
6.2	Motivation . . . . .	123
6.3	Security Requirements . . . . .	124
6.3.1	Threats . . . . .	124
6.3.2	Design Goals . . . . .	125
6.4	Trusted Virtual Disk Images . . . . .	126
6.4.1	Assumptions . . . . .	127
6.4.2	Ensuring Confidentiality . . . . .	128
6.4.3	Integrity Protection . . . . .	129
6.4.4	Creating Integrity Metrics . . . . .	131
6.4.5	Policy Model . . . . .	135
6.4.6	Software Enforcement . . . . .	135
6.4.7	Metafile . . . . .	136
6.5	Life Cycle . . . . .	139
6.5.1	Initialisation . . . . .	139
6.5.2	Deletion . . . . .	139
6.5.3	Backup . . . . .	140
6.5.4	Sparse Format . . . . .	140
6.5.5	Migration . . . . .	141
6.5.6	Snapshots . . . . .	141
6.6	Security Analysis . . . . .	142
6.7	Considerations . . . . .	145
6.7.1	Fragmentation . . . . .	145

6.7.2	Performance	146
6.7.3	Swap-space	146
6.8	Usage Scenario	146
6.9	Discussion	147
6.10	Summary	148
<b>7</b>	<b>LaLa: A Late Launch Application</b>	<b>149</b>
7.1	Introduction	150
7.2	Motivation	151
7.2.1	Design Goals	152
7.3	Background	153
7.3.1	Trusted Execution Technology	153
7.3.2	Launch Control Policies	156
7.3.3	OpenVZ	157
7.4	Implementation Details	158
7.4.1	Requirements and Limitations	160
7.4.2	Hardware Platform	160
7.4.3	Instant-on OS	161
7.4.4	The Late Launch Process	162
7.4.5	OVZ Virtual Machine	163
7.4.6	Trusted Management Domain	165
7.4.7	Environment Teardown	167
7.5	Security Analysis	168
7.6	Discussion	172
7.7	Summary	174
<b>IV</b>	<b>Conclusion</b>	<b>175</b>
<b>8</b>	<b>Summary and Conclusion</b>	<b>176</b>
8.1	Summary	176
8.2	Conclusion	180
8.3	Directions for Future Work	181
8.3.1	Runtime Integrity	181
8.3.2	Inter VM Communication	183
	<b>Bibliography</b>	<b>185</b>

# List of Abbreviations

ABI	Application Binary Interface
ACM	Authentication Code Module
ACPI	Advanced Configuration and Power Interface
AIK	Attestation Identity Key
AML	ACPI Machine Language
AMT	Active Management Technology
API	Application Programming Interface
APM	Advanced Power Management
ASL	ACPI Specification Language
BIOS	Basic Input Output System
BSP	Bootstrap Processor
CA	Certificate Authority
CMK	Certifiable Migratable Key
CMS	Conversational Monitor System
CP	Control Program
CPL	Current Privilege Level
CPU	Central Processing Unit
CRTM	Core Root of Trust for Measurement
CR< <i>n</i> >	Control Register < <i>n</i> >
CSC	Cambridge Scientific Centre
CTSS	Compatible Time-Sharing System
DAA	Direct Anonymous Attestation
DAC	Discretionary Access Control
DEV	Device Exclusion Vector
DLL	Dynamic Link Library
DMA	Direct Memory Access
DRTM	Dynamic Root of Trust for Measurement
DSL	Damn Small Linux
EEPROM	Electrically Erasable Programmable Read-Only Memory

EK	Endorsement Key
EMSCB	European Multilaterally Secure Computing Base
EPTs	Extended Page Tables
GDT	Global Descriptor Table
gPTs	guest Page Tables
gTCB	guest Trusted Computing Base
HAL	Hardware Abstraction Layer
HLVM	High-level Language Virtual Machine
hPTs	hardware-assisted Page Tables
hTCB	hypervisor Trusted Computing Base
HVM	Hardware Virtual Machine
I/O	Input/Output
ID	Identification
IDE	Integrated Drive Electronics
IDT	Interrupt Descriptor Table
ILP	Initiating Logical Processor
IOMMU	Input Output Memory Management Unit
IPC	Inter Process Communication
ISA	Instruction Set Architecture
iTPM	integrated TPM
IVMC	Inter VM Communication
KVM	Kernel-based Virtual Machine
LCP	Launch Control Policy
LPC	Low Pin Count
MA	Migration Authority
MAC	Mandatory Access Control
MIT	Massachusetts Institute of Technology
MLE	Measured Launch Environment
MLS	Multi-Level Security
MMU	Memory Management Unit
NGSCB	Next Generation Secure Computing Base
NPTs	Nested Page Tables
NSA	National Security Agency
NV	Non-Volatile
NX	No eXecute
OOB	Out-Of-Bound
OpenTC	Open Trusted Computing
OS	Operating System

OSPM	OS Power Management
OVZ	OpenVZ
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCR	Platform Configuration Register
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PPC	PowerPC
PTs	Page Tables
RIM	Reference Integrity Metric
RNG	Random Number Generator
RSA	Algorithm for public-key cryptography named after Rivest, Shamir and Adleman
RTM	Root of Trust for Measuring
RTR	Root of Trust for Reporting
RTS	Root of Trust for Storage
SCP	Secure Copy
SHA	Secure Hash Algorithm
SL	Secure Loader
SLB	Secure Loader Block
SLOC	Source Lines of Code
SMI	System Management Interrupt
SML	Stored Measurement Log
SMM	System Management Mode
SMRAMC	SMRAM Control (register)
SMX	Safer Mode Extension
sPTs	shadow Page Tables
SR-IOV	Single Root I/O Virtualisation
SRAM	Static Random Access Memory
SRK	Storage Root Key
SRTM	Static Root of Trust Measurement
SSH	Secure Shell
STM	SMM Transfer Module
SVM	Secure Virtual Machine
TC	Trusted Computing
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCPA	Trusted Computing Platform Alliance

TIS	TPM Interface Specification
TLB	Translation Lookaside Buffer
TMD	Trusted Management Domain
TP	Trusted Platform
TPM	Trusted Platform Module
TTP	Trusted Third Party
TVDI	Trusted Virtual Disk Image
TXT	Trusted Execution Technology
VDI	Virtual Disk Image
VM	Virtual Machine
VMCB	Virtual Machine Control Block
VMCS	Virtual Machine Control Structure
VMM	Virtual Machine Monitor
VMX	Virtual Machine Extension
vPCR	virtual PCR
VPN	Virtual Private Network
vTPM	virtual TPM

# List of Figures

2.1	Computer system architecture, based on [246]. . . . .	30
2.2	Type 1 hypervisor, based on [1]. . . . .	34
2.3	Type 2 hypervisor, based on [1]. . . . .	35
2.4	Hybrid hypervisor, based on [1]. . . . .	35
2.5	Normal ring protection scheme. . . . .	37
2.6	Binary translation. . . . .	38
2.7	Para-virtualisation. . . . .	38
2.8	Three levels of memory virtualisation. . . . .	40
2.9	Shadow page table management. . . . .	42
3.1	Trusted Platform Module design, based on [103]. . . . .	55
3.2	Static Root of Trust for Measurement. . . . .	63
3.3	Dynamic Root of Trust for Measurement. . . . .	65
4.1	Simplified picture of a trusted virtualisation architecture. . . . .	75
4.2	Current virtualisation trust boundaries. . . . .	83
4.3	Optimistic view on virtualisation trust boundaries. . . . .	84
5.1	Traditional ring protection scheme. . . . .	105
5.2	Intended ring use, according to Grawrock [103]. . . . .	106
5.3	Ring deprivilging. . . . .	106
5.4	VMX root mode/non-root mode. . . . .	109
5.5	Conceptual overview. . . . .	111
6.1	Sample VDI implementation in XEN, based on [25]. . . . .	127
6.2	Sample TVDI implementation in XEN, based on [52]. . . . .	130
6.3	Overview of integrity construction. . . . .	131
6.4	Creating integrity execution flow. . . . .	133
7.1	TXT initialisation phase. . . . .	154
7.2	TXT policy flow, based on [121]. . . . .	156

7.3	Step (1), measured launch. . . . .	162
7.4	Step (2), live migration process. . . . .	164
7.5	Step (3), launch full-feature OS. . . . .	165
7.6	Policy Enforcement and Decision Points. . . . .	167

# List of Tables

6.1	Actors, libraries and actions involved in TVDI operation. . . . .	129
6.2	Metafile properties. . . . .	137

# Chapter 1

## Introduction

### Contents

---

<b>1.1</b>	<b>Motivation</b>	<b>18</b>
<b>1.2</b>	<b>Contribution</b>	<b>20</b>
<b>1.3</b>	<b>Thesis Organisation</b>	<b>21</b>
<b>1.4</b>	<b>List of Publications</b>	<b>23</b>

---

*Sometimes, in fact, the role of the attacker is to invent that which has not been thought of.*

– Bev Littlewood

### 1.1 Motivation

Machine virtualisation is not a new technology, but has experienced a resurgence of interest amongst industry and academia over the last decade. While Machine virtualisation was originally designed to partition expensive mainframe hardware, in recent years its focus has shifted to provide multiple execution environments on commodity systems. As a consequence, different solutions have emerged to accommodate machine virtualisation on the most commonly used commodity platform, the x86 architecture. Platforms are frequently chosen based on the availability of key applications which often constitute legacy applications. While the commercial success of machine virtualisation is based to some extent on the ability to execute legacy Operating Systems (OS), its containment capability is an interesting property

## 1.1 Motivation

---

with regard to secure system research. There are, as we will outline in the following chapters, a number of security issues associated with machine virtualisation and modern commodity platforms.

Firstly, traditional Operating Systems only offer poor isolation guarantees and are vulnerable to a large number of attacks. As applications rely on the assurances provided by the underlying Operating System, the model of building secure applications on top of such insecure systems seems flawed. There is [197, 239, 241, 251] and has for some time been [28, 137, 193, 284], a strong movement to provide more secure services in the form of more secure Operating Systems. Unfortunately, many secure Operating Systems break legacy concerns as well as being notoriously hard to configure and manage [26]. Moreover, as pointed out by Karger et al. [136] during their reflection on the lessons learned from the first Multics security evaluation in the early 1970s, many of the vulnerabilities identified 40 years ago are still present in modern Operating Systems today, including buffer overflows, memory segmentation and ineffective use of hardware features such as No eXecution (NX) flags.

Additionally, the size of many modern Operating Systems has uncontrollably grown in size and are often enriched with sophisticated features. Complexity, however, is often held responsible as the single biggest issue in software security [47], whereas the amount of vulnerabilities is considered proportional to code size [185]. Thus the complex and feature-rich structure of today's Operating Systems results in a large Trusted Computing Base, which renders security guarantees or code evaluation virtually impossible.

Nevertheless, the recent resurgence of interest and advances in machine virtualisation technology, have created a window of opportunity for new security solutions: a virtualised platform allows the concurrent execution of multiple Operating Systems at different trust levels. The basic hardware interfaces of machine virtualisation, allow for a much simpler and stronger control structure. As a consequence, secure and insecure systems can coexist while relying on a much smaller code base for isolation. Therefore machine virtualisation potentially offers to bridge the gap between backward compatibility and more modern systems without compromising user experiences. There exist many different types and variants of virtualisation that are unified under the virtualisation umbrella, for simplicity reasons we will use the term virtualisation when referring to machine virtualisation in the remainder of this thesis.

## 1.2 Contribution

---

On one hand, the Trusted Computing initiative as formed by the Trusted Computing Group (TCG), aims to increase the resistance of commodity platforms to software-based attacks [260]. Trusted Computing (TC), enables a local or remote party to assess the hardware and software state of a platform in order to evaluate the trustworthiness of that particular platform. Thus, a Trusted Platform (TP), is able to reliably prove its current state, which effectively allows a remote party to make trust decisions about that platform. Current implementation of Trusted Platforms on commodity hardware however, lack the ability to strongly isolate trusted from untrusted components. Thus, in terms of trustworthiness the evidence they provide is difficult to assess. This shortcoming is mostly due to the lack of sufficient hardware and software protection capabilities.

On the other hand, although virtualisation is able to strongly isolate complete Operating Systems from each other, it lacks the facilities to provide authentication to remote entities. TC can potentially complement virtualisation to provide such functionality; however security is a holistic approach and virtualisation as well as Trusted Computing are only basic building blocks. Additionally, with the recent enhancements of hardware support for virtualisation as well as trust technology in modern platforms, CPU manufacturers are increasingly willing to dedicate chip real estate in order to provide security functions.

**Thesis Statement.** *Virtualisation allows commodity platforms to gradually move towards a more secure execution environment, whilst at the same time satisfying legacy concerns and preserving user experience. This thesis investigates the combination of virtualisation, Trusted Computing technology as well as recent hardware advances on commodity platforms to propose an approach and mechanisms whereby these technologies could be utilised to move towards a more trustworthy virtualisation infrastructure.*

## 1.2 Contribution

Following a detailed examination of state of the art virtualisation and trust technologies, we conclude that a sensible trustworthy virtual infrastructure requires more protection mechanisms than each of the technologies are able to provide. Based on an analysis of past secure system research, we present the requirements for a trusted virtual infrastructure as well as outlining platform specific limitations. In order to construct a trusted virtual framework we investigate the security properties of the

### 1.3 Thesis Organisation

---

trusted virtualisation building blocks at various architectural layers. We therefore begin with the hypervisor layer in Chapter 5, followed by the storage layer in Chapter 6 and we finish on an application layer in Chapter 7. Consequently, the main contribution of this thesis can be summarised as follows:

- We propose to utilise the currently unused CPU protection mechanisms, in order to deconstruct and improve on a hypervisor's Trusted Computing Base.
- We propose a novel scheme for Trusted Virtual Disk Images as a means of transparently providing integrity and confidentiality to Virtual Machine storage systems.
- We introduce an unique approach that combines the latest hardware virtualisation and trust technologies to segregate Operating Systems at different trust levels. Our approach preserves existing user experiences and at the same time addresses tighter security requirements.

### 1.3 Thesis Organisation

The remainder of this thesis is divided into three distinctive parts and organised as follows:

**Part I - Background:** The first part of this thesis is comprised of Chapter 2 and Chapter 3, which provide the technical basis for the remainder of the thesis. Chapter 2 provides a historical overview of the evolution of virtualisation, followed by the classification of various different types of virtualisation, with a definition of the terminology used throughout the remaining parts of the thesis. This chapter also provides an introduction to the many technical challenges which must be solved to enable virtualisation on commodity systems. Finally, the chapter concludes with a security discussion on state of the art virtualisation technology and concepts.

Chapter 3 introduces the notions of Trusted Computing and of Trusted Platforms. We first provide a historical overview and further discuss the technical implementation of, as well as the requirements for, a Trusted Platform. We then introduce new trust technology features, which have recently become

### 1.3 Thesis Organisation

---

available on commodity systems. Moreover, we identify and discuss limitations of the current Trusted Computing model on commodity platforms.

**Part II - Problem Definition and Related Work:** The second part provides an overview of the motivation behind creating a trusted virtualisation layer. Moreover, we identify the basic properties which are required for trusted virtualisation as well as outlining a set of challenges based on the previous technical discussion. These challenges lead us to formulate the problem definition which we address in the subsequent chapters. Additionally, we identify related work which overlaps in goals or motivation.

**Part III - Improving the Trusted Virtual Infrastructure:** In this part of the thesis we investigate how the Trusted Virtual Infrastructure can be improved with regard to the previously defined requirements. The solutions we discuss in Part III, represent one of, potentially, many different ways to improve on trusted virtualisation. However, we have based our concept and design on recent technologies available to us in order to mitigate current platform weaknesses.

In Chapter 5 we therefore demonstrate how the Trusted Code Base of a hypervisor could be reduced to provide a more meaningful attestation as well as better isolation guarantees. We describe how the Trusted Computing Base of a hypervisor can be separated with existing hardware features. Consequently, we reassess the existing definition of Trusted Computing Base and apply the terminology to a layered hypervisor design as well as to unused hardware protection mechanisms. We discuss various application scenarios which arise from the unused protection mechanisms and show how their application could help to mitigate security threats in a virtualised system. Moreover, we thoroughly discuss the shortcomings of our proposal, as well as comparing the proposal to relevant work in this area.

Chapter 6 highlights how the integrity and confidentiality of a virtual disk image can be maintained throughout its life-cycle. We demonstrate the motivation, goals and principles behind our Trusted Virtual Disk Image (TVDI) design, following the discussion of the security requirements and challenges for disk images. We discuss the design implementation in detail, based on the XEN para-virtualisation model, and follow up with a discussion on the typical life-cycle of a TVDI. Subsequently, we provide a security analysis, founded on our threat model defined in Chapter 6.

In Chapter 7 we exploit new hardware trust technologies, and together with the building blocks investigated in the preceding chapters, construct a more

## 1.4 List of Publications

---

trustworthy virtual infrastructure. We then investigate how Operating Systems at different trust levels can coexist and how segregation of corporate and private Virtual Machines could be achieved without compromising user experiences. We describe our implementation in detail as well as identifying its application scenarios. Moreover, we outline the constraints of the current hardware platforms as well as software limitations. In the subsequent security discussion we consider past and present attacks on the new hardware trust technologies, as well as outlining our adversary model. Our analysis of this work shows that our prototype is very practical compared to related research.

**Part IV - Conclusion:** In this chapter we review the requirements identified in Chapter 4. We then reflect on our contribution that is intended to improve the trusted virtual infrastructure based on the defined requirements. Finally, we conclude the thesis and outline possible directions for future work.

## 1.4 List of Publications

This thesis contains material that was previously published with Allan Tomlinson [92, 93, 94, 95], material published with Chris I. Dalton [89], Chris I. Dalton and Richard Brown [90] as well as material published with Chris I. Dalton and Allan Tomlinson [91].

These publications form the basis of this thesis as follows:

Chapter 4 is based on the previous findings in [93] as well as [90]. The content has significantly evolved over time and has accordingly been updated for this thesis. The basis for the work in Chapter 5 has been published in [91] and [95]. The concepts presented in Chapter 6 have previously been published in [94] and [92], and the content of [89] forms the foundation for Chapter 7.

The ideas discussed in Chapter 7 as well as the resulting proof-of-concept prototype were created by myself under the supervision of Chris I. Dalton and Richard Brown during the course of a six month internship at the Hewlett-Packard Labs, Bristol. Moreover, the work described in Chapter 7 has currently a patent pending [56].

## Part I

# Background

## Chapter 2

# Virtualisation

### Contents

---

<b>2.1</b>	<b>Introduction</b>	<b>26</b>
<b>2.2</b>	<b>Historical Overview</b>	<b>26</b>
<b>2.3</b>	<b>Taxonomy</b>	<b>29</b>
2.3.1	Emulation, Simulation and Virtualisation	31
2.3.2	Formal Requirements	32
2.3.3	Hypervisor and Virtual Machine Monitor	33
<b>2.4</b>	<b>Machine Virtualisation</b>	<b>35</b>
2.4.1	Intel x86 Architecture	36
2.4.2	Hypervisor Implementations	38
2.4.3	Memory Virtualisation	40
2.4.4	Device and I/O Virtualisation	41
2.4.5	Summary	44
<b>2.5</b>	<b>Security Discussion</b>	<b>45</b>
<b>2.6</b>	<b>Summary</b>	<b>48</b>

---

*Today's PC mistakes were made on the mainframe in 1967 and fixed by 1968.*

– Greg Price, in 2004

## 2.1 Introduction

---

### 2.1 Introduction

The most common virtual machines are actually Operating Systems (OSs) themselves. They are so ubiquitous that their role as a virtual machine provider is often overlooked. Today's user processes are still executed that it seems they exclusively control the processor as well as all available resources. In reality, each process has its own, separated resource context, which has been allocated by an OS. The OS then multiplexes or time shares the underlying resources and schedules processes using a fixed set of algorithms. As outlined in this chapter, this concept is almost identical to that of virtualisation, with the exception of the layer in which the abstraction is placed.

The ideas of time sharing a computer system, the abstraction from hardware resources and virtualisation have been investigated as early as the 1950s. Virtualisation and its associated spin-off technologies have experienced a resurgence of interest in recent years and are currently a hot topic in both the academic as well as the commercial world.

In the following section, we will define the terminology and context that is required to understand this thesis.

### 2.2 Historical Overview

In the early mainframe era, computer hardware was very expensive, rare and only accessible by a few experts. In addition, the available resources were shared by a large number of different users, who often had conflicting requirements. This could be, for instance, a different programming language and also an incompatible OS. To overcome this issue and to increase utilisation of the expensive mainframe infrastructure, the idea of a time based sharing scheme was formed [249]. Examples were Massachusetts Institute of Technology's (MIT) Compatible Time-Sharing System (CTSS) and its successor, Multics, which led the way to the development of Unix [231]. Some of the basic concepts of those early OSs, such as segmentation, paging and multiplexed execution are still present in today's OSs.

## 2.2 Historical Overview

---

During the development of the CTSS, MIT encountered difficulties that required modification of the underlying hardware. CTSS was developed on a series of IBM machines, and after implementing the proposed enhancements, the resulting IBM 7090 design became the model of all time-sharing OSs [271]. Hardware modifications consisted of a second set of 32K words, an address relocation register to realise virtual memory and memory protection. At the same time, IBM had been concurrently working on their System 360 (S/360) commercial model family, which was later shipped without address translation functionality.

Development of CTSS was transferred to MIT's "project MAC", which was spawned in 1963. The project's goal was to further investigate and develop time-sharing concepts and applications. Project MAC later morphed into the MIT Laboratories for Computer Science. At the same time IBM initiated its Cambridge Scientific Centre (CSC), which would make their existing S/360 series more suitable for relevant customers such as MIT. However, the new S/360 shipped without address translation and was later turned down by MIT [271]. This development made two important impacts: firstly, MIT's researchers were forced to use an alternative platform for their future development of Multics; secondly, IBM initiated their own time-sharing research. During the mid 1960s, IBM hosted the M44/44X research project, which was an experimental evaluation of the emerging time-sharing systems at that time. The term *Virtual Machine* (VM) originated from this project: the main IBM 7044 (M44) machine was capable of running multiple copies of the 7044 as *virtual machines* (44x) [196].

In 1965, IBM's CSC developed the so-called CP/CMS on the basis of CTSS. The Control Program (CP) was a basic control program which allowed the system to run multiple concurrent copies of the Cambridge Monitor System or Conversational Monitor System (CMS). The earliest version of CP was CP-40, which could run up to 14 instances of the CMS, a simple single-user time-sharing OS [179]. The CP-40 led to the CP-67 and CP-370, and was later reimplemented to become a commercial success as the IBM VM/370 [3, 55]. IBM's popular z/VM is a direct successor of the VM/370, which is still in use today.

During the 1980s, Personal Computers (PCs) became affordable for the general public and were dominated by multi-user OSs and later by single-user OSs such as DOS. In the late 1980s and early 1990s, PCs became so affordable that the need for sharing vanished almost completely. Virtualisation, as a technology, became a niche market and a research domain only.

## 2.2 Historical Overview

---

The recent renaissance of virtualisation is mostly driven by the commercial desire to run legacy applications and OSs on incompatible hardware or software. For example, virtualisation allows the parallel execution of incompatible OSs on the same platform, such as MacOS, Linux and Windows family OSs. Connectix was amongst the early adopters of virtualisation technology, and released virtualPC 1.0 for the Apple Macintosh in 1996. VirtualPC was designed to allow the execution of Microsoft Windows family OSs on Apple's MacOS platforms. MacOS at this time built upon the PowerPC (PPC), while Microsoft used the Intel IA-32 x86 architecture. Even though VirtualPC was an emulator, it was amongst the first commercially available products to highlight the importance of virtualisation to commodity platforms. Connectix was later acquired by Microsoft in 2003 [180].

The WINE [53] project, on the other hand was initiated to allow the execution of Windows applications on Linux. Created in 1993, the WINE project introduced a compatibility layer, simulating a Windows Application Programming Interface (API) on Linux. Therefore, it allows a set of applications originally written for Windows to be executed on a WINE compatible architecture.

Stanford University hosted project Disco in 1997 [40]. The main goal of Disco was to execute the Silicon Graphics' IRIX commodity OS on scalable multiprocessors, in particular cache coherent Non-Uniform Memory Architecture (ccNUMA) systems. Disco was the first to provide many of the mechanisms that are still present in today's virtualisation technology, such as a virtual CPU, virtual memory and virtual I/O devices.

In 1998, VMware was founded as a spin-off from Stanford University. In the same year VMware filed a patent, which describes how dynamic binary translation could be used to virtualise Intel's x86 hardware architecture [63]. The concept of binary translation had been previously proposed by the IBM DAISY system [72] to execute PPC as well as x86 systems on very long instruction word (VLIW) systems. However, in 1999 VMware introduced their VMware Virtual Platform, which is often considered to be the first commercial x86 virtualisation product [274].

The Denali [283] project in 2002 was amongst the first research projects which investigated virtualisation for its isolation properties. Denali was revolutionary in two ways: firstly, in contrast to previous virtualisation approaches, Denali was designed to be an isolation kernel and not designed to execute unmodified legacy OSs.

## 2.3 Taxonomy

---

Secondly, as the OS of VMs had to be modified, it also introduced a new virtualised interface which was incompatible with the previous Application Binary Interface (ABI).

Already in 1973 Madnick et al. [163] investigated how virtualisation could be used to isolate co-resident OSs. In recent years, many researchers have begun to argue that the isolation properties of virtualisation are beneficial for the overall security of a platform [49, 115]. However, Garfinkel et al. [88] as well as Arcre et al. [17] expressed a different view and highlighted the new risks introduced with virtualisation. Consequently, we will conclude this chapter with a discussion of the current state of virtualisation security in Section 2.5.

The benefits of virtualisation and reasons to use or avoid virtualisation are manifold and have been widely discussed commercially [181, 257, 276, 277] as well as in academic publications [49, 88, 143, 184, 248] but are beyond the scope of this thesis.

## 2.3 Taxonomy

There exist many different types and variants of virtualisation that are unified under the virtualisation umbrella. As a result, we need to define the terminology used in the remainder of this thesis and to confine non-relevant areas. A loose definition could define virtualisation as a means of translating or aggregating different resources, providing a unified view of the underlying resources. However, from an operational point of view, virtualisation maps its own interfaces onto the interfaces of the underlying resources, in essence, providing another layer of abstraction. In contrast to abstraction, however, virtualisation does not necessarily hide or simplify implementation [246]. Consequently, any arbitrary resource might be virtualised.

From a conceptual point of view however, the virtualisation infrastructure provides the underlying basic building blocks and services, which enable the use and management of virtualisation for an organisation or system. In the context of this thesis, we want to focus on virtualising the physical computer (machine) resources. Figure 2.1 outlines the typical computer architecture interfaces and the position of various abstraction layers important for virtualisation. The many varying machine virtualisation types differ in the interface they provide to a resource consumer:

## 2.3 Taxonomy

---

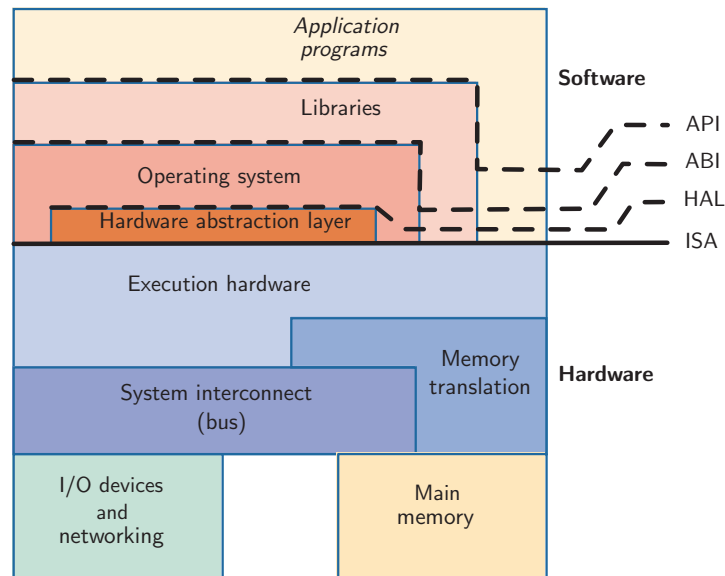


Figure 2.1: Computer system architecture, based on [246].

**Language-based VM** Language-based high-level VM (HLVM) implements predefined virtual hardware. Every target platform must implement and support the execution of byte-code written for this virtual hardware. HLVMs are usually geared towards platform portability and not performance. However, to increase performance, the HLVM code might be translated to native machine code for the host platform in real time. Well known examples of HLVM are Microsoft Common Language Infrastructure [183] and Sun Microsystems Java virtual machine [272].

**Process and kernel VM** Similar to the process isolation provided by a standard OS, process and kernel virtualisation executes in the same OS context. In process and kernel virtualisation the interfaces for the VM are provided by either a library (API) or the OS kernel (ABI) itself. Process or kernel isolation execute in the same kernel environment and address space, but are separated by different name spaces. Representatives are BSD-style JAIL [131], Linux-based OpenVZ [198] and Windows-based FVM [293].

**Co-designed VM** The hardware/software world is segregated by yet another interface, the Instruction Set Architecture (ISA). The ISA contains native processor commands or opcodes, which in turn are interpreted by the underlying processor implementation. Processors may export a common ISA but implementation of the micro-architecture might differ: this is the case for Intel's

## 2.3 Taxonomy

---

Itanium (IA-64) [240] and the Transmeta Crusoe [59] architectures. This form of virtualisation is also referred to as co-designed VMs [246].

**Physical and logical VM** An alternative form of virtualisation can be achieved by either physically or logically partitioning available resources. In physical partitioning, hardware resources are redundant and dedicated to a particular VM. In logical partitioning, hardware is time multiplexed to all VMs. Physical and logical partitioning, however, is mainly common on mainframes and requires low-level firmware to manage the resource partitioning [35]. Examples include, the successor of the IBM System/370, the IBM ESA/390, and subsequently the IBM System z/Series [117].

**System VM** The predominant virtualisation technology on commodity systems, as of today, is machine or full virtualisation. Machine virtualisation provides the same hardware interfaces to multiple VMs, and thus creates an illusion of multiple physical machines. In the context of this thesis, we want to focus on machine virtualisation and consequently need to define the term machine:

**Definition 1.** *A machine is a complete execution environment, consisting of CPU, memory, storage and I/O resources.*

Accordingly, a Virtual Machine (VM) consists of those virtual resources and thus provides the basic execution environment for an OS.

**Definition 2.** *A virtual machine is a virtual execution environment, consisting of a virtual CPU, virtual memory, virtual storage and virtual I/O resources.*

The domain that resides inside a VM is subsequently called a *guest*, while a *host* provides the environment.

Machine virtualisation docks onto the Hardware Abstraction Layer (HAL). Therefore, both the VM and the host share the same ISA. The intended purpose of the HAL is to allow portability and hardware independence for the majority of the OS kernel. In the following sections we will discuss how virtualising on a HAL level can be used to efficiently implement machine virtualisation.

### 2.3.1 Emulation, Simulation and Virtualisation

It might be difficult if not impossible for an ordinary user to distinguish between virtualisation, emulation and simulation; however the key difference is the means of

## 2.3 Taxonomy

---

their implementation. Virtualised environments are executed on the same ISA [246]. Emulation however, imitates a different ISA, which allows it to run machine code that was not written for this specific architecture. The imitation of the foreign ISA has to be complete and precise, including hardware bugs, to allow the original code or firmware to run on the emulated hardware. Examples of emulation are MAME [164] and Bochs [34].

While emulation imitates hardware, simulation imitates a software environment. Simulation follows a pre-programmed model: a pre-defined input triggers a specific pre-defined output. The output could be realised following the easiest or least complex implementation, as it is not bound to hardware or software compatibility. Common software implementations of simulations are SimOS [219] and Simics [273].

### 2.3.2 Formal Requirements

In 1974 Popek & Goldberg defined in their paper *Formal Requirements for Virtualizable Third Generation Architectures* [209], the prerequisites for a computer architecture to support virtualisation. For a VM, three properties of a control program are of particular interest:

**Equivalence property** - All programs executed under the control program must behave identically when running directly on a physical machine.

**Resource control property** - The control program must be in exclusive control over hardware resources.

**Efficiency property** - Most instructions are executed directly on the hardware, with no intervention at all by the control program.

Consequently, Popek & Goldberg conclude, “A virtual machine is taken to be an efficient, isolated duplicate of the real machine.” Apart from defining the properties of the control program, they also define and classify the ISA requirements of a physical machine to support virtualisation:

**Privileged instructions** can only be executed if the processor is in a supervisor state, or privileged mode. Privileged instructions will cause a trap to the privileged mode, if executed with insufficient privileges.

## 2.3 Taxonomy

---

**Control sensitive instructions** are those which affect the resource configuration of a platform.

**Behaviour sensitive instructions** are those instructions whose behaviour depends on the configuration of resources.

Based on their findings, Popek & Goldberg proposed the following theorems for a virtualisable architecture:

**Theorem 1.** *For any conventional third generation computer<sup>1</sup>, 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.*

**Theorem 2.** *A conventional third generation computer is recursively virtualizable if it is: (a) virtualizable, and (b) a VMM without any timing dependencies can be constructed for it.*

### 2.3.3 Hypervisor and Virtual Machine Monitor

All machine virtualisation technologies feature some form of control program or structure. The earliest terminology used for this control structure was *supervisor*. The expression supervisor derives from the privileged control program, which resided in the extra bank of memory of the modified IBM 7090 [271]. The supervisor program was in charge of the hardware, managing interrupts as well as handling the system's inputs and outputs. With the CP-40 system, which implemented a supervisor program for every CMS, the term *hypervisor* evolved among IBM engineers [271, 292].

Waldspurger wrote in 2002: “A hypervisor is a virtualization platform that allows multiple operating systems to run simultaneously on a single host computer [278].” On the other hand, Robert Goldberg [100] coined the term ‘Virtual Machine Monitor’ in 1974: “In these systems, much of the software for the simulated machine executes directly on the hardware without software interpretation. Systems of this kind are called virtual machine systems, the simulated machines are called virtual machines (VMs), and the simulator is called the virtual machine monitor (VMM).”

The VMM is also referred to as hypervisor if it is executed on the hardware directly and hosted-VMM if it is executed in the context of an OS. In the remainder

---

<sup>1</sup>Note, the term third generation computer may be extended to commodity platforms.

## 2.3 Taxonomy

---

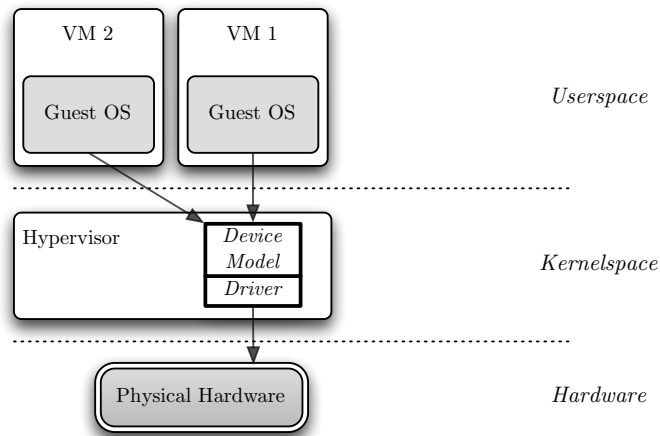


Figure 2.2: Type 1 hypervisor, based on [1].

of this thesis we will use a generic definition and employ the term hypervisor when referring to the control program. Nevertheless, machine virtualisation has evolved over time and hypervisors can be further classified into three types [216]:

**Bare-metal hypervisors (type 1)** - Bare-metal hypervisors are executed directly on the physical hardware platform. They are in sole charge of the hardware, and in this sense, are OSs themselves. The family of bare-metal hypervisors represent the classic form of a control program. Examples include, the previously discussed CP/CMS. More recent examples consist of VMware ESX(i) Server, XEN version 1.0 and L4 microkernels such as OKL4. See Figure 2.2 for a schematic description of a type 1 hypervisor.

**Hosted hypervisors (type 2)** - As outlined in Figure 2.3, hosted hypervisors rely on an OS for hardware abstraction and driver management. The hypervisor runs as a software application within the context of a conventional OS. Well known representatives of a type 2 hypervisor are VMware Workstation products, QEMU and VirtualBox.

**Hybrid hypervisors** - Hybrid hypervisors cannot clearly be classified as type 1 or type 2 hypervisors, and are consequently referred to as hybrid hypervisors. A hybrid hypervisor is split up into two parts. A first part consists of a small control structure which runs directly on hardware. The second is a management part which runs as a VM with special privileges. Much of the functionality that would be handled by either of the previous types is now outsourced into a management VM. For instance, device drivers are located in

## 2.4 Machine Virtualisation

---

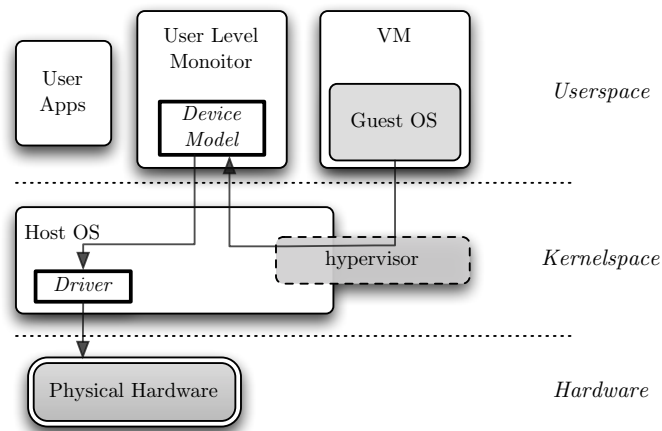


Figure 2.3: Type 2 hypervisor, based on [1].

the management VM and are passed through the hypervisor, directly to the hardware. The XEN hypervisor version 2.0 and above, is the most prominent example, and more recently Microsoft's Viridian. See Figure 2.4.

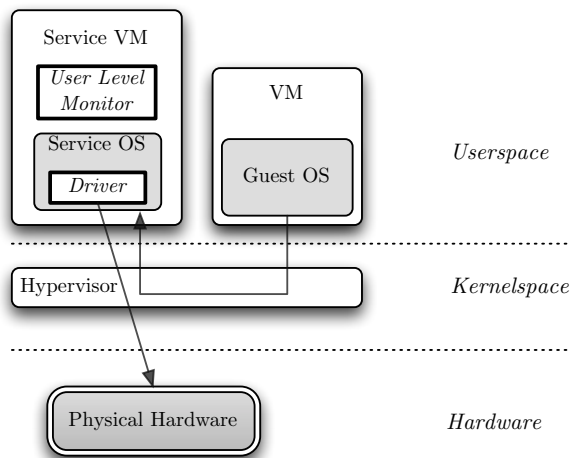


Figure 2.4: Hybrid hypervisor, based on [1].

## 2.4 Machine Virtualisation

Following the above definition of a machine, this section contains an outline of how a machine's basic building blocks can be virtualised.

## 2.4 Machine Virtualisation

---

The physical resource provider for machine virtualisation in the following is realised by the x86 architecture. On commodity systems, Intel's x86 architecture is exceptionally favoured and thus reality dictates this architecture. Even though alternatives such as the ARM architecture seem to be better suited for the low power requirements on mobile devices, when running legacy OSs, the x86 architecture remains predominant. We believe this predominance is mainly due to its backwards compatibility and the large amount of legacy applications still in use today. As a result we focus on the most dominant hardware architecture for commodity platforms available today.

Legacy OSs assume they are being executed in the supervisor mode (also known as ring 0 in ring-terminology), of the CPU. According to the foregoing definition, a hypervisor needs to be in control over the physical hardware; in other words, it is executed in supervisor mode. Virtualising consequently requires placing a hypervisor layer under the OS. Privileged instructions can be made to trap into the hypervisor, by executing them on a lower privilege level. This mechanism is known as ring deprivileging or ring compression. Unfortunately, to support ring compression on the x86 several technical and pragmatic challenges must be solved.

### 2.4.1 Intel x86 Architecture

OSs for the x86 generation hardware are designed to run on bare-metal hardware directly and do not fully understand the concept of resource sharing. Moreover, the x86 architecture lacks the ability to support virtualisation and is not virtualisable under the previously discussed definitions [216]:

- The x86 architecture was not designed to be virtualisable and contains instructions which do not generate an exception when executed in an unprivileged mode. For instance, OSs use *POPF* to pop CPU flags from the stack, but *POPF* will fail silently when executed in unprivileged mode.
- The x86 ISA contains unprivileged yet sensitive instructions, which result in the unprivileged usermode being able to access privileged states. For instance, a VM can read the segment selector, which stores the Current Privilege Level (CPL).

## 2.4 Machine Virtualisation

---

Hence those instructions could be executed by unprivileged code, touch resources, and never trap into the privileged mode. The x86 architecture contains 17 non-virtualisable instructions, which are generally referred to as sensitive instructions [216]. However, as will be discussed in Section 2.4.2, by cleverly adapting the hypervisor design, the x86 hardware is indeed virtualisable. As a result, Popek & Goldberg’s definition is sometimes referred to as *classically virtualisable* because they define virtualisation as a simple trap and emulate operation [4].

### 2.4.1.1 Ring Protection

For the understanding of this thesis we want to focus on the ring protection scheme of both the 32-bit and 64-bit x86 architecture. The distribution of extended and special purpose architectures, such as Intel’s Itanium or Transmeta’s Crusoe chip, is marginal and they are therefore beyond the scope of this thesis. Furthermore, we ignore the ring compression issues surrounding 64-bit platforms because this is not directly relevant for the concepts discussed below.

An x86 CPU implements four different modes of operation: System Management Mode, Real-address mode, Protected mode and Virtual 8086 mode [123, 124]. The CPU is initiated by the BIOS in 16-bit real-mode, which is a legacy mode mainly used during start-up. After start-up the CPU makes a transition into 32-bit or 64-bit protected mode, which provides different privilege levels called rings. The ring protection scheme is based on a 2-bit privilege level enabling the CPU to determine four different separate levels of privilege from ring 0 – with the most privileges to ring 0 – with the least privileges. The Current Privilege Level (CPL) is stored in the lower two bits of the segment selector.

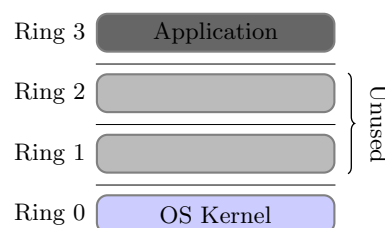


Figure 2.5: Normal ring protection scheme.

Depending on which ring level code is being executed, the program has access to different CPU functionality and features. Traditionally, an OS kernel, including

## 2.4 Machine Virtualisation

---

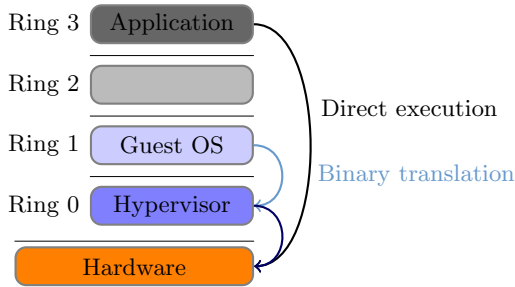


Figure 2.6: Binary translation.

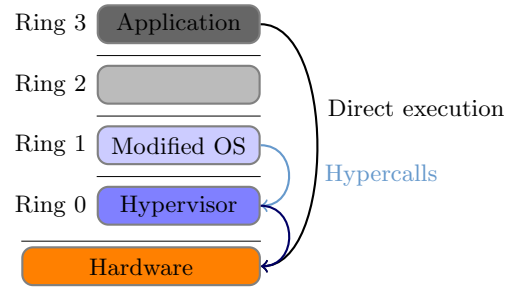


Figure 2.7: Para-virtualisation.

device drivers, runs with the highest privileges and no restrictions in ring 0. Applications are placed inside ring 3, the least privileged level. Rings 1 and 2 are generally unused. Figure 2.5 illustrates the privileges in a non-virtualised context.

### 2.4.2 Hypervisor Implementations

As discussed previously, the x86 architecture is not naturally virtualisable, and thus ring compression in itself is not enough to support virtualisation. The following three solutions have emerged to help implement an efficient hypervisor.

#### 2.4.2.1 Dynamic Binary Translation

A naive approach would be to emulate all machine code instructions on a VM and never execute directly in the underlying hardware. This allows an isolated duplicate of the real machine, but violates the efficiency property. Dynamic binary translation only traps and emulates those instructions that are sensitive and allows the VM direct execution of non-sensitive instructions on the hardware directly. The guest OS running inside the VM is not informed of being virtualised, because sensitive instructions are transparently converted during execution. The caching of already translated instructions evens out the initially high translation overhead. Efficient hypervisors can be built by dynamically analysing the instruction stream and emulating non-virtualisable instructions [4]. Figure 2.6 illustrates a binary execution flow.

## 2.4 Machine Virtualisation

---

### 2.4.2.2 Para-virtualisation

Para-virtualisation modifies the VM OS kernel to replace non-virtualisable instructions with *hypercalls*, which trap into the hypervisor. Hypercalls are similar to system calls in traditional OSs. Further changes to the hypervisor's interfaces are made to provide virtualisable interfaces for interrupt handling and memory management. In addition, para-virtualisation also provides a simplified interface for virtual I/O devices. Guest OSs that run on a different hardware interface have to be adopted to the new interface [25]. This adoption eliminates the overhead of traps and emulation, resulting in increased performance. A para-virtualised design is depicted in Figure 2.7.

### 2.4.2.3 Hardware Assisted

Hardware vendors are rapidly adapting to the demand for virtualisation. As a result, AMD (AMD-V) [7], and Intel (VT-x) [120], have developed concepts to retrofit virtualisation to the x86 platform. The hardware assisted virtualisation introduces a host and a guest mode into hardware. The host mode is intended to run a hypervisor, while the guest mode is designed to run VMs at their intended privilege level. Transitions to the host mode are called *VMExits* and transitions to the guest are referred to as *VMEntries*. All guest instructions can now be intercepted by the hypervisor, but privileged and sensitive instructions are set to automatically trap to the host mode. A new data structure – the Virtual Machine Control Block (VMCB) in AMD terminology and Virtual Machine Control Structure (VMCS) in Intel terminology – manages control state as well as the state of the virtual CPU. Note, for simplicity we will use the Intel terminology in the remainder of this thesis and use VMCS when referring to VMCB or VMCS. The VMCS holds the state of a VM including segment registers, CR3<sup>2</sup> and interrupt descriptors. The VMCS is restored when a VM is active and stored when the VM is not active. This is very similar to traditional OS context switches and is often referred to as a world switch. Furthermore, the VMCS can be programmed to exit on guest page faults, on access to privileged page tables and also on memory-mapped devices. Trap conditions can now be programmed into the hardware. Instruction emulation and state changes are still realised in software. While both Intel's and AMD's concepts are similar, they are not compatible at an instruction level.

---

<sup>2</sup>Control Register 3 when virtual memory is used.

## 2.4 Machine Virtualisation

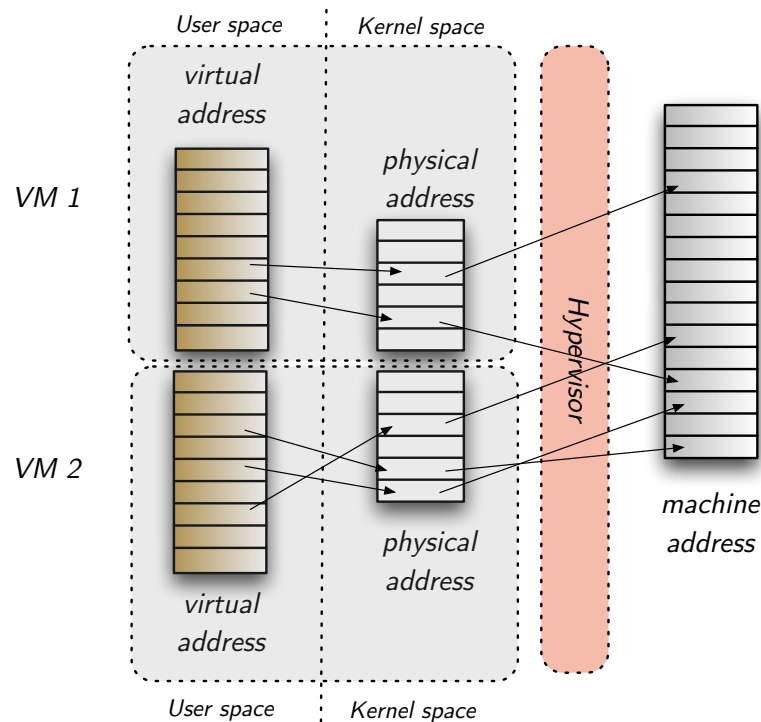


Figure 2.8: Three levels of memory virtualisation.

### 2.4.3 Memory Virtualisation

Together with the CPU, a platform's memory has to be shared and dynamically allocated to co-resident VMs. In a traditional OS applications use virtual memory addresses which are then mapped by the OS to the physical memory addresses. Applications use a contiguous address space which does not represent the physical layout of the system's memory. The OS therefore manages the address translation in Page Tables (PTs) and is assisted by a hardware Memory Management Unit (MMU). The MMU translates virtual addresses to physical addresses, as well as handling the Translation Lookaside Buffer (TLB) cache. As outlined in Figure 2.8, to support multiple OSs simultaneously, additional abstraction from the physical address is necessary.

In the following section we use and extend the definition provided by Bugnion et. al [40] and Waldspurger [278]:

## 2.4 Machine Virtualisation

---

- A *virtual address* is an arbitrary address managed by the VM.
- A *physical address* is a software abstraction used to provide the illusion of hardware memory to the virtualised OS.
- A *machine address* refers to actual hardware memory.

Para-virtualised guests may use modified interfaces to reflect the changes in memory management. However, to support unmodified guests the hypervisor has to fully virtualise the address translation. As highlighted in Figure 2.8, the virtual guest continues to maintain virtual-to-physical mappings, while the hypervisor is mapping the guest *physical addresses* to physical *machine addresses*. As outlined in Figure 2.9, in software-based address virtualisation, the hypervisor uses shadow Page Tables (sPTs), derived from the guest Page Tables (gPTs). The hypervisor marks the gPT as write protected, which results in a page fault if a guest tries to modify an entry. Alternatively, the hypervisor may virtualise the TLB, allowing the guest to directly modify the gPT, but a page fault will occur on reading the destination address. As page faults transfer control to the hypervisor, addresses can be translated accordingly [10]. Either way, memory virtualisation imposes a significant performance overhead, due to necessary page table maintenance performed by the hypervisor.

To reduce the amount of costly world switches, hardware vendors have introduced Extended Page Tables (EPTs)<sup>3</sup>, and Nested Page Tables (NPTs)<sup>4</sup>. Hardware-assisted Page Tables (hPTs), implement an additional set of page tables so virtual guests can modify their own machine addresses in a separate set of PTs. On one hand, this results in increased performance by avoiding the overhead of page table maintenance. On the other hand, it reduces the hypervisor's ability to monitor or interpose on guest memory.

### 2.4.4 Device and I/O Virtualisation

Initially, device virtualisation on mainframe computers was straightforward. The CP-67/CMS used a channel-based design founded on dedicated control processors [179]. The standardised channel I/O access allowed a secure and efficient device sharing model.

---

<sup>3</sup>Intel

<sup>4</sup>AMD

## 2.4 Machine Virtualisation

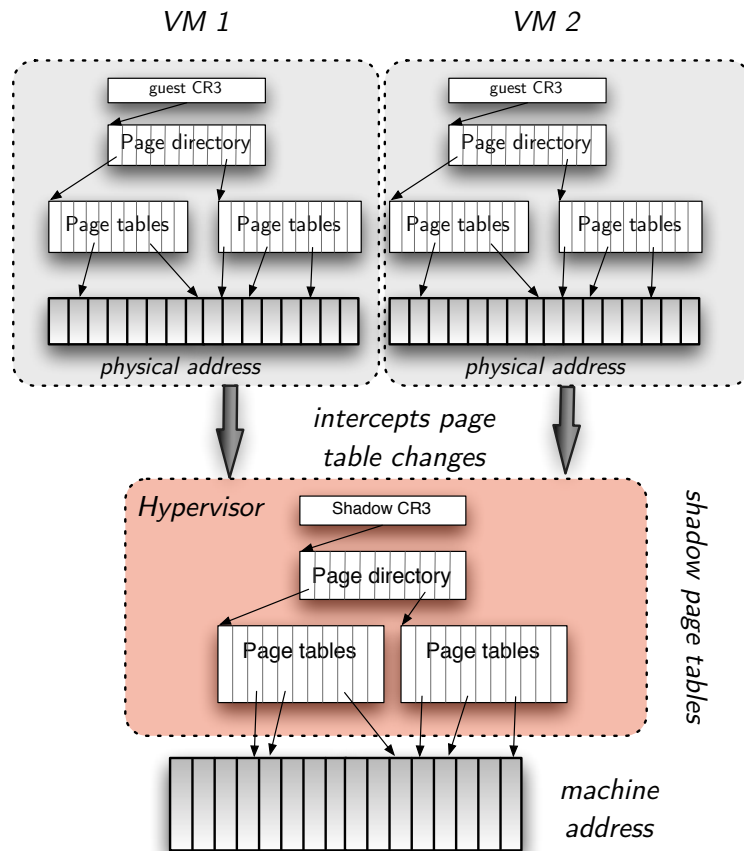


Figure 2.9: Shadow page table management.

The rich and diverse I/O devices for the x86 architecture render I/O virtualisation the most challenging part of commodity platform virtualisation. Modern platforms support a variety of I/O devices from different vendors with diverse programming models. As discussed next, three device models have emerged to address I/O virtualisation. While these device models enable I/O virtualisation, they are lacking in terms of either security or performance. Efficient and secure I/O sharing is currently an open research area [135].

### 2.4.4.1 Device Emulation Model

Every VM is populated with a set of virtual standard hardware devices, regardless of the physical devices present on the platform. The virtual guest runs the standard device driver for the emulated device, and if trying to read or write to the device,

## 2.4 Machine Virtualisation

---

traps into the hypervisor. The hypervisor handles the request by either performing the necessary emulation, or by passing the request on to the device emulation entity. On an x86 architecture, hardware is usually accessed using ‘IN’ and ‘OUT’ instructions. The emulation of the particular hardware is subsequently performed outside the VM and interpreted by the software which understands the semantics of those instructions. Using this model allows standard guest drivers to be used, but introduces two difficulties. Firstly, it inflicts an overhead due to world switching between the VMs, the hypervisor and, possibly, the device emulator. Secondly, the device model has to be complete – including software bugs – to allow device drivers to operate accordingly. The emulated device model is used by all VMware products [253], KVM [146] and XEN with hardware assisted VM guests [210].

### 2.4.4.2 Para-virtual Device Model

The para-virtualised I/O model offers an alternative to device emulation, by implementing a special device driver in the virtual guest. This model is based on a split design: a front-end driver in the guest VM connects via shared memory to a back-end driver in the driver domain. The guest VM does not have direct hardware access, but is informed of being virtualised and co-operates with the hypervisor. The driver domain has direct access to hardware and multiplexes guest requests. Expensive world switches are reduced and complex emulation is omitted, therefore the performance of a para-virtual model is better than in the device emulation model. A prerequisite of this reduction however, is the requirement of guest OS compatible front-end drivers being built. Depending on the virtualisation solution used, various classes of drivers are available, for example network and disk device drivers. This model is used by XEN for para-virtualised guests [210], KVM with virtio [221] and in VMware (if VMware tools are installed [253]).

### 2.4.4.3 Direct Assignment Model

Under some instances it might be desirable to directly map a device to a particular VM and grant it exclusive access. This is, for example, the case for the driver domain in XEN [83]. Moreover, if only one VM requires access to a display device, while the remaining VMs are service providers without the need for a display device, a direct assignment model might be chosen. However, as most modern device drivers

## 2.4 Machine Virtualisation

---

use Direct Memory Access (DMA) operations with physical addresses, this poses a security issue which has to be safeguarded against. Solutions – such as Intel Directed I/O [1] and AMD IOMMU [8] – address this security issue in hardware by mapping device DMA addresses into the guest’s actual memory. An Input Output Memory Management Unit (IOMMU), much like an MMU, translates device virtual addresses to machine addresses. IOMMUs were first proposed in 1975 for the Multics Secure Front-End Processor (SFEP) [33]. Whilst an IOMMU allows the safe, direct assignment of a device to a VM, it does not solve the issue of device sharing itself. On single-user platforms, such as commodity mobile devices, display devices must inherently be shared. Displaying potentially sensitive data from multiple VMs on different trust levels is a notoriously hard problem to solve.

### 2.4.4.4 Future Outlook

Despite the recent adoption of IOMMUs on commodity hardware, the hurdle of efficient device sharing is yet to be overcome. The Peripheral Component Interconnect Special Interest Group (PCI-SIG), published specifications to define extensions to the PCI Express standard in order to provide safe device sharing [201]. For example, the Single Root I/O Virtualisation (SR-IOV), allows partitioning of a PCI device function into many virtual functions, which are then directly mapped onto the VM’s address space via an IOMMU. This technique outsources the multiplexing and safeguarding into hardware, whilst removing the need for device emulation, offering a near native performance.

### 2.4.5 Summary

In this section we have given an overview of, and identified, the key challenges in machine virtualisation. We started by defining the terminology and outlining the obstacles that are associated with virtualising the x86 architecture. We defined and confined the areas of interest that are relevant for the remainder of this thesis. Extended literature on machine virtualisation exists and is detailed in the bibliography [4, 135, 218, 246, 253].

## 2.5 Security Discussion

Virtualisation has the potential to fundamentally change the way computing resources are consumed. However, new products and technologies are emerging quickly, and are being deployed with little consideration for security concerns. With the increasing popularity of modern virtualisation techniques, there is a corresponding increase in the threat level so it is vital to understand that virtualisation does not improve security inherently.

Security aspects of virtualisation have been an ongoing field of research since the technology emerged in the 1960s. One of the first analyses was carried out by Madnick and Donovan in the early 1970s [163]. They argued that security failures were similar to reliability failures. Moreover they discuss the probability of a program violating the security of another, concurrent program. Most importantly, they identified one of virtualisation's key security attributes: its isolation properties.

### Historical Discussion

In their 1974 definition, Popek and Goldberg pointed out that a VM is a duplicate of the original machine, barring timing effects. As discussed the year before by Lampson [147], the difference in time on a multi-program environment could leak information in the form of a covert channel. Lipner [158] proposed to address this issue by creating a virtual time, separated from the physical host time. However, keeping time itself [275] and managing the risks of covert channels in virtual environments remain difficult problems today [130]. A more recent security audit conducted by Ormandy demonstrated the presence of exploitable security flaws in most VMs [199]. Furthermore, as pointed out by Garfinkel et al. [88], the management of a sophisticated virtual infrastructure holds even more security risks.

### Virtualisation Dectection

Many papers have documented the use of VMs for malware analysis [54, 79]. Such papers have generated new thinking amongst malware designers, who are now trying to hide their malicious behaviour while they suspect they are inside a VM. Consequently, the ability to detect the presence of a hypervisor is of great interest, for

## 2.5 Security Discussion

---

both malware authors and malware hunters alike [45, 159, 195]. Some approaches to hypervisor detection, such as localisation of the Interrupt Descriptor Table (IDT) and Global Descriptor Table (GDT), may not work with future virtualisation hardware support of the x86 architecture. However, identifying a VM by registry string search, vendor specific MAC addresses or looking for a virtualisation support application (for example, an application responsible for clipboard management), are methods likely to continue to be reliable in the future. In particular, a VMware specific implementation of a communication channel allows VM detection with high probability. This communication channel may be triggered by the guest OS by filling the processor register with ‘VMXh’ – a specific value indicating a VM action. Modifying the guest OS to use rootkit-like techniques to prevent VM detection might be useful in a hostile environment, such as honeypots or malicious code analysis, but is certainly not applicable on a wider scale. For example, a patch provided by Korchinsky [145] allows honeypot specific VMware modifications, whereas Kirch [142] provides configuration advice to defer detection.

### Virtualisation Rootkits

Abusing virtualisation to mask malware as a hypervisor-based rootkit has been independently presented by both Rutkowska [222, 286] and Zovi [297] in 2006. We can classify virtualisation rootkits into two subcategories:

1. Software virtualisation-based rootkits.
2. Hardware virtualisation-based rootkits.

King et al. [141] presented a proof-of-concept software rootkit for hosted hypervisors such as Microsoft’s Virtual PC and VMware’s Workstation in 2006. These rootkits had to initially be installed with sufficient privileges that they could subsequently modify the boot process to launch their own code before control is passed to a hypervisor. The detection of such malware is fairly easy because they have to be installed onto persistent storage. Additionally, the x86 non-virtualisable instructions give a good indication of the presence of a hypervisor. Software-based rootkits also have a major disadvantage in that they have to implement virtualisation functionality and hardware support completely and precisely. This renders an universally applicable rootkit rather difficult to implement, but still poses a threat for bare-metal hypervisors which are tightly tailored to a specific set of hardware.

## 2.5 Security Discussion

---

Unfortunately, a more serious threat emerged with the hardware support for virtualisation introduced by AMD and Intel. The Blue Pill by Rutkowska [222] claimed to be an undetectable rootkit, hence creating a lot of controversy. The stealth property was based on the fact that the malware could be executed as a hypervisor and thus run underneath the actual OS. This design certainly makes detection, even for experienced users, more difficult, but not impossible [80]. Firstly, there will be a mapping between physical hardware and the virtual interfaces, which an OS can detect. Secondly, the consumption of resources – such as CPU time and memory – will create a detectable anomaly on the system. Those anomalies will also be reflected in latency and timing characteristics. More importantly, these techniques allow the detection of all hypervisors, and not only hardware virtualisation-based rootkits. With forthcoming advanced virtualisation support for the x86 architecture, even more sophisticated methods to hide malware are likely to emerge. These will be discussed in Section 4.2.4, and include exploiting either the System Management Mode (SMM) [73], or the Advanced Configuration and Power Interface (ACPI) [68].

### Multi-Level Security

A small number of research projects are trying to exploit virtualisation’s isolation property to provide Multi-Level Security (MLS), such as sHype [228] and Net-Top [178]. Virtualisation is used to separate multiple OS instances on different security levels. According to the MLS requirements for hypervisors as described by Karger [132], hypervisors can be classified into pure isolation hypervisors and sharing hypervisors:

**Isolation hypervisors** split a machine into separated partitions with no resource sharing. The segregated partitions behave like separate physical machines. Isolation hypervisors are mainly used in expensive mainframes, such as the PR/SM system for IBM’s z/Series [42]. On commodity desktop systems, however, isolation hypervisors may be used to isolate VMs of different security levels that do not have the requirement of sharing information. This is particularly useful, for instance, to aggregate multiple client machines that a military analyst must operate [133]. The National Security Agency’s (NSA) NetTop [178] project is such an example.

**Sharing hypervisors** allow most of the physical resources to be shared between VMs by permitting multiplexed access to fixed resources. Depending on the

## 2.6 Summary

---

MLS, protecting information in a sharing hypervisor can be as basic as file access control [133]. Examples of sharing hypervisors with security kernels include VAX VMM [137], KVM/370 [98] and Xenon [171]. These hypervisors were designed with MLS and high assurance in mind, and require significant changes to existing hypervisors or even a complete redesign.

Isolation hypervisors can be very costly and also can be found lacking in terms of performance and usability. Hence they are not very common on commodity systems. Efficient sharing hypervisors are difficult to implement on the x86 hardware if high assurance is paramount. The common x86 hardware imposes restrictions on the security those platforms are capable of providing. As will be discussed in the next chapter, providing confidentiality and integrity protection to a single commodity OS is a difficult challenge in itself. However, compared to the large footprint of a full-featured OS, hypervisors are usually relatively small – millions of Source Lines Of Code (SLOC) in a typical OS as opposed to only a few thousand in a hypervisor. Moreover, the interfaces of a hypervisor are much more narrow, well defined and less susceptible to alteration. Consequently, hypervisors have become the system of choice for retrofitting security on commodity platforms [50, 86, 238, 242].

## 2.6 Summary

In this chapter we have looked at the historical evolution of virtualisation, beginning with the early mainframes through to recent developments on the x86 architecture. In addition, the various types of virtualisation have been classified depending on their computer architecture interface. Further, we outlined the different challenges of virtualising the x86 hardware as well as alluding to possible solutions. We finished this chapter with a security discussion and outlined past and current hypervisor security research.

Machine virtualisation is, however, a very powerful addition to a wide range of applications, including software development, deployment and workload consolidation. The motivations and applications for virtualisation are manifold and range from desktop consolidation [184], green IT [277] and cost reduction [276] to containment [88]. Diverse reasons are discussed by various authors [181, 257] and are beyond the scope of this thesis.

# Chapter 3

## Trusted Platforms

### Contents

---

<b>3.1</b>	<b>Introduction</b>	<b>50</b>
<b>3.2</b>	<b>Trusted Computing</b>	<b>50</b>
3.2.1	Historical Overview	51
3.2.2	Trusted Platform	52
3.2.3	Trusted Computing Base	54
3.2.4	Trusted Platform Module	54
3.2.5	TPM Functional Overview	61
3.2.6	Static Root of Trust for Measurement	62
3.2.7	Dynamic Root of Trust for Measurement	64
<b>3.3</b>	<b>Discussion</b>	<b>67</b>
<b>3.4</b>	<b>Summary</b>	<b>71</b>

---

*Each user of computers must decide what security means to him. [...] Since there are many different sets of needs, there can't be any absolute notion of a secure system.*

– Butler W. Lampson

### 3.1 Introduction

---

## 3.1 Introduction

Unfortunately, the word trust is one of the most commonly used and least understood terms in computer security research. Gollman [102] points out that the many possible interpretations and misperceptions of trust are bad for security. Consequently, for the remainder of this thesis we will follow the definition of trust as shaped by the Trusted Computing Group (TCG) [260]:

“Trust is the expectation that a device will behave in a particular manner for a specific purpose.”

The Trusted Computing Group is a non-profit organisation whose intention is to develop and define industry standards for Trusted Computing (TC), and security technologies. According to the TCG, its specifications promise to enhance security in computing environments, as well as strengthen individual rights and privacy. The goals of TC, as well as its definitions have created controversy and sparked criticism [236, 294], to the extent that the TCG has been accused of censorship and monopolisation [13]. The TCG embraced much of the criticism and subsequently addressed the privacy concerns which had been raised.

## 3.2 Trusted Computing

The term “Trusted Computing” derives from *trusted system* which, in terms of security engineering, means a system that must be trusted [60]. A user of a trusted platform could therefore infer that a computer system is completely trustworthy but, as discussed previously, it means the system can only be trusted to behave in a particular manner according to its intended purposes. As noted by Martin [166], trust alone does not imply security, it merely requires predictable behaviour. As further outlined by Martin [166] the careful speaker might also distinguish between a trusted, trustworthy and trustable system. From a technology point of view however, the platform can only be trusted not to execute unauthorised code.

“**Trust**; a firm belief in the reliability or truth or strength etc. of a person or thing” [202].

## 3.2 Trusted Computing

---

This definition according to the Concise Oxford Dictionary raises the question: what kind of trust is needed for Trusted Computing?

Following the notion of social and behavioural trust, one can differentiate between whether a platform should be trusted or can be trusted [203]. Thus, TC can mean many different things to different people in different contexts. Under the terms of this thesis, Trusted Computing is understood as a security engineering concept.

### 3.2.1 Historical Overview

The concept of securing or trusting a computing system is itself not a novel idea. Like virtualisation it dates back to the early days of the mainframe era. It is not surprising then, that the history of virtualisation and TC are interlinked. During the mid 1970s there was a strong movement to retrofit security to mainframes, for instance the IBM VM/370 [19, 99] as well as the VAX security kernel [137, 138]. In 1973, Bell and LaPadula first presented a mathematical model for a secure system [27]. The paper defined the actors, objects and interactions necessary to build a secure system. In the early 1980s, the US government funded a research project on computer security. The outcome was a series of standards, known as the rainbow series, to evaluate trusted systems [190]. Perhaps the most well known – the orange book [60] – describes and defines a trusted computer system in great detail.

Those models were written in the context of mainframe computers and unfortunately do not match the requirements of modern computing platforms. The assumption that a computer is isolated in a secure room with limited user access has completely transformed into the opposite. Users today have physical access to a computer and often operate with a multitude of applications and even multiple OSs at the same time. In early 2000, when the trend in the computer industry had already turned to the PC, Balacheff et al. [21] began to ask how a computer platform could be trusted.

In 1999, the Trusted Computing Platform Alliance (TCPA) was founded to develop industry standards for TC. The TCPA released their first specifications and guidelines in 2001, defining and describing TC. At the ideological heart of the specifications are ideas for more secure environments without compromising functionality, privacy and individual rights. At the heart of the technical specification, the TCPA placed a trust anchor – the Trusted Platform Module (TPM). As described in Sec-

## 3.2 Trusted Computing

---

tion 3.2.4, the TPM is the root for most trust operations on a platform [261, 262, 263]. The TCPA was later disbanded and replaced by the TCG. Nevertheless, the TCG has adopted the key deliverables of the TCPA and provides backward compatible hardware and software interface specifications as well as marketing.

Microsoft first launched its Trusted Computing initiative in 2002 under the code-name Palladium, and subsequently renamed it to Next Generation Secure Computing Base (NGSCB). NGSCB is designed as a secure OS which relies on the concepts designed by the TCG [187, 204]. No products have yet been released by Microsoft based on NGSCB and the future roadmap remains unclear.

The Perseus [251] project's aim is the development of a trustworthy computing framework. The Perseus security architecture foresees a minimal security kernel for commodity systems with optional trusted computing hardware. The Kernel resides above the hardware layer and contains a resource management layer as well as a trusted software layer. On top of that layer are legacy OSs as well as new security-critical applications. The concept's main advantage rests on its reduced complexity and auditable code blocks. The European Multilaterally Secure Computing Base (EMSCB) [77], is developing a trustworthy platform with open standards to solve security issues of conventional platforms. Under multilateral security the EMSCB understands the enforcement of security policies of different parties, including end-users and industry. The outcome of the EMSCB's work so far – known as Turaya [78] – is based on the Perseus security framework and the L4-Microkernel. Turaya is a proof-of-concept design, relying on the TPM to provide integrity measurements of the microkernel.

In 2005 the Open Trusted Computing initiative (OpenTC) [197], was founded with the goal of providing an open Trusted Computing framework built with open-source software. The framework relies on the TPM with the aim of increasing the security of the core OS [197].

### 3.2.2 Trusted Platform

A Trusted Platform (TP) is understood as a system which can be trusted by a local or remote entity. The platform can be considered trustworthy if it can reasonably prove to the challenging entity when challenged that it can be trusted – hence the platform and challenger establish a link of trust or trust relationship. As defined

## 3.2 Trusted Computing

---

by Pearson [203], a trusted platform is a computing platform that has a trusted component, probably in the form of built-in hardware, which it uses to create a foundation of trust for software processes. Consequently, the difference between a Trusted Platform and a non-Trusted Platform is derived from a trusted component – a Trusted Platform Module (TPM), for example. Note, the TCG does not mandate a hardware implementation of the trusted component. However, as hardware generally allows better tamper resistance and key protection, in reality the choice of implementation will almost always be a hardware-based solution. Nevertheless, a hardware component is designed to increase the security of a computing platform by offering the user a greater assurance of the platforms' expected behaviour [186]. Following the TCG definition, the decision of whether a system is trustworthy or not lies with the user alone.

Trust is notoriously difficult, if not impossible, to measure [172]. However, in order to get any meaningful metrics from a trusted platform, the TCG has anchored their foundation of trust in three so-called roots of trust, which provide protected capabilities, integrity measurement and integrity reporting [260]:

**Root of Trust for Measurement** - The Root of Trust for Measuring (RTM), could be any instance capable of making reliable integrity measurements. As a consequence, the RTM is implicitly trusted. On PCs the main CPU will carry out this task while control is handled by the Core Root of Trust for Measurement (CRTM) – typically a piece of code embedded into the platform's BIOS. As will be further outlined in Section 3.2.6 and 3.2.7, an RTM can be initiated dynamically or statically.

**Root of Trust for Storage** - The Root of Trust for Storage (RTS), is responsible for storing the integrity measurements as created by the RTM. It does so by using special purpose registers as will be outlined in Section 3.2.4.7. Furthermore, it must safeguard the various keys as described in Section 3.2.4.5, as well as the storage referred to in Section 3.2.4.5. The RTS is provided by the Trusted Platform Module.

**Root of Trust for Reporting** - The Root of Trust for Reporting (RTR), is accountable for reporting the information held by the RTS to a third party. This mechanism is usually referred to as attestation, binary attestation or remote attestation [22]. Attestation is the method by which a TP reports its characteristics with regard to its trustworthiness. The Trusted Platform Module provides the RTR.

## 3.2 Trusted Computing

---

### 3.2.3 Trusted Computing Base

The many platform components, technologies and concepts which are responsible for the correct functionality and security of a computing platform constitute the Trusted Computing Base (TCB). In other words, a platform solely relies on the TCB components for functionality and security enforcement and therefore, to evaluate a platform's trustworthiness, it is necessary to assess its TCB.

The Trusted Computer System Evaluation Criteria [60], defines a TCB as follows: "The heart of a trusted computer system is the Trusted Computing Base (TCB) which contains all of the elements of the system responsible for supporting the security policy and supporting the isolation of objects (code and data) on which the protection is based." The criteria further demands the TCB to be as small as possible: "[. . .] a TCB should be as simple as possible consistent with the functions it has to perform."

Modern OSs however, support a feature-rich set of functionality and an increasingly complex security policy model, which unfortunately increases the complexity of its TCB. Moreover, the perception of the TCB may also differ from a user's, software vendor's and platform supplier's point of view. We will further discuss the issue of TCB complexity in Section 4.2 and investigate its improvement in Chapter 5.

### 3.2.4 Trusted Platform Module

The centre piece of the Trusted Computing Group's work is the Trusted Platform Module (TPM). The TPM is generally realised as a hardwired chip which builds the trust anchor for the overlying software stack. As the chip contains integrated input, output and memory, it may be classified as a microcontroller. Realised as tamper-resistant hardware, it incorporates cryptographic functionality to provide asymmetric key generation, encryption, digital signatures and a hash engine. Figure 3.1 depicts the TPM's functional overview described in the following sections. Moreover, it contains a Random Number Generator (RNG), able to produce random numbers of high quality. The default delivery state is usually disabled (see Section 3.2.4.11), and it only becomes fully functional after it has been enabled and been taken ownership of. This procedure is also known as opt-in and it will be further described in Section 3.2.4.11.

## 3.2 Trusted Computing

---

The TPM specifications have evolved over time, initiating with version 1.1b and the most recent being version 1.2 in revision 103 [261, 262, 263]. The next generation TPM.Next [265] specification is currently under development. The TPM specifications only describe functionality and do not dictate internal TPM design. It should be noted that a TPM might be realised in software but on commodity platforms it is commonly implemented as a hardware component. Vendors are free to implement chips through a different methodology; for instance Sadeghi et al. [225] criticise the fact that the TCG does not specify a maximum time limit for key creation. This can result in significantly different software behaviour, depending on the TPM's implementation. As reported by McCune et al. [169] different TPMs produce a significantly different response time to TPM commands, such as *Seal*, *Unseal* and *Quote*. Additionally, the RNG's output ranges from 136 Bytes to 1257 Bytes per single request, which can inflict delays if multiple requests are necessary [225].

Nevertheless, as we will demonstrate in Section 4.1, the TPM is a key component in a trusted virtualisation infrastructure as it contributes the RTR and RTS. Together with the RTM the TPM is the key building block for trusted initialisation and faithful reporting of the current platform state, both of which are essential requirements for building a trusted virtualisation platform. The information provided in the following sections is based on the existing TPM specification [261, 262, 263]; however we want provide only the background information necessary for the remainder of this thesis. Extensive literature on trusted computing and on the TPM in particular already exists. For further reading consult [22, 47, 84, 186, 203].

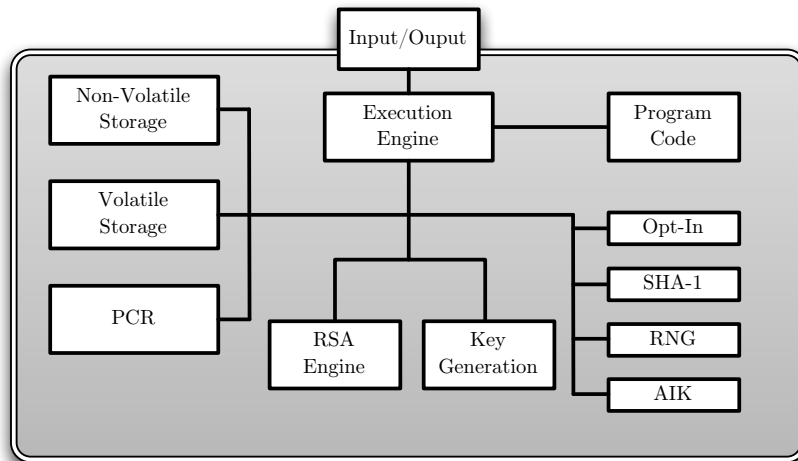


Figure 3.1: Trusted Platform Module design, based on [103].

## 3.2 Trusted Computing

---

### 3.2.4.1 Owner, User, Operator

As discussed by Grawrock [103], a trusted platform differentiates between three roles:

**An Owner** controls the platform, for instance by deciding on policies and delegating capabilities. It is important to note that an owner can be located remotely and may not be the person operating the platform. This would be the case if the owner is an IT-department for example.

**A User** exploits the platform resources granted by the owner. A user can thereby be a human or a process.

**An Operator** physically manipulates the platform. An operator may be a user or an owner.

Thus in a corporate environment the roles would be separated. The owner would be the IT-department while the role of user and operator is held by an employee. In a private environment all roles might be held by the same person.

### 3.2.4.2 Input and Output

The Input/Output (I/O) port provides the interface between the TPM and other platform components – all TPM commands must pass via this interface (see Figure 3.1). A TPM command is represented by a unique ordinal number as defined in [263]. The TPM specifications themselves do not define a distinct connection interface. The TCG PC Client Specific TPM Interface Specification or (TIS) [259], however, assumes the TPM is connected to the chipset via the Low Pin Count (LPC) bus. The LPC bus is designed for low-bandwidth (4-bit wide, at 33.3Mhz), legacy I/O devices [119]. As pointed out by Grawrock [103], accessing the TPM via the LPC bus from a very restricted environment is an important design decision. Neither a BIOS nor any security sensitive code can trust a complex software stack to address the TPM on its behalf.

## 3.2 Trusted Computing

---

### 3.2.4.3 Execution Engine

The execution engine is the command parser of the TPM. It first ensures that the input format is correct and then executes the internal program code associated with a command [186]. Conceptually, the program code is the CRTM but on commodity systems the CRTM is realised as part of the BIOS firmware.

### 3.2.4.4 Program Code

The program code is responsible for the validation of the entire command bit stream, the parameters and, if required, the command authorisation [103].

### 3.2.4.5 Non-Volatile Memory

Non-Volatile (NV) memory is typically implemented as an Electronically Erasable Programmable Read-Only Memory (EEPROM), and is used to preserve keys and other persistent state information while the TPM is not supplied with power.

**Endorsement Key** is a 2048-bit key pair generated by the manufacturer in order to protect information and, together with a certificate for the corresponding public key, verify its authenticity. The Endorsement Key (EK) is never directly used for encryption or signing. The EK is a non-migratable key and the private part always remains inside the TPM. If a chip fails there are no means to access the EK. Under version 1.2 of the TPM specification, it is theoretically possible for an owner to create their own EK and revoke the one supplied with the TPM.

**Storage Root Key** is a 2048-bit key pair, which is first generated when a user takes ownership of the TPM. The Storage Root Key (SRK) creates a tree-like sub-key hierarchy by encrypting each key beneath. The private part of the SRK never leaves the TPM but it enables the platform to store sub-keys externally. However, to subsequently use a protected key it has to be transferred to the TPM and be decrypted internally.

## 3.2 Trusted Computing

---

**Owner Authorisation Secret** is a 160-bit shared secret which is created together with an SRK during the adoption of the TPM's ownership. The secret must be presented if executing TPM commands from the category, "TPM-owner authorised" commands.

### 3.2.4.6 Volatile Memory

The volatile memory – usually realised as Static Random Access Memory (SRAM) – is used internally to perform data manipulation. TPM manufacturers can decide how much volatile memory to implement and how it is used. This may include storing authentication and transport sessions as well as storage for keys. Volatile memory has the advantage of a much longer lifespan, as NV memory offers only a limited amount of write cycles. As some platforms may enter a sleep state – for example laptops – current information stored in volatile memory must be secured, either via storing it in NV parts of memory or by providing an extra battery backup.

### 3.2.4.7 Platform Configuration Registers

A Platform Configuration Register (PCR) is a 160-bit wide storage location for integrity measurements – the size of a SHA-1 hash value. While the general TCG specifications define a minimum of 16 PCRs [260], the TIS dictate that a compliant TPM must implement at least 24 PCRs [259]. PCRs are not directly writable – they are only “extendable” using the TPM command interface. Extending occurs by concatenating the current and new values and then calculating the cryptographic digest as follows:

$$\text{PCR}_{number} = \text{HASH} ( \text{PCR}_{old} || \text{New value to add} )$$

This approach has two interesting properties. Firstly, it creates ordered records of previous digests and secondly, updates are not commutative.

PCRs can either be static or dynamic. On a PC,  $\text{PCR}_{[0-15]}$  are defined as static, whereas  $\text{PCR}_{[16-23]}$  are allocated for dynamic usage [259]. Static PCRs are not resettable during operation and are only reset when the TPM initialises during a platform reset. Dynamic PCRs however, can be reset independently by software

## 3.2 Trusted Computing

---

and hardware running under sufficient privileges [103]. Section 3.2.7.1 has a detailed description of the privilege levels.

### 3.2.4.8 RSA Engine

The RSA [214] engine is utilised during digital signing and key wrapping operations. Implementing the RSA algorithm requires the TPM to comply with the PKCS #1 v2.1 standard [220]. The default key size is 2048 bit and, when used internally in the TPM, the minimum size.

### 3.2.4.9 Key Generation

In order to generate post-production RSA keys internally, the TPM is supplied with an asymmetric key generation engine. Keys can either be used for data encryption or for digital signatures. The TCG specification requires the EK and SRK to be stored in a shielded location and also requires the RSA implementation to be IEEE P1363 [118] compliant.

The key's length, attribute, purpose and authorisation data has to be specified upon creation. Types could, for example, be signing, storage, identity, authchange, bind, legacy and migrate [262]. Length can be 512, 768, 1024 and 2048 bits – where 2048 is the minimum recommended key size. A key attribute is migratable or non-migratable. The authorisation data can be a key password or migration password.

### 3.2.4.10 Key Migration

A certificate for a non-migratable key and its security properties may be created by the TPM on which it was generated. A certifiable migratable key (CMK), can be migrated but also retains properties which the TPM, on which the CMK was generated, can certify. When a CMK is created, control of its migration is delegated to a migration (selection) authority. In this way, controlled migration of the key is made possible whereby an entity other than the TPM owner makes some contribution to the decision as to where the CMK can be migrated to. This ensures that the certified security properties of the key are retained.

## 3.2 Trusted Computing

---

### 3.2.4.11 Opt-in

The TPM specification requires the TPM to be an opt-in device and to be shipped as the customer requires – usually disabled. Opt-in requires the platform owner to undergo a specific procedure to willingly enable the TPM. This must include the verification that a human is operating the platform by proof of physical presence, for example by pressing a button.

### 3.2.4.12 SHA-1 Engine

The TPM must implement an SHA-1 engine which complies with the FIPS 180-1 standard [191]. SHA-1 engine application includes authorisation, message integrity protection and authentication checks. Due to known vulnerabilities in SHA-1, future TPM implementations are likely to implement different or additional hash algorithms [103].

### 3.2.4.13 RNG

The TPM specification foresees that a TPM incorporates a true RNG, but again implementation is manufacturer specific. The RNG is used for symmetric and asymmetric key generation, nonce creation as well as for seeding faster pseudorandom number generators. A more detailed discussion about the TPM's RNG can be found at [186].

### 3.2.4.14 AIK

The Attestation Identity Key (AIK), provides anonymous proof that an entity is communicating with a real TPM implementation. It allows the remote entity to gain information about the internal status of the TPM without revealing a specific TPM. Therefore, it is an un-linkable alias for the EK. A TPM may generate an arbitrary number of AIKs. The creation of an AIK requires the owner to transmit the Endorsement, Platform and Conformance certificate along with a new PubKey ID to a Trusted Third Party (TTP), or Privacy CA. The TTP verifies the certificates and replies to the ID request by sending a signed ID back to the TPM.

## 3.2 Trusted Computing

---

### 3.2.4.15 Platform credentials

Multiple credentials, including a platform credential, an EK credential and conformance credentials should be provided by every trusted platform. These credentials describe the properties of the trusted platform.

### 3.2.5 TPM Functional Overview

In this section we describe the key features which arise from the basic building blocks discussed previously.

#### 3.2.5.1 Integrity Measurements and Reporting

Integrity measurements carry information about the platform's state. Measurements are usually hashes over a program binary and configuration files. An integrity metric is the extended chain of measurements stored inside the TPM's PCR [203]. Each measurement of the chain of events is kept outside the TPM in the Stored Measurement Log (SML). This provides evidence to attest to the current platform state. Together with the integrity measurements stored in the PCR and the SML, the RTR can provide a challenger with faithful information about the current platform state and the past chain of events. The challenger therefore, specifies a set of PCRs, provides a nonce and the host responds to the attestation request with the *TPM\_Quote* command [263]. The TPM then signs the requested information using one of the TPM's AIK private keys and returns the information to the challenger. This process is referred to as Platform Attestation. To address privacy concerns, the TCG specification 1.2 describes Direct Anonymous Attestation (DAA), to protect the platform's identity [37]. Privacy concerns on TC is not an essential part of this thesis, but the interested reader might consult [24, 151, 152] for further discussion on this topic.

#### 3.2.5.2 Binding and Sealing

Binding is the process by which information is cryptographically tied to a specific TPM. When the TPM encrypts external data – using a public key corresponding to a non-migratable private key – that information is considered “bound” to a specific

## 3.2 Trusted Computing

---

TPM. As outlined in Section 3.2.4, the TPM encrypts and decrypts this data internally and releases it for use on its platform; thus the encrypted information never leaves the TP unprotected.

Sealing is an extended form of binding and one of the fundamental features of the TC concept. Sealed data is not only uniquely bound to a specific TPM, but also associated with integrity metrics representing a distinct platform configuration. The sealed data can only be decrypted by the TPM when the platform is in the requested state. For instance, symmetric keys can be unsealed and data deciphered only if the platform is in a pre-defined state.

During the sealing process a caller can specify an arbitrary PCR value – the platform may or may not be in this specified state. The TPM only enforces the correct PCR during an unseal operation [103].

### 3.2.6 Static Root of Trust for Measurement

The Static Root of Trust for Measurement (SRTM), provides a means of extending the CRTM by building a chain of trust based on integrity measurements. To bootstrap a TP the SRTM offers two distinct modes of operation: a reported/authenticated boot and an enforced/secure boot. The SRTM relies on the TPM's PCRs to store integrity metrics. As discussed in Section 3.2.4.7, PCRs can only be extended and not arbitrarily set, even though physical TPM reset attacks have been demonstrated [139, 247]. Those PCRs reflect a platform configuration since the last platform reset and consequently the RTM is labeled as a static.

#### 3.2.6.1 Authenticated Boot

An authenticated boot process measures and records the platform's boot procedure. The main objective of an authenticated boot is to extend the trust boundaries of the initial root of trust by iteratively measuring and reporting objects which are not included within the trust boundary. As outlined in Section 3.2.2, on a PC platform the RTM is generally implemented as part of the BIOS, which must be implicitly trusted and therefore acts as the core RTM (CRTM). As the BIOS is simply firmware being executed on the platform's main CPU, the CPU effectively operates as the

## 3.2 Trusted Computing

---

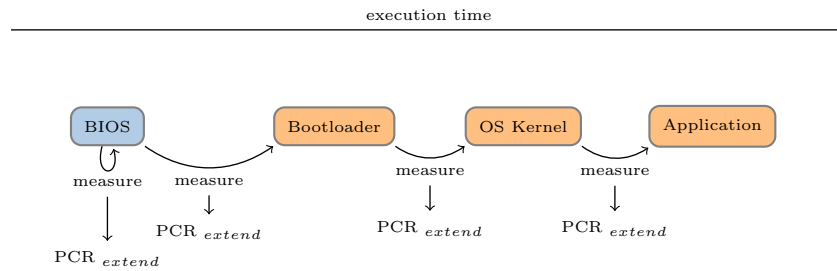


Figure 3.2: Static Root of Trust for Measurement.

RTM. The RTS is provided by the TPM. We briefly describe the execution flow of an authenticated boot in the following (refer to Figure 3.2):

- The CRTM measures itself, as well as the remaining BIOS, and reports its integrity to the TPM. The TPM stores this first measurement into a PCR – on a PC platform typically  $\text{PCR}_{[0]}$  – afterwards, control is passed on to the BIOS.
- The BIOS then continues to measure any present option ROM<sup>1</sup> as well as the OS loader. Once these measurements are reported to the TPM the control is handed over to the OS loader.
- This process is continued – as shown in Figure 3.2 – until all required objects are measured and reported.

Because of its transitive nature, this process is also referred to as establishing transitive trust [103].

### 3.2.6.2 Secure Boot

Whilst an authenticated boot reports a platform state, a secure boot enforces a specific platform configuration. A secure boot will prevent a platform from booting if a reported integrity measurement does not match the corresponding pre-defined Reference Integrity Metric (RIM). For a more detailed discussion on secure booting refer to [129] and [84].

---

<sup>1</sup>option ROMs essentially are BIOS extensions.

## 3.2 Trusted Computing

---

### 3.2.7 Dynamic Root of Trust for Measurement

The main difference between a static and a Dynamic Root of Trust for Measurement (DRTM) is that the latter can be initiated at any time without requiring a platform reset. Because it is executed after the platform has already been initialised, this process is also referred to as *late launch*. As described in Section 3.2.4.7, a PCR can either be static or dynamic. The TIS [259] defines  $\text{PCR}_{[0-15]}$  as static while  $\text{PCR}_{[16-23]}$  are dynamic and thus resettable. In order to distinguish between an SRTM and a DRTM, all bits of  $\text{PCR}_{[17-23]}$  are set to “1” during the TPM initialisation phase. A dynamic reset causes a resettable PCR to be set to “0”. Access to dynamic PCRs varies according to the access privileges – represented by localities as highlighted in Section 3.2.7.1. For instance,  $\text{PCR}_{[17]}$  can only be reset by a CPU instruction from locality 4, only issued after the platform has been instructed to perform a DRTM. In addition to resetting the PCR, the platform is set into a known, trusted state, without performing a complete platform reset. Figure 3.3 illustrates a DRTM which should be contrasted with the SRTM shown in Figure 3.2.

Setting the platform into a trusted state includes a reconfiguration of the MMU and IOMMU to prevent unauthorised access to protected memory regions. Moreover, the CPU is reset and, in a multicore environment, all but one of the cores are shut down and interrupts are disabled during the initialisation phase.  $\text{PCR}_{[17]}$  is then filled with the hash of the software which will gain control after the DRTM has finished. Instead of creating a consecutive chain of measurements – starting with the BIOS – late launch enables the platform to start a measured environment at any time after the platform’s initial boot. In theory, this excludes the BIOS from the chain of trust, however, as we will discuss in Section 4.2.4, this is not yet fully applicable. The code launched after a DRTM could be a kernel or arbitrary code, but throughout the course of this thesis we assume the code launched during a DRTM incorporates a hypervisor.

In the following section, we will briefly discuss the DRTM implementations realised by AMD and Intel as well as highlighting their differences. The DRTM, specifically on the Intel platform, will be discussed in more detail in Section 7.3.1.

---

<sup>2</sup> $\text{PCR}_{[16]}$  is only used for debugging purposes.

## 3.2 Trusted Computing

---

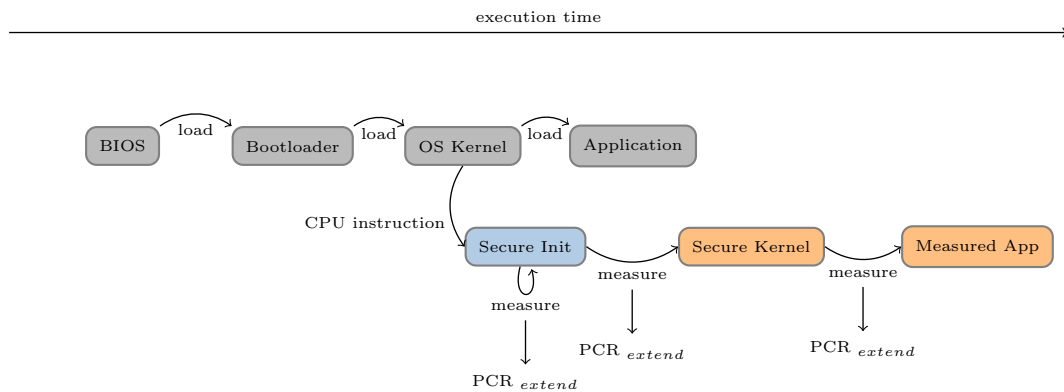


Figure 3.3: Dynamic Root of Trust for Measurement.

### 3.2.7.1 Locality

A DRTM requires an unforgeable communication mechanism between the TPM and the DRTM. For this purpose, the TPM uses localities – a hardware addressing scheme to allow the TPM to distinguish different privilege levels, similar to the x86 protection rings described in Section 2.4.1.1. The TIS [259] defines six localities and Grawrock [103] outlines their usage as follows:

**Locality 4 : Trusted hardware** - Locality 4 is the highest level defined and should exclusively be accessed by the trusted hardware of the DRTM. Only locality 4 has the capabilities to send data to the TPM to extend the Locality 4 PCR, as well as resetting PCRs. Locality 4 PCR ( $\text{PCR}_{[17]}$ ), also holds the first measurement of the DRTM.

**Locality 3: Auxiliary components** - The use of this locality is optional and is designated for processes executed directly after the trusted hardware is initialised. If locality 3 is used it is implementation dependent.

**Locality 2: Runtime environment** - This represents the default locality for the trusted OS after locality 3 and 4 have been initialised.

**Locality 1: Trusted OS environment** - An optional environment under the control of the trusted OS.

**Locality 0: The legacy environment** - The locality for the Static RTM and its chain of trust.

## 3.2 Trusted Computing

---

**Locality Legacy** - This is Locality 0 as used to provide backwards capability for TPM version 1.1, which did not implement localities.

### 3.2.7.2 AMD Secure Virtual Machine

On AMD's Secure Virtual Machine (SVM) platform the DRTM is initiated via the *SKINIT* CPU command [9]. *SKINIT*, which only takes a physical base address as argument, must be issued with kernel-level privileges (ring 0). The memory address describes the length and location of the Secure Loader Block (SLB). The SLB is a code block – limited to 64 Kbyte – that contains the Secure Loader (SL), which initialises the SVM hardware and includes the trusted software to perform the DRTM. Upon calling *SKINIT*, the CPU enters flat 32-bit protected mode with paging disabled, while also setting up a Device Exclusion Vector (DEV), to protect the SLB's memory from DMA. In a multiprocessor environment, an interprocessor handshake protocol causes all but a single Bootstrap Processor (BSP) to enter a halted state. Interrupts and debugging capabilities are disabled during this phase to prevent software from regaining control over the CPU. Afterwards, the CPU causes the dynamic PCRs to be reset and transmits the SLB to the TPM in a manner that cannot be forged by software. The TPM extends the SLB's hash into PCR<sub>[17]</sub> and the CPU jumps unconditionally to the SL entry point. Once the SL gains control over the platform it can act as an RTM. The SL could however, include arbitrary code and the validation of the SL code is not enforced by hardware. It lies with other entities to evaluate the trustworthiness of the SL code, based on the measurements gained during *SKINIT*. The code could later be evaluated via remote attestation, but initially is free to execute.

### 3.2.7.3 Intel Trusted Execution Technology

The technological goal in Intel's Trusted Execution Technology (TXT) [121], of achieving a DRTM is very similar to AMD's SVM, but the implementation method varies and the two approaches are not compatible at a binary level. Intel TXT is demonstrated in more detail in Section 7.3.1, and hence we restrict ourselves here to emphasising the differences to AMD's SVM. Intel's late launch occurs in two phases: firstly, the initial CPU instruction GETSEC[SENTER] triggers the verification of a platform-specific Authentication Code Module (ACM). The ACM is a cryptograph-

### 3.3 Discussion

---

ically signed binary module provided by Intel. Verification is achieved by injecting a public key into the platform's chipset during the manufacturing process. However, the integrity metric of the ACM is extended to  $\text{PCR}_{[17]}$  while omitting to send the whole code block to the TPM for hashing. In the second phase, once authenticated, the ACM initialises the platform and then measures the pending code block which is set to be executed after the initialisation. The code executed after the ACM is usually referred to as Measured Launch Environment (MLE), and comparable to AMD's SL. However, the MLE is subject to policy enforcement by the ACM as well as being extended into  $\text{PCR}_{[18]}$ . The policy enforcement mechanism is also discussed in detail in Section 7.3.2.

Like AMD's SVM, Intel protects against DMA, interrupts and debugging attempts. In conclusion, the main difference is the vendor supplied ACM module which, like a BIOS, handles the platform initialisation. The ACM is potentially a break-once-run-everywhere approach, where security of the AMD approach varies with SL implementation. While on Intel platforms the ACM, respectively the main CPU, acts as a CRTM, on AMD platforms the TPM acts as the CRTM. Additionally, it is arguable whether a user or vendor supplied initialisation module offers better security properties. TXT has the advantage of being able to enforce a platform owner policy as well as a platform user policy. Section 7.3.2 provides a detailed discussion of Intel's policy enforcement mechanisms.

### 3.3 Discussion

In summary, to satisfy the TCG's definition of a trusted system, it seems sufficient for a platform to present the correct integrity metrics, where integrity metrics are measurements of semi-fixed properties over the lifetime of a platform, which do not change or, if changed are detectable. This model seems fine if looking at fixed hardware properties, and tolerable to a limited extent for low-level firmware such as the CRTM, but becomes increasingly difficult when applied to complex software. For instance, measuring an application usually requires the binary image to be measured prior to loading and executing the application in memory. Applications are however mostly dynamically linked and required functions are offloaded into Dynamic Link Libraries (DLLs), for reuse. As a result the corresponding DLL must be included in the metric identifying the application's binary. There are two apparent issues in this model. Firstly, a DLL contains a multitude of different functions and has to be

### 3.3 Discussion

---

included if the application only uses a single one, and hence the DLL's other functions become implicitly trusted. Additionally, as pointed out by Sailer et al. [230], this poses a challenge to the scalability of such an architecture. It also results in a changed integrity metric for the application itself (if the DLL has been updated or changed). Mostly this will not be under the influence of the application developer. Secondly, the execution of the binary might not be static and its behaviour might change dynamically, even though it is represented by the same static measurement.

Applications and OSs are often very changeable, with patching and upgrading taking place during runtime. Lyle et al. [162] conclude that attestation in itself is not sufficient for evaluating the trustworthiness of a platform. An application could change its behaviour deliberately or accidentally over time, while still being represented by a static measurement – a time of check to time of use (TOCTTOU) condition. This change in behaviour has a profound impact on the previous trust assumptions.

Traditional commodity OSs lack the ability to achieve strong process isolation [5]. In conventional OSs, isolation can be delivered by separating the address space between processes with an MMU and by preventing unprivileged access to the latter. As further discussed by Aiken et al. [5], these protection mechanisms can incur significant overheads and, as a result, most OSs map processes into the same address space without further protection. Separating processes and preventing unauthorised access to trusted applications is a general weakness of the previous TC model. We will further elaborate on the isolation weaknesses on commodity platforms in Section 4.2.

Trust in a platform generally requires a subset of components to be explicitly trusted. On a PC platform this includes the TPM implementation, the main CPU, the chipset and typically any component that might compromise the platform's security. In order to gain a certain amount of assurance, these components are generally required to function correctly and therefore compose the Trusted Computing Base (TCB). We will discuss the TCB in greater detail in Chapter 5. Unfortunately, the current model of trusted computing for the PC platform, as specified by the TCG, includes components in the TCB which cannot explicitly be trusted. The traditional PC architecture was designed as an open architecture to allow everyone to use and contribute to it. This enabled the PC to evolve and find use in a wide range of applications. It has however also led to a great deal of legacy applications and backward compatibility which poses a security threat today.

### 3.3 Discussion

---

The SMM (System Management Mode), is a prime example of such a legacy burden from the past. The SMM is one of four modes of operation that a CPU could be in – the others being real-mode, protected mode and virtual 8086 [123, 124]. Initially designed for debugging purposes for the Intel 386SL, it is now a fixed part of all x86 chips including compatible chips from different manufacturers such as AMD. The SMM is used by the platform manufacturer to handle various platform-specific events such as thermal or power management. Due to the importance of such events the SMM possesses high privileges and can access most platform resources directly without any restrictions [123]. Consequently, the SMM must be part of the TCB. However, as demonstrated by Dufлот et al. [65] and also discussed by Embleton et al. [73], if the SMM is not safeguarded correctly it can be compromised by rogue software. This poses a security threat to all applications running on the x86 architecture, but has a more profound impact on trusted platforms. We will discuss the use of SMM in detail in Section 4.2.4. As further discussed by Dufлот et al. [67], similar problems exist with the Advanced Configuration and Programming Interface (ACPI).

Those entry mechanisms into the TCB could be exploited to change the integrity metric or simply the behaviour of any trusted application, with serious impact on the previous trust model. As we will emphasise in the following paragraphs and further discuss in the remainder of this thesis, more protection mechanisms are required for a trusted virtualisation infrastructure.

Apart from the previously discussed issues with the SMM and ACPI, another possible change in the trust assumptions is currently emerging. Intel recently introduced a new generation of Active Management Technology (AMT) [122], which allows platform configuration and management in an Out-Of-Bound (OOB) fashion. OOB means access to low-level functionality without OS involvement – even if the platform is powered off or in a sleep state. AMT is implemented as a low-level application, which is executed on the chipset (Northbridge). The chipset usually refers to a combination of Northbridge and Southbridge which link together the various components on a computing platform. For simplicity, we group them as one logical entity and refer to them as the chipset. The chipset, an embedded microcontroller, is similar to a platform’s main CPU as it can be programmed to execute any code. Traditionally the code executed on the chipset is special purpose, high performance, low-level firmware comparable to a conventional BIOS. However, AMT implements a fully-fledged management webserver and is also able to redirect platform devices, such as IDE and floppy devices. In this sense the combination of

### 3.3 Discussion

---

chipset and firmware is in no way different to a CPU executing a traditional OS. Moreover, the AMT code is independent of the main OS as well as the main CPU. Additionally, it has dedicated access to the network interface to implement remote access as well as OS independent network filtering.

The introduction of AMT forces the management code to be included in the TCB of every platform. The AMT like any option ROM can be included in the SRTM but this has profound trust implications as AMT allows remote access to any aspects of a platform without the OS being aware of the access. Furthermore, vulnerabilities in web-based services account for a large number of remote, exploited vulnerabilities [44] and thus offer a potential entry point for malware. As pointed out by Tereshkin et al. [266], the AMT could be used to bypass the SRTM. However, the attack described by Tereshkin et al. is currently only applicable to a specific chipset (Intel Q35), and further hardware protection mechanisms – for instance, protecting memory regions with an IOMMU – can help to prevent such an attack.

As the execution of code on the chipset is independent from the main execution environment, Bulygin [41] exploited this to implement an OS independent integrity scanner and virtualisation-based rootkit detector. The so-called DeepWatch is embedded into the chipset’s firmware, hence it runs underneath every hypervisor and also every potential rootkit. DeepWatch is capable of scanning, detecting and possibly sanitising a virtualisation-based rootkit. Unfortunately, the signature-based approach has certain disadvantages. Firstly, the malware could adapt easily by changing its signature and secondly, the low-level position of the scanner renders maintenance difficult.

Future chipset revisions (Intel Q45 [127]), will use the chipset’s capabilities to implement more functionality – for instance an integrated TPM (iTPM). All this functionality has to be integrated into the TCB, eventually swelling the TCB to the point where trusting it becomes effectively meaningless. Additionally, a compromise of the AMT subsystem can potentially expose the TPM’s inner workings to an attacker. This can include arbitrary access to PCRs as well as keys. Moreover, the evaluation of such enhancements lies only with the manufacturer as this code is delivered as pre-installed binary firmware, which is often tightly bound to a platform by cryptographic protection. The RTM must gain control over the system at exactly the right time – measurements after the RTM has been compromised have no meaning.

### 3.4 Summary

---

Consequently, in order to create a meaningful chain of trust as well as potentially excluding the BIOS, the AMT code and the main CPU from the trust chain, the CRTM should be moved into or below the chipset itself.

On one hand, TC may provide unique measurements of software components as well as the platform state; however, the value of such a metric is arguable if the underlying platform fails to isolate trusted from untrusted components. On the other hand, one of the outstanding properties of virtualisation is the ability to provide strong isolation. The above discussion highlights the importance of further protection strategies for a TP. It is not surprising then that there is a strong convergence between the two technologies. Consequently, we will discuss in the following chapter how TC can be used to provide authentication for virtualisation and how virtualisation can deliver strong isolation for TC.

### 3.4 Summary

In this chapter we introduced the notions of Trusted Computing and Trusted Platforms as well as describing their technological implementations. We highlighted the hardware capabilities which a Trusted Platform must possess, and also those which it can optionally implement. Additionally, we described the recently introduced Dynamic Root of Trust for Measurement, which provides an important building block for the concepts we outline in the remainder of this thesis.

Moreover, we discussed some of the implicit hardware challenges that come along with the trust assumptions of the TC model. The TCG specifications are constantly evolving and numerous authors have discussed possible applications and challenges for trusted computing. For instance McCune discussed the problem of creating the first point of trust for user-based attestation [170], where Halder et al. [105] discuss fine-grained attestation using virtualisation. McCune further outlines how the TCB for applications on commodity systems could be reduced [167]. OSLO, the Open Secure Loader [139] and the tboot project [128], are real life exploitations of TC technology which seek to shorten the chain of trust by excluding the BIOS from the TCB. Further Balfe et al. [23] highlight high-level hurdles which impede the widespread use of TC. The interested reader can find additional information on a wider range of Trusted Computing applications in [22, 47, 84, 85, 165, 186, 203].

## Part II

# Problem Definition and Related Work

# Chapter 4

## Problem Definition

### Contents

---

<b>4.1</b>	<b>Trusted Virtualisation</b>	<b>74</b>
4.1.1	Motivation	74
4.1.2	Requirements	76
<b>4.2</b>	<b>Challenges</b>	<b>78</b>
4.2.1	Isolation Issues	78
4.2.2	Trusting the Hypervisor	80
4.2.3	Management Domain	81
4.2.4	Platform Limitations	83
4.2.5	I/O Device Sharing	87
4.2.6	Virtualising the TPM	88
<b>4.3</b>	<b>Discussion</b>	<b>91</b>
<b>4.4</b>	<b>Related Work</b>	<b>94</b>
4.4.1	Virtualisation and Isolation	94
4.4.2	Trusted Computing	97
<b>4.5</b>	<b>Summary</b>	<b>99</b>

---

*So in war, the way is to avoid what is strong and to strike at what is weak.*

– Sun Tzu

### 4.1 Trusted Virtualisation

As we will discuss in Section 4.2.1, commodity OSs typically fail to provide strong isolation and offer only poor levels of assurance. Thus the model of building trusted applications upon weak assurance systems seems flawed. This was very well researched over 30 years ago [11, 157] and continues to be acknowledged in academia [150, 160]. Nevertheless, much academic work has been published with the assumption of proper process isolation in commodity OSs.

Virtualisation can help by providing a much more profound degree of segregation, essentially by shifting the isolation boundaries from a process level to an OS level. This allows special purpose, secure OSs to concurrently coexist with general purpose and legacy OSs, allowing virtualisation to be applied to a wide range of application scenarios. Most importantly, the user experience is not changed by retrofitting isolation schemes to commodity OSs – which usually requires a compromise in terms of usability. Security solutions, such as SELinux [96] or grsecurity [104], impose a remarkable amount of configuration and maintenance work, while crucially restricting a user’s ability to perform their daily tasks [26]. Special purpose OSs however, can be significantly smaller in size and complexity than commodity OSs. Potentially, they are composed of only a boot loader and an application [169] – therefore removing the complexity previously required to retrofit security mechanisms.

By combining TC with virtualisation technology we can create multiple personae with different security requirements on a single platform. Figure 4.1 depicts a sample trusted virtualisation architecture, supporting an SRTM and a DRTM.

#### 4.1.1 Motivation

Traditionally, the TC concept only supports a single OS with, typically, one physical TPM on a platform. Even though Microsoft’s NGSCB offers the execution of multiple isolated software compartments, control and hardware access is still handled by a single isolation kernel [204]. The extension of the root of trust would consequently be performed in a pre-defined sequence. Machine virtualisation, however, allows a dynamic creation and destruction of concurrent OSs in the form of multiple VMs which result in a tree-like trust dependability between the hypervisor and its VMs [211]. Whereas applying the TC concept to the base virtualisation layer seems

## 4.1 Trusted Virtualisation

---

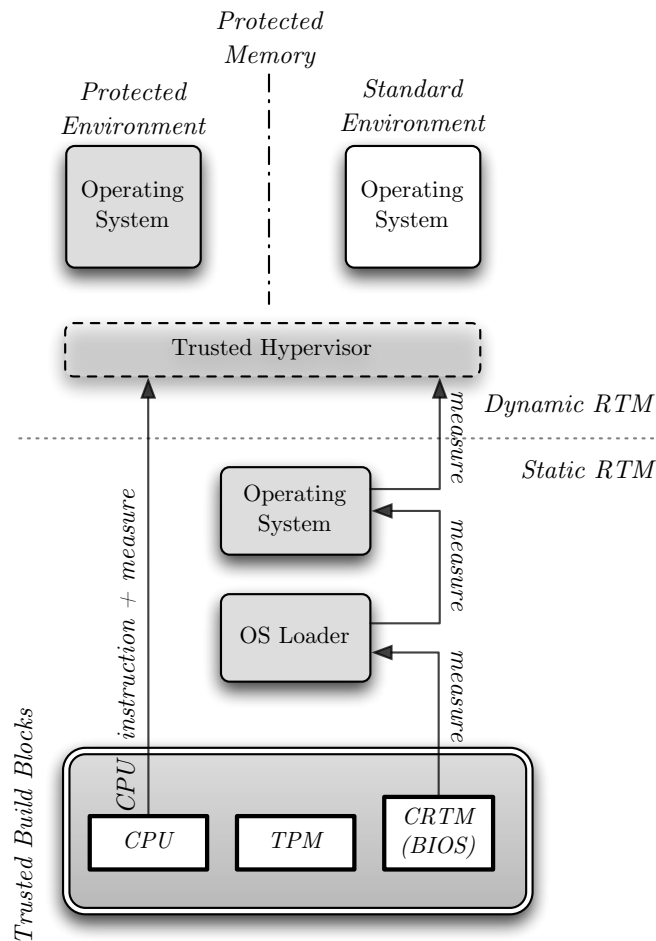


Figure 4.1: Simplified picture of a trusted virtualisation architecture.

feasible and has been already demonstrated by Garfinkel et al. [86], providing similar guarantees to VMs is a much more complex task. Firstly, the physical capacities of the TPM chip are limited and it only could accommodate a very limited amount of VMs. We will further explore the issue of TPM sharing in Section 4.2.6. Additionally, the loose binding between VMs and their host physical platform poses a variety of challenges for TC. For example, attesting the platform state to a remote entity requires a one-to-one binding between a single OS and the physical platform.

Virtualisation also undermines the assumptions that a machine has some sort of physical representation – for instance a fixed installation on a hard drive. This previously explicit binding is loosened and traditional hypervisors unfortunately offer only weak authentication mechanisms. Therefore, VMs or applications, which need to authenticate themselves to remote entities, fail to provide evidence of their authen-

## 4.1 Trusted Virtualisation

---

ticity. VMs might also migrate to different physical platforms, which is equivalent to swapping the hardware during operation. In the most simple case, the remote platform might not even have a TPM implementation at all. In a more complex scenario, applications might depend on previously existing encryption keys, which were bound to the previous host platform.

On one hand, virtualisation in itself poses challenges to the established view of a TP and, on the other, TC can address some of the security and assurance concerns which arise with virtualisation. For example, TC can determine the identity of a hypervisor, allowing a local or remote entity to gain certain information about the isolation layer. In turn the hypervisor can increase confidence in correctness of the hosted applications and provide assurance that they are operating free from any interference. Nevertheless, trusted virtualisation does not give evidence of the level of isolation or protection – it just reports on one of many possible configurations [103].

### 4.1.2 Requirements

A meaningful assurance statement for any application requires trust at all layers, including the management frameworks, protocols and hardware. Throughout the course of this thesis we will focus on the basic building blocks required to provide a more meaningful trustworthy virtualisation layer. For an entity to gain greater confidence in the correct operation of a virtual application, we require the trusted virtualisation layer to provide more properties than simply the sum of the TC and virtualisation building blocks. Consequently, we identify several basic trusted virtualisation requirements in the following sections. Further discussions of trusted virtualisation requirements are presented by Grawrock [103].

#### 4.1.2.1 Isolation

Secure isolation of execution environments at a machine level is essential to provide confidentiality and integrity assurance. Security sensitive code should be isolated from all other software, firmware and hardware. Ideally, the platform can prove to a remote party its isolation properties. A trusted virtualisation platform can inherit some of these properties from the strong isolation guarantees provided by a hypervisor.

## 4.1 Trusted Virtualisation

---

We will therefore discuss in Chapter 5, how the confidence in the isolation properties of a hypervisor can be strengthened, and how the value of attestations of these properties can be increased.

### 4.1.2.2 Attestation

VMs, as well as the hypervisor, must be able to cryptographically identify themselves to a remote entity. A remote party needs to gain assurance of the nature of the system and the status of individual execution environments, in order to be able to determine that the platform holds the desired properties. From a high level point of view we can achieve this today by applying existing TC technology and concepts – for instance, authenticated boot, property-based attestation – to the hypervisor. However, as will be outlined in Section 4.2.6, this holds certain challenges and restrictions.

### 4.1.2.3 Policy Control

Knowing what is executed and enforcing software behaviour are important properties for trustworthy virtualisation. Well defined, robust and enforceable policies for VMs and the hypervisor are required as a consequence. We will discuss access control mechanisms in Section 4.2.1 as well as Intel’s policy enforcement model in Section 7.3.2.

### 4.1.2.4 Complexity

Complexity is typically understood to be the single biggest issue in software security [47]. Hypervisors are still considered simple software components with thousands of lines of code as opposed to millions of lines of code in commodity OSs. Hypervisors only need to provide relatively simple, well defined hardware interfaces, which reduces internal complexity drastically. Special purpose VMs can greatly reduce software complexity while remaining compatible with existing hardware interfaces. This supports a wider range of application through portability, as well as applying a stricter set of access control policies without compromising user experiences. We will investigate the issue of complexity further in Section 4.2.1 and

## 4.2 Challenges

---

demonstrate how hypervisor security can be improved by dissecting software complexity in Chapter 5.

### 4.1.2.5 Compatibility

Most modern virtualisation technology can run unmodified legacy OSs, thus allowing a wide range of existing applications to be executed. The simple hardware interfaces a hypervisor presents to VMs offer a great amount of compatibility – an important property to satisfy any legacy concerns. At the same time the strong isolation guarantees allow trusted and untrusted VMs to coexist. We will further discuss the scenario of segregating private and corporate VMs in Chapter 7.

## 4.2 Challenges

The following section will emphasise some of the many technical and conceptual obstacles which need to be overcome to meet the above requirements and to construct a reliable, robust and predictable virtualisation layer. We will focus on low-level platform properties with regards to the previous requirements. A broader discussion on trusted virtualisation can be found in the related work pointed out in Section 4.4.

### 4.2.1 Isolation Issues

Notably, Multics was the first OS designed to be a secure system [134, 136]. The vulnerability analysis carried out by Karger et al. [134] and their iterative approach to security penetration, led the way for many security analytic approaches. As Karger et al. note, Multics was the direct predecessor of UNIX, and many of the vulnerabilities found in Multics are still present in today's commodity OSs [136]. For instance, buffer overflows, insufficient use of hardware features, such as NX flags and memory segmentation, can be traced back to vulnerabilities in Multics.

In the past, time-sharing systems showed that users have conflicting interests as well as highlighting the need for protection and isolation of concurrent users. If applied to today's platforms, there is no need for per-user isolation. It is necessary

## 4.2 Challenges

---

to isolate and protect the resources from the user, as is shown in the subsequent paragraphs.

Most modern commodity OSs implement a Discretionary Access Control (DAC) mechanism, empowering the user to exclusively control available resources. Under the DAC model, every object is assigned a controlling owner who manages and delegates the objects' permissions. This model is fine if the user is sensible and no malicious code is acting on behalf of the user. Unfortunately, userspace application vulnerabilities account for a large number of system compromises [29, 212], for example, email viruses and web-browser hijacking. Additionally, DAC also applies to administrative accounts, which are often used by default, thus rendering the delivery of security guarantees very difficult.

On the contrary, secure OSs often implement Mandatory Access Control (MAC) mechanisms, which places a control layer above all system resources. MAC policies usually define formal authorised subjects to allow access to certain labelled objects, based on pre-defined policies. Those policies then govern the interaction between subject and object, effectively containing users, as well as malicious code acting on their behalf. MAC policies cannot prevent application compromises, but they can contain potential damage in the event of a compromise, thus offering more precise security guarantees than DAC policies. MAC policies are historically associated with Multi-Level Secure (MLS) systems and especially with the Bell-LaPadula [27] information classification model. The most common application for MAC policies are military grade secure OSs, such as Honeywell's SCOMP [97] and HP CMW [296]. Unfortunately, MAC policies are difficult and costly to implement [26] and also notoriously hard to integrate into corporate management structures. For those reasons, they are not very common on commodity OSs.

Challener et al. [47], as well as Lampson et al. [148], conclude that developing secure software is extremely difficult and that pure software-based solutions are inadequate for security. They suggest three main reasons for this shortcoming:

- Software complexity – modern software systems consist of millions of lines of source code;
- Retrofitting security while maintaining backward compatibility;
- Existing security solutions are software-based themselves – thus security systems have their own bugs.

## 4.2 Challenges

---

The FLASK bug [223] is a prime example of a software security solution which itself presents the entry point for exploitation. It also underlines the difficulties of adding code and complexity to retrofit security. Lampson et al. [148] further argue that the lack of hardware-based security mechanisms is another obstacle for security applications.

Nevertheless, the main property which virtualisation and MAC policies share is containment: like MAC policies, virtualisation cannot prevent application compromises but can mitigate their impact on other OSs and applications. Also, virtualisation is much easier to apply to commodity systems than MAC policies while at the same time it ensures backward compatibility. Isolation (requirement 4.1.2.1) and compatibility (requirement 4.1.2.5) are critical prerequisites in a corporate environment and consequently virtualisation is becoming a key enabler for a more secure computer architecture. However, to address the tight security requirements in, for example, corporate environments, virtualisation must also be able to support enforceable policies (requirement 4.1.2.3).

A secure OS design has more properties than containment and MAC policies. The interested reader might consult [60, 101, 103, 258] for further discussion on secure OS design.

### 4.2.2 Trusting the Hypervisor

Hypervisors differ from past software sandboxing techniques as they address isolation at a much lower level. Rather than isolating single processes, a hypervisor isolates whole OSs. Each isolated OS is further responsible for its own integrated security protection mechanisms. The hypervisor only handles access to the hardware and provides each isolated guest with the illusion of running directly on dedicated hardware. If a malicious party manages to compromise one VM, it should not be possible to access resources used by other VMs or the host platform.

Given the fact that the hypervisor is responsible for isolation, less code will result in less bugs thus fewer potential security flaws but also less functionality. While the code base of hypervisors, such as XEN, is relatively small – XEN 3.3 contains approximately 150k SLOC [82] – it still forms a large code base which has to be included in the Trusted Computing Base. This figure excludes the millions of SLOC in the privileged management compartment, which currently must be included in the overall

## 4.2 Challenges

---

TCB. It seems that vulnerabilities in such a large TCB are inevitable; as a result, the security of every application which depends on a large TCB is at risk. Proving the correctness of such a large and complex TCB correct seems infeasible; hence there is a strong movement to disaggregate the management domain [189] and reduce the TCB of the XEN hypervisor [12, 91, 188]. Consequently, reducing the complexity (requirement 4.1.2.4) and maintaining compatibility (requirement 4.1.2.5) are essential prerequisites for a trusted virtual infrastructure. Nevertheless it is still possible under some restricted assumptions, for example, on non-x86 hardware, to prove that a hypervisor will behave as expected [144].

As identified in Section 4.1.2.2, additional guarantees are necessary to convince a third party that a particular hypervisor and a certain virtualised infrastructure are currently executing. Numerous research projects are exploiting the TC mechanisms to ensure and attest that only defined and therefore trusted hypervisors are loaded [86, 230]. The only security property they potentially attest to is a large TCB, which almost certainly contains exploitable vulnerabilities. A more meaningful, that is more fine-grained attestation (requirement 4.1.2.2), along with stronger isolation properties (requirement 4.1.2.1) than current hardware and software can provide, is necessary. Hence, in the following chapters, we will further investigate how the TCB can be strengthened and how segregation of co-resident VMs can be improved.

Rooting the trust and security properties in hardware with the assumption that hardware protection supersedes software protection is a commonly used security principle. Unfortunately, this contradicts the hardware abstraction goals of virtualisation, and more importantly, can be a dangerous assumption [69, 70, 288]. With the ongoing convergence of virtualisation and TC, it is also not surprising that chip manufacturers such as Intel and AMD combine virtualisation-enabled hardware with trusted extensions to help provide a virtual execution environment with security properties rooted in hardware.

### 4.2.3 Management Domain

Many virtualisation models implement a special privileged VM. This VM holds not only special privileges to instantiate new VMs, but also contains all device drivers by default. For a para-virtualised model this is usually referred to as management VM, or control VM, whereas for hosted hypervisors this is the host OS itself. In XEN

## 4.2 Challenges

---

terminology this is also labelled Domain 0 (Dom 0) as it is the first domain spawned with special privileges. Moreover, Dom 0 is in charge of all physical resources, including system memory and the graphics adapter, as well as acting as the back-end for the I/O driver model.

Dom 0 usually consists of a full-blown OS including numerous userspace tools for VM creation. This has a serious impact on the assurance of such a system. The special privileges for VM creation are exposed to all userspace applications, and hence any compromise of these privileges subsequently undermines the security of co-resident VMs. For example, in the XEN model the code to build VMs is dynamically linked into the Python interpreter, which in turn must be included in the TCB [189]. The whole of Dom 0 must be included in the TCB of the trusted virtualisation layer as a result. As highlighted in Figure 4.2, this results in an ‘L-shaped’ TCB: Horizontally the TCB includes the hypervisor, and vertically it includes the management domain. Not only does this produce a virtually unbound TCB, but it also undermines the isolation assurance of all applications running in co-hosted VMs. In turn this has a profound impact on the complexity (4.1.2.4) and isolation (4.1.2.1) requirements. Similar trust implications apply to the location of the device drivers as we will discuss in Section 4.2.5 and the TPM as outlined in Section 4.2.6.

Anderson et al. [12] demonstrate how the execution environment of a management domain can be drastically reduced by stripping and separating functionality into special VMs. A comparable proposal to outsource the VM building process into a special builder VM has been made by Murray et al. [189]. Both comprise a minimalistic kernel based on the XEN MiniOS [12, 189] and a set of Inter VM Communication mechanisms (IVMC). This excludes userspace applications and drastically reduces the amount of code included in the TCB.

To improve on the isolation (4.1.2.1) and complexity (4.1.2.4) requirements, we will further discuss how we can improve the TCB by exploiting new hardware features in Chapter 5. Figure 4.3 highlights an improved TCB design, which excludes most of the userspace applications as well as the standard Linux kernel.

## 4.2 Challenges

---

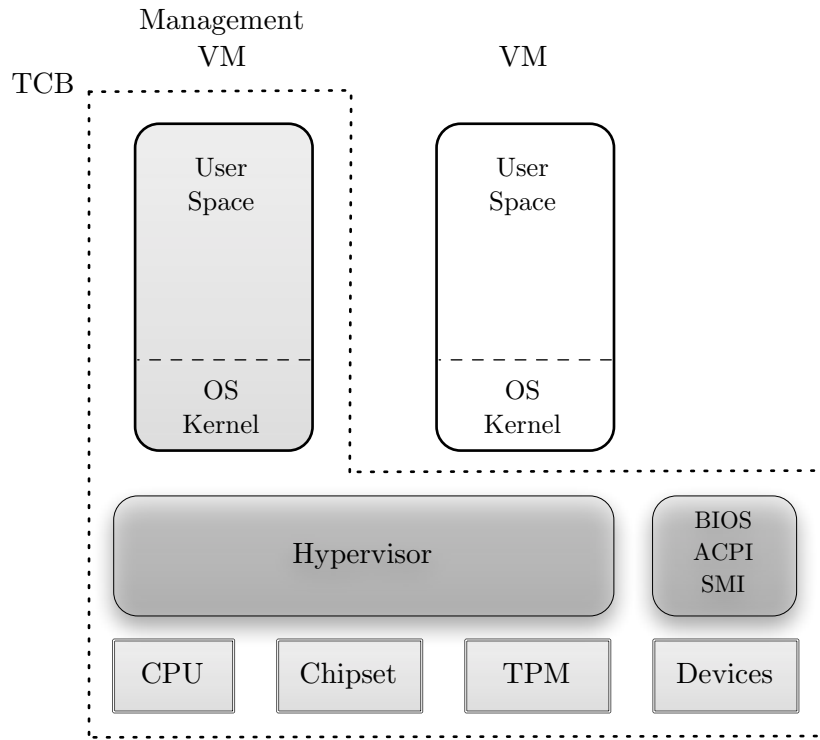


Figure 4.2: Current virtualisation trust boundaries.

### 4.2.4 Platform Limitations

The platform objects responsible for its correct operation are granted maximum privileges and generally consist of the TCB – for example the CPU, the chipset, the TPM and the hypervisor. Earlier we identified compatibility (4.1.2.4) as a key requirement for a trusted virtualisation infrastructure. Compatibility is one of the main benefits of the x86 architecture but, unfortunately, certain aspects of the CPU design as well as the platform’s configuration and management interfaces, dramatically limit the isolation guarantees (requirement 4.1.2.1) and thus trust assumptions. In Figure 4.2, the dotted line illustrates the current trust boundaries of a virtualised commodity platform while Figure 4.3 outlines an optimistic TCB design. Even though the previously discussed DRTM aims to exclude certain configuration and initialisation aspects from the TCB, some existing objects – for instance the System Management Interrupt (SMI) and Advanced Configuration and Power Interface (ACPI) – need to be implicitly trusted [67]. The problem arises from the fact that the BIOS installs a different set of handlers for both the ACPI and SMI, which can be called even after a DRTM has been initiated. We will further elaborate on ACPI

## 4.2 Challenges

---

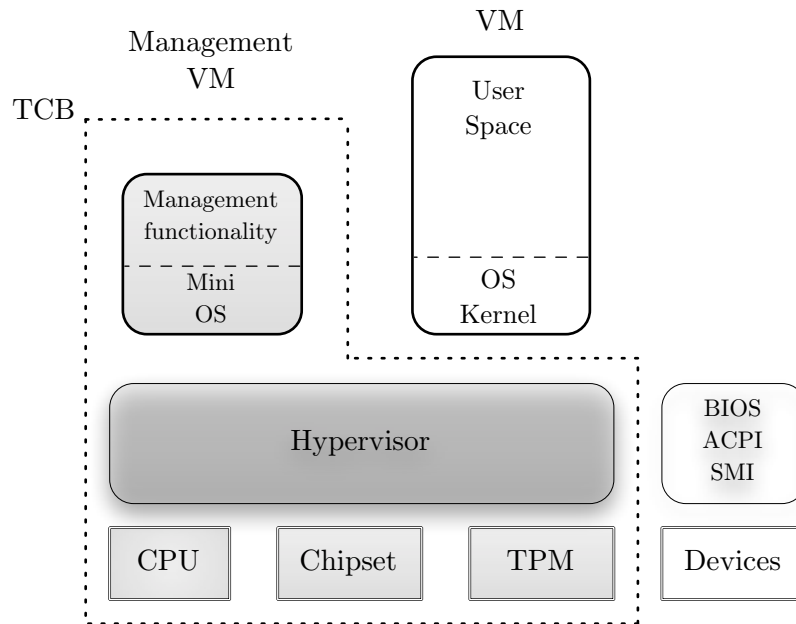


Figure 4.3: Optimistic view on virtualisation trust boundaries.

and SMI in the following paragraphs.

### 4.2.4.1 System Management Mode

As briefly mentioned in Section 3.3, the SMM is one of four modes of processor operation and was initially designed for debugging purposes. Today it is used to execute the SMI handler which is defined and created by the platform supplier. The system BIOS installs the SMI handler in system memory (SMRAM), which is then dispatched to handle numerous platform events, for example, power management or device emulation. Because of the importance of certain functions, such as thermal events, the SMM mode is not subject to any security mechanism and can access most platform resources directly without any restrictions [123] – this is very similar to ring 0 privileges. As the SMM is a concurrent mode of CPU context, which neither the OS in protected mode (ring 0), nor the hypervisor in HVM root mode (ring -1) are aware of, it is sometimes referred to as ring -2 [65, 69, 288]. An SMI event can either be triggered by a physical hardware interrupt or software which have ring 0 privileges.

## 4.2 Challenges

---

Gaining control over the SMI handler is a very powerful tool for any attacker and potentially undermines all previous trust assumptions. Consequently, most chipset BIOSs prevent direct access to the SMRAM region by software which executes outside the SMM by setting the *D\_LCK* bit in the SMRAM Control register (SMRAMC) [69]. The fact that some chipsets fail to protect the SMRAM has been exploited for privilege escalation [65, 66] as well as rootkit implementations [73].

Modern platforms take extra precautionary steps to protect the SMRAM from write and read access – even from system software. Consequently, to gain access to the SMRAM, an attacker needs to bypass system SMM memory protection. This has been successfully demonstrated by Wojtczuk et al. [287] with a memory remapping bug in the Intel Q35 chipset BIOS and a further undisclosed vulnerability. Moreover, Wojtczuk [288] and Duflot [69] both discuss the possibility of modifying SMRAM regions with cache poisoning. They both exploit a flaw in the SMM security model: the chipset protects the SMRAM address which it “assumes” to be within the correct address ranges, while the CPU might be using a different address. Duflot et al. [67] demonstrate that such an attack is feasible even on various vendor-specific platforms, by relocating the SMI handler via cache poisoning.

Attacks on the SMM have been proven possible and, according to Grawrock [103], removing the SMM from a platform is considered impossible. However, attacks on the SMI hold certain limitations: for instance modifications to the SMRAM are non-persistent, thus a system has to be compromised again after reboot. Alternatively an attacker may succeed in injecting a malicious BIOS to install a persistent rootkit. Unfortunately, unauthorised modification of a BIOS has also entered the realm of possibility as further demonstrated by Wojtczuk et al. [290]. The SMM attacks described by Wojtczuk and Duflot are very platform-specific and thus might not be applicable on a wider scale. Nevertheless they still pose a threat to platforms which are specifically targeted. For instance a victim might be a high profile government agent or, alternatively, a malicious party could use the platform-specific nature of this type of attack to threaten to release a platform-specific SMM bomb in order to blackmail a particular platform vendor.

In order to exclude the SMM from the TCB and safeguard access to the SMI, Grawrock [103] proposes a SMM Transfer Module (STM). The STM is envisioned as a hypervisor (or proxy) to the SMM and policy enforcement point, which accepts SMIs. It is not yet clear who is responsible for the creation of an STM, the CPU

## 4.2 Challenges

---

manufacturer, the platform manufacturer, or the OS designer. As a result, no STM exists today. However, the STM must be part of the TCB and included in the platform measurement chain. Additionally, to prevent a cache poisoning attack, Duflot et al. propose a CPU modification, although details have not yet been published [67].

### 4.2.4.2 Advanced Configuration and Power Interface

The Advanced Configuration and Power Interface (ACPI) is the successor to the Advanced Power Management (APM) interface – specified by an industry consortium [2]. The goal of ACPI is to provide generic interfaces to an OS for platform-specific events, such as device configuration and power management. Device configuration typically takes place during initialisation through the BIOS, while power management is handled by system software. The ACPI design consists of an ACPI BIOS, ACPI registers and ACPI tables. For simplicity, we will focus in the subsequent paragraph on ACPI tables and their interpretation (a detailed implementation description of all aspects of ACPI can be found at [2, 68, 110]). ACPI tables are written in a higher level language – the ACPI Specification Language (ASL) – and compiled into machine interpreter language – the ACPI Machine Language (AML).

System software would usually incorporate an OS Power Management (OSPM) module which, together with an AML interpreter and an ACPI device driver, would execute the functions specified in the ACPI tables. The tables themselves are organised in a tree-like structure which branches over all devices on a platform while the power management events are the leaf nodes. On a specific power event the OSPM triggers the AML which is contained in the corresponding leaf node of a device tree [68].

Unfortunately, the ACPI model requires the AML content of an ACPI table to be implicitly trusted. ACPI events were designed to be OS independent and platform-specific, and hence the OSPM is missing the context to determine the semantics of the table content. As a result, the ACPI event is unconditionally processed and the AML code is executed. As first shown by Heasman [110], the ACPI tables could be used to inject a rootkit into a platform. Duflot et al. [68] later discussed this initial ACPI design flaw as well as presenting a sophisticated ACPI rootkit. Furthermore, Duflot pointed out that in the current ACPI design, neither the chipset hardware, nor the OSPM are able to guarantee that the AML content is legitimate.

## 4.2 Challenges

---

Thus, the ACPI tables as well as the OSPM must be included in the TCB as well as in the platform’s measurement chain as illustrated in Figure 4.2. While the ACPI tables can be integrated into the integrity checks during the SRTM, modifications at a later stage are likely to remain undetected. Duflot et al. [68] propose to cryptographically sign the tables, as well as statically or dynamically analyse the semantics of the ACPI tables. The Intel TXT model however, treats an ACPI power event as an attack and responds with a TXT-shutdown [103], potentially disrupting the power management on those platforms. We will further discuss how the impact of malicious ACPI tables can be mitigated via code segregation in Chapter 5.

Even tighter restrictions apply to the scope of ACPI table exploits: ACPI tables are not only machine and platform dependent but exploits also require an OS-specific, and potentially application-specific, implementation [68]. Nevertheless, the current ACPI implementation changes the trust implication of the previous TC model.

### 4.2.5 I/O Device Sharing

We discussed the difficulty of device and I/O virtualisation in Section 2.4.4. Moreover, we identified complexity as the single biggest issue in software security. The challenge is not only to efficiently share hardware resources, but also to provide sufficient guarantees that sharing resources does not undermine the isolation (4.1.2.1) or compatibility (4.1.2.4) requirements. A very pragmatic example is the sharing of a display adapter: whichever VM has control of the display adapter controls the output to the user whether it is a trusted VM or not. Hence an untrusted VM could mimic the output of a trusted VM or application.

There are many challenges to be met if devices are to be securely shared, and a large number of solutions have been proposed. Generally, device driver code is considered the greatest source of security bugs [18] and of poor quality [51], which disqualifies their addition to the TCB. A variety of solutions have been proposed to confine device drivers [12, 153, 176, 242, 256], or reconstruct a more secure I/O model [83, 232] to execute under the control of a low-level system layer. Additionally, DMA capable devices can read and write arbitrary memory regions, and hence standard drivers in an untrusted domain cannot be used without further safeguards. Recent advances in hardware support for an IOMMU introduced by Intel [1] and AMD [8], promise to mitigate at least the DMA issue, but do not address sharing

## 4.2 Challenges

---

in itself. Karger et al. [135] analyse the performance and security impact of I/O virtualisation, while Willmann et al. [285] discuss various DMA protection strategies and performance trade-offs for direct I/O access by untrusted VMs in detail.

The chosen I/O protection strategy greatly affects the level of intra and inter VM isolation, while at the same time has a significant impact on I/O performance and compatibility: a para-virtualised approach to I/O virtualisation reduces the cost and complexity in contrast to a full device emulation model, but requires the guest OS to cooperate and support special device drivers for new interfaces – thus impacting compatibility (requirement 4.1.2.5). The para-virtualised model however, currently only supports a limited class of device drivers – for instance networking and storage – due to the simplified interface design. Anderson et al. [12] demonstrate how device drivers can be removed from the TCB of the management domain and be isolated in special device driver domains. The separation and disaggregation of VMs has very similar properties to microkernels [153] or exokernels [76]. Nevertheless, as will be discussed in Chapter 5, in combination with hardware protection, this segregation significantly increases the isolation (requirement 4.1.2.1) a trusted virtualisation layer can provide. Ideally, as has been proposed by McCune [167], no drivers would be part of the TCB at all. However, the author acknowledges that the practical application of such an approach is limited.

### 4.2.6 Virtualising the TPM

In order to satisfy the attestation requirement (4.1.2.2) a VM must be able to cryptographically identify itself. On non-virtualised platforms remote attestation could be achieved with a Quote Command [263]. On a virtualised platform, the physical resources are not controlled by the OS anymore, but by a privileged entity. The existing TC model requires a one-to-one binding between the physical TPM and the OS. Virtualisation, however, decouples the binding of the physical platform and the virtualised OS, thus a different model is required to extend the existing TPM-based trust to VMs.

On one hand, the TPM was not designed for use in a virtual environment so new mechanisms for TPM sharing are necessary. On the other hand, in a process called migration, virtualisation allows VMs to dynamically move from physical platforms and therefore migration might break attestation. This section provides an overview of a number of proposed methodologies that enable a TPM to be shared amongst

## 4.2 Challenges

---

multiple VMs. Nevertheless, virtualising the TPM is an ongoing field of research and many issues are yet to be solved.

**vTPM as a guest service** - A software virtual TPM (vTPM) could, for example, be provided to guest applications via a TPM emulator [250]. This provides good isolation of the TPM resources from co-resident VMs, but compromises of the guest's ring 0 will also undermine the guest's TPM implementation. The problem arises from the complexity of the implementation and the lack of adequate protection within the VM. Moreover, the chain of trust assumes the platform to be measured prior to execution of the TPM emulator: hence the chain of trust cannot be initiated with a TPM emulator. This means that software TPMs offer the least amount of assurance and should only be used for debugging purposes. As the TPM is contained within a VM the TPM also becomes migratable.

Nevertheless, a TPM emulator allows a high amount of compatibility and flexibility, for instance by providing each VM with their own EK, AIK and SRK. Self-generated EKs may, however, be rejected by remote parties.

**vTPM as a management service** - One dedicated vTPM daemon receives TPM requests from every VM. The daemon is located in a special TPM or management domain (for instance Domain 0) and is responsible for tracking vTPM state changes and protecting secrets. This is the standard vTPM model as implemented by XEN [31].  $PCR_{[0-8]}$  are mapped read-only to VMs and reflect the state of the hypervisor and management domain, while  $PCR_{[9-15]}$  are available to VMs. Resettable PCRs might be exploited to trace VM state changes of virtual PCRs [75]. Secrets and vTPM states are contained within the memory and application context of the vTPM daemon in its corresponding domain (for instance Domain 0). Consequently, any compromises of the daemon process will reveal the states and secrets of all vTPMs. Moreover, the management domain must act as the RTM for all VMs and so must be trusted for all measurements provided to VMs.

In the para-virtualised model described by England et al. [75] all VMs share the same EK, which potentially breaks VM migration. Para-virtualised TPMs may provide a fully functional, but not compatible TPM interface [75] and therefore require a special back-end and front-end driver combination. However, exposing most of the TPM features to VMs directly, requires certain functions – for example clearing the TPM – to be safeguarded. The large TCB footprint of the management domain renders assurance in a TPM imple-

## 4.2 Challenges

---

mentation rather difficult, so England et al. [75] proposed the implementation of the vTPM model as a hypervisor service. Alternatively, the vTPM might be outsourced into a dedicated driver or service VM [12].

**vTPM as a hypervisor service** - Functionally, this is very similar to the aforementioned para-virtualisation approach, but with the difference of shifting the burden of vTPM management and emulation to the hypervisor. Even though the hypervisor, as discussed in Section 4.2.2, is generally a more trustworthy entity, similar trust implications as in the previous model have to be made: any compromises of the hypervisor potentially exposes virtual PCRs (vPCR) and states of the vTPMs. Moreover, the hypervisor must act as a RTM for VM measurements, thus adding more code to its TCB.

In Chapter 5 we propose an approach to the disaggregation of trusted code within the hypervisor, and discuss how this could be used to strengthen the isolation of a vTPM implementation. This allows for a much more fine-grained code separation and thus better protection of TPM contexts and secrets.

**vTPM as hardware extensions** - Hardware extensions to multiplex the different TPM resources offer the greatest amount of assurance for vTPMs. For example, a TPM could contain a simple VM context which is uniquely assigned to a VM and handled internally in the TPM. Stumpf et al. put forward such a multi-context TPM extension with loadable vTPM contexts [252]. However, this raises the questions – how many VMs need to be simultaneously supported, and how many keys does a TPM need to be able to hold? Moreover, it leaves the problem of VM migration unaddressed.

We already mentioned the iTPM implementation in Section 3.3. The iTPM as part of the platform’s chipset seems a suitable solution for a context-based TPM implementation. Moreover, an iTPM could simply include VM contexts in future firmware updates, allowing vTPMs to be successively deployed. Also an iTPM offers simple and flexible extensibility, for example by requesting an instant extension to the existing number of vTPMs or EKs from a corresponding authority. Virtual certificates or intermediate manufacturer signing keys could loosen the binding between hardware resources and allow an on-the-fly creation of valid EKs or SRKs for vTPMs. Adding hardware support for virtualisation to the TPM itself is probably the most straightforward approach; however such a solution currently seems years away.

### 4.3 Discussion

---

Virtualising the TPM holds many challenges and, as a result, is an ongoing field of research. For instance, for a VM to attest to its state it must also attest to the state of the virtualisation layer. This attestation could be achieved in an iterative attestation process, sometimes referred to as deep quote [75]. Also, migrating a vTPM potentially allows the cloning of a TPM, and therefore some aspects of the vTPM implementation might be deliberately designed to break migration. Protection strategies for internal state and cryptographic keys might also vary. Sadeghi et al. [226] demonstrate how a direct mapping of PCRs could be translated into abstract attestable properties. GvTPM on the other hand is a general vTPM framework with varying security and performance profiles [233]. While the TCG itself has established a virtualised platform working group, no publications or specifications are available at the time of writing. Further discussion on vTPM and its associated challenges can be found at [31, 75, 208].

### 4.3 Discussion

The previous sections have shown that virtualisation and TC technology are increasingly converging. There are compelling reasons for this convergence. Nevertheless, a trusted virtualisation infrastructure is more than the simple combination of these technologies. A sensible trusted virtualisation infrastructure needs more protection and guarantees than either of the two technologies can provide on their own. We identified that TC needs better underlying protection mechanisms to, on one hand guarantee isolation (requirement 4.1.2.1), while virtualisation needs greater assurance and attestability (requirement 4.1.2.2) of its isolation properties on the other. As applications are not able to protect themselves, they rely on the underlying OS for isolation guarantees. Without additional protection from software and hardware layers below, trust assumptions in commodity OSs do not hold.

TC and virtualisation seem to both complement and contradict each other simultaneously: TC tries to improve platform security by anchoring trust into physical, unchangeable platform properties, while virtualisation aims to provide as much hardware independence as possible. Moreover, complex software components might enable a hypervisor to act as a RTM to measure virtual hardware and software states of a VM. Complex hypervisor software however, might undermine the assurance in the quality of such measurements (requirement 4.1.2.4).

### 4.3 Discussion

---

A trusted virtualisation layer needs great confidence in the software layers below. In addition we require an owner to be able to enforce and control the platform and prevent unauthorised modifications by the platform operator. It is therefore vital to securely boot a defined hypervisor and management components to install further security services in order to satisfy the policy requirement (requirement 4.1.2.3).

Eventually, trusted virtualisation will become tomorrow's legacy and will potentially result in another layer of abstraction on top of the trusted virtualisation layer. Some of those issues have already become apparent, in the form of a recursive attestation (deep quote) as outlined by England [75]. Therefore maintaining compatibility (requirement 4.1.2.5) to past and future applications is an important property to satisfy for a trusted virtualisation layer. We will discuss the integration of legacy virtualisation into our improved hypervisor structure in Section 5.5.3.

Based on the previous sections, we have identified three key areas in trusted virtualisation we would like to discuss in the final chapters of this thesis: Chapter 5 lays the basic foundation to increase the trustworthiness of the lower hypervisor level. Chapter 6 builds upon the previous layer to provide trusted storage. Chapter 7 embraces the foregoing concepts to provide a trusted segregation of co-resident VMs. Therefore, the main contribution of this thesis is to address the three key issues described in detail below:

#### **Chapter 5** *Separating Trusted Computing Base with Hardware.*

For a meaningful attestation (requirement 4.1.2.2) as well as increased isolation guarantees (requirement 4.1.2.1), code which must be included in the TCB must be minimised (requirement 4.1.2.4). We outline in Chapter 5 how the TCB of a hypervisor can be separated using existing hardware features, to increase isolation and reduce the TCB. We further discuss various application scenarios, which allow trusted virtualisation to benefit from a strengthened policy decision point (requirement 4.1.2.3), as well as improved support for legacy applications (requirement 4.1.2.5).

#### **Chapter 6** *Trusted Virtual Disk Images.*

Especially in a virtual environment the traditional integrity and confidentiality assumptions for persistent storage no longer apply. A virtual hard disk image can be altered and copied without the knowledge of the legitimate owner. Consequently, Chapter 6 highlights how the integrity and confidentiality (requirement 4.1.2.1) of a virtual hard disk can be maintained throughout its

### 4.3 Discussion

---

life-cycle. Our Trusted Virtual Disk Image design allows the transparent application of these properties to satisfy any legacy concerns (requirement 4.1.2.5), as well as enabling the legitimate owner to retain control over the image and enforce policies (requirement 4.1.2.3), after the image's deployment. Moreover, we employ the TC binding and sealing mechanisms to allow a local or remote party to gain confidence in the Trusted Virtual Disk Image and virtualisation infrastructure being used (requirement 4.1.2.2). Finally, our design allows the storage subsystem to be removed from the TCB, which reduces complexity (requirement 4.1.2.4), and at the same time further increases isolation (requirement 4.1.2.1).

#### **Chapter 7** *LaLa: A Late Launch Application.*

On traditional commodity platforms, users are often forced to operate with full-blown OSs that only offer weak protection. Often, in a corporate environment, a platform is centrally managed and the users' abilities to operate these platforms are very restricted. In Chapter 7 we investigate how OSs at different trust levels can coexist to achieve a segregation of corporate and private VMs (requirement 4.1.2.1). By segregating trusted and untrusted components we are able to reduce the overall complexity (requirement 4.1.2.4), as well as enabling a more fine-grained attestation of trusted components (requirement 4.1.2.2). Additionally, the combination of legacy OSs with trusted components allows user experiences and compatibility (requirement 4.1.2.5) to be maintained. Finally, we integrate Intel's policy enforcement mechanism into our policy enforcement concept, to ensure only an authorised hypervisor and trusted components can be loaded (requirement 4.1.2.3).

This thesis investigates the combination of virtualisation, Trusted Computing technology as well as recent hardware advances on commodity platforms and outlines how those technologies could be utilised to move towards a more trustworthy virtualisation infrastructure. Trusted virtualisation holds many more challenges which cannot all be addressed within the scope of this thesis. We will direct the interested reader towards extensive literature on that subject in the subsequent section.

## 4.4 Related Work

Much work has been published in the area of Trusted Computing, virtualisation and the combination of these technologies. In the following sections we will identify related work which overlaps with the goals of this thesis. A specific examination of related work can be found in the discussion section in each subsequent chapter.

Virtualisation is frequently used to provide isolation; hence we review related work in Section 4.4.1 that exploits the functionality of hypervisors as well as microkernels to provide isolation. Virtualisation, however, often depends on complex software to provide isolation; consequently a large body of work has been published focusing on reducing the complexity and improving on the TCB of virtualised systems.

In Section 4.4.2 we discuss related work in this area. We also consider related work in Section 4.4.2.2 that exploits hardware protection mechanisms to improve the protection guarantees of a platform. While we certainly share the goal of improving the trustworthiness of virtualisation in many ways with commercial [204, 207], academic [86, 229] and non-commercial [77, 197] projects, we base our foundations on newly available hardware features, protection mechanisms and trust technologies available on commodity systems.

### 4.4.1 Virtualisation and Isolation

There exist numerous commercial and academic efforts to provide stronger isolation guarantees to commodity OSs. Two common approaches are virtualisation and microkernels. In the following paragraphs we discuss relevant work in these areas.

#### 4.4.1.1 Hypervisors

As pointed out in Section 2.2, virtualisation was first investigated as an isolation provider in the early 1970s [163]. The theoretical discussion by Madnick et al. [163] and the case study of the VM/370 integrity by Attanasio et al. [19], were followed by the efforts of Gold et al. [98, 99] to retrofit security to the VM/370 [55] in the form of the KVM/370. The endeavours of Gold et al. focused on using the hypervisor

## 4.4 Related Work

---

as a minimal isolation layer as well as reducing its TCB. We certainly share the goals and concepts of a minimal TCB with previous work in the mainframe area, but technology and tools have evolved over time and we therefore focus on state of the art commodity platforms with different requirements instead.

In the year 2000, NetTop [178] was proposed as a proof-of-concept hypervisor based on VMware which, in combination with multiple MAC-based OSs, aimed to consolidate government workstations. NetTop mostly depends upon the isolation guarantees provided by the MAC-based OS as well as the hypervisor. Unfortunately both suffer from large and complex TCBs. sHype [229] is a research hypervisor focused on IBM server hardware with the goal of providing strong isolation and MAC on a hypervisor level. Currently the basis for sHype is XEN – including hypervisor and management domain in its TCB – which adds to the complexity of the TCB. Additionally, policy decisions are placed in the management domain, which renders the value of those policy decisions questionable. SecVisor [238], is a small hypervisor that ensures code integrity for commodity OS kernels. It also aims to provide small code sizes to facilitate formal verification and manual audit. Bitvisor [242] proposes a new para-passthrough hypervisor design to enforce I/O security. However, both designs only support a single VM and SecVisor requires OS kernels to be specifically ported to its architecture which therefore disrupts compatibility.

Terra [86] is based on TC technology, exploiting the TPM to authenticate software running inside its VMs. Importantly, Terra allows legacy applications to run as so-called “open-box” VMs, while it provides VMs labeled as “closed-box” for security critical applications. The hypervisor is based on a VMware hosted OS which must be trusted; thus security properties derived from such a large TCB remain arguable.

Overshadow [50] builds on different views of physical memory depending on the VM performing the memory access. Overshadow cryptographically isolates an application inside a VM from the OS it is running on, essentially cloaking its memory context. While Overshadow aims to retrofit security in commodity systems by the use of virtualisation, we seek to improve the trustworthiness of virtualisation and its building blocks.

Qubes OS [224], is a recent effort by Rutkowska and Wojtczuk to provide strong security for desktop systems by using the isolation properties of virtualisation. Its security principles are based on secure system boot using TC technology and strong

## 4.4 Related Work

---

separation of security critical functions, such as networking, storage and user interface. The current way Qubes OS overlays and protects the integrity of its file system limits guest OS and application support to Linux. Qubes OS is in a very early of development stage and many of the design features, such as the use of TC technology, are planned but not yet fully implemented.

### 4.4.1.2 Microkernels

The use of microkernels as an alternative technology to enable virtualisation as well as an isolation provider has been extensively documented in the literature [112, 114, 217]. The following paragraphs summarise related work based on microkernels, with similar goals to ours.

EROS [239] is a fast, capability-based microkernel design for the x86 platform. It is the third implementation of GENOSIS [81] – a secure OS prototype initially designed for the VM/370. However, EROS requires applications to be ported to its architecture and interfaces, thus breaking compatibility. Singularity [116] is a microkernel which groups software isolated processes in the same address space and relies on static analysis to ensure secure communication between them. Singularity also requires applications to be specifically rebuilt to match its interfaces.

Nizza [109, 113] and PERSEUS [251] are both L4 [156] microkernel-based approaches towards reducing the TCB of security sensitive applications. Both leverage L<sup>4</sup> Linux [108] to provide additional support for legacy Linux applications. Heiser et al. [111] and Peter et al. [205] aim to reduce the TCB of legacy code by separating functionality into small servers and rely on a microkernel for isolation. Denali [283] is a lightweight isolation kernel which reduces its complexity by stripping the comprehensive emulation. Denali modifies the interfaces of the underlying platform, breaking compatibility in favour of performance and reduced complexity. One of our main objectives as stated in Section 4.1.2, however, is to provide compatibility for legacy and future applications.

One might detect from the current trend of disaggregating and outsourcing functionality of hypervisor-based virtualisation resemblances to a microkernel design. The dissection of the management and driver domain into small attestable modules, and the adoption of Inter Process Communication (IPC), are just two of many examples. In this thesis, we do not advocate or oppose hypervisors, microkernels

## 4.4 Related Work

---

or any other approach as the most appropriate solution for implementing trusted virtualisation. The following papers discuss the advantages and disadvantages of various solutions in great detail [107, 111, 217].

### 4.4.2 Trusted Computing

Many early efforts to realise TC on commodity systems required the creation of a complex chain of integrity measurements. The secure boot scheme proposed by Arbaugh et al. [15] and the trusted boot scheme by Sailer et al. [230] require software to rely on the previously executed pieces of code. The inclusion of complex and privileged software into the TCB led to various efforts to reduce, and exclude such code from the TCB. In the subsequent paragraphs we discuss related work on TCB reduction as well as recently proposed hardware protection schemes.

#### 4.4.2.1 TCB Reduction

The work of Schaefer et al. [234] in 1984 and the later extension in 1987 by Shockley et al. [243], discussed the partitioning and reduction of a TCB into smaller subsets. They argued that smaller subsets allow for a chain of simpler evaluations, which leads to the conclusion that the overall system is correct. Shockley et al. [243] discussed a hierarchical TCB design organised in protection domains. This is very similar to our proposal, discussed in Chapter 5, which can be regarded as an application of their discussion to x86 hardware protection.

More recently, P-Maps [227] and Flicker [168] have been proposed; both are efforts to reduce the TCB and improve runtime integrity of security sensitive applications. While the two approaches offer a secure execution environment utilising the DRTM, they both require applications to be specifically written for this environment. The TCB of such a design is certainly tiny, but this comes with a high development effort as such environments typically lack sophisticated debugging and testing capabilities.

The work of Murray et al. [188, 189] discusses how the TCB of XEN-based systems could be reduced by using disaggregation techniques as well as trusted libraries. Hohmuth et al. [114] proposed the extension of hypervisors with IPC mechanisms –

## 4.4 Related Work

---

a concept borrowed from microkernels. They reasoned that IPC in the hypervisor becomes necessary to allow interactions with untrusted components. The argument is similar to our concepts discussed in Chapter 5: adding functionality to the TCB – such as IPC – does not decrease the overall trustworthiness by default. Microkernels traditionally heavily exploit IPC and consequently are frequently used to separate and wrap code of different trust levels. However, we propose in Chapter 5, to utilise the available and unused hardware protection mechanisms on commodity systems.

### 4.4.2.2 Hardware Protection

Many different extensions and often radical new hardware designs, such as XOM [155], TIARA [244], AEGIS [254], Secret-Protected architecture [149], or the IBM 4758 crypto-card [71], have been proposed in order to improve protection guarantees. To some extent the TPM itself can also be regarded as a less radical platform extension as well. The major disadvantage of any new hardware architecture is the lack of support for legacy software as the cost of deployment and development effort for new software. Consequently, many researchers focus on less radical – and thus minor – changes to existing components in an attempt to maintain compatibility.

McCune et al. [169] suggest modification of the existing x86 hardware to increase performance of their minimal TCB code execution architecture. Their recommendations are based on the experience of their previous work [168], which unveiled performance penalties inflicted by the TPM and restriction of the DRTM.

Bratus et al. [36] also suggest modification to the x86 architecture in order to extend and reinterpret memory segmentation to provide kernel security isolation. While hardware enforced context isolation might be desirable, it requires recreation and reorganisation of existing applications to reflect the different contexts.

Simple modifications to an architecture are feasible, as chip manufacturers are increasingly dedicating CPU real estate to security functions. The addition of hardware support for DRTM by AMD [7] and Intel [103], as well as the recent integration of AES hardware acceleration on Intel chips [125], underlines this further. However, some modifications proposed do require changes to the foundation of the CPU design; some demand significant redevelopment and may also hinder the execution of legacy applications. Moreover, these schemes do not take into consideration the existing and unused hardware protection mechanisms, which we will discuss in Chapter 5.

### 4.5 Summary

In this chapter we have provided a high-level overview of the motivation behind creating a trusted virtualisation layer. We found that such a layer requires more protection guarantees than simply the sum of the building blocks of TC and virtualisation. We discussed that TC needs better underlying protection mechanisms to, on one hand guarantee isolation, while virtualisation needs greater assurance and attestability of its isolation properties on the other. Moreover, we recognised that hypervisors are rapidly increasing in complexity, and this trend might undermine the assurance in the quality of hypervisor integrity measurements. We identified basic properties which are required for trusted virtualisation as well as outlining a set of challenges which are inherited from the existing infrastructure. These challenges led us to formulate the problem definition which we address in the remainder of this thesis. Moreover, we also identified related work which overlaps in goals or motivation and additionally outlined how the discussed work can be distinguished from our proposal.

## Part III

# Improving the Trusted Virtual Infrastructure

## Chapter 5

# Separating Trusted Computing Base with Hardware

### Contents

---

<b>5.1</b>	<b>Introduction</b>	<b>102</b>
<b>5.2</b>	<b>Motivation</b>	<b>103</b>
<b>5.3</b>	<b>Background</b>	<b>104</b>
5.3.1	Trusted Computing Base	104
5.3.2	Ring Protection	105
<b>5.4</b>	<b>Trusted Computing Base for HVMs</b>	<b>108</b>
5.4.1	HVM Protection Rings	108
5.4.2	Code Separation	109
<b>5.5</b>	<b>Usage Scenarios</b>	<b>111</b>
5.5.1	Inter VM Communication	112
5.5.2	Virtual Private Networks	113
5.5.3	Legacy Virtualisation	114
5.5.4	vTPM Implementation	115
5.5.5	ACPI	116
5.5.6	Policy Control	117
<b>5.6</b>	<b>Considerations</b>	<b>118</b>
5.6.1	Performance	118
5.6.2	Development Effort	118
5.6.3	Device Sharing	119
<b>5.7</b>	<b>Discussion</b>	<b>119</b>

## 5.1 Introduction

---

5.8 Summary . . . . .	120
-----------------------	-----

---

*Everything should be made as simple as possible, but no simpler.*

– Albert Einstein

## 5.1 Introduction

In this chapter we explore how recent advances in virtualisation support for commodity hardware could be utilised to reduce the TCB and improve the code separation of a hypervisor. To achieve this, we reassess the definition of the TCB and illustrate how segregation of different code blocks could be enforced by hardware protection mechanisms. We argue that many software-based efforts at TCB reduction and separation can benefit from utilising those hardware capabilities. Furthermore, we outline and describe several virtualisation-based application scenarios, which will benefit from our proposal.

Numerous authors have suggested the use of virtualisation for security, fault isolation or assurance [49, 115, 163]. The argument for using virtualisation for security improvements is based on the reduction in the code base which has to be trusted. Rather than trusting an OS to isolate processes and share resources, in a virtualised system a hypervisor is entrusted with those tasks. As outlined in Section 4.1, a hypervisor consists of only a few hundred thousand lines of code as opposed to the millions of lines of code in a modern OS and thus potentially offers better security guarantees. Reducing the TCB is not a novel idea and is a well understood and discussed concept in system security research [114, 169, 189, 234, 243, 245]. With the recent resurgence of interest in virtualisation and its application in a security context, numerous authors [86, 169, 189, 245] have investigated how the TCB can be reduced and security critical functions could be isolated. This separation of privileged code, or disaggregation, is well known and is an actively discussed concept of, for example, XEN-based systems [188, 189].

Other authors have argued that a virtualised system is more secure than a non-virtualised system, despite adding code to the TCB [49]. This argument is based on the isolation properties and fault containment introduced with virtualisation. In

## 5.2 Motivation

---

this sense virtualisation is just a coarse-grained isolation scheme which, in contrast to traditional OS isolation, does not operate on a process, but below the kernel level. Arguably, the security gains are based on the fact that a hypervisor controls resources and executes beneath and hence outside the control of the traditional OS kernel. This, however, does shift the burden of separation and isolation from the OS kernel to the hypervisor. Currently, hypervisors are still considered more trustworthy than traditional OSs based on their smaller TCB, but they are growing fast in functionality and size. Consequently this trust may become misplaced and so alternative methods may be required to reduce the TCB.

In this chapter we explore the possibility of employing the currently unused hardware protection mechanism of modern CPUs. We suggest making use of those currently unused protection modes to separate and isolate security critical functions *within a hypervisor*. We want to emphasise that TCB reduction is not a new idea, but that recent developments in virtualisation technology have created new challenges as well as opportunities. Consequently, we want to seize the moment to learn from past mistakes and outline possible directions for future virtualisation research. We therefore illustrate the application of our concept to various security critical virtualisation services and demonstrate how it could be utilised to mitigate the security concerns raised in Chapter 4.

## 5.2 Motivation

Unfortunately, from a security perspective, functionality often takes precedence over secure design leaving security to be retrofitted afterwards. As outlined in Section 2.4.1.1 and 5.4.1, there are protection mechanisms in modern CPUs which are not utilised at all. The historical decisions taken not to use all protection mechanisms have resulted in legacy system support issues for today's designers. For instance, of four x86 protection modes available, only two have been used up until recently. Utilising the unused protection modes is possible but inflicts a significant amount of development and maintenance work on software developers.

On the positive side, CPU manufacturers are increasingly addressing security concerns in hardware and are willing to trade chip real estate for security functions. We discussed – in Section 3.2.7 – how modern platforms are able to support an alternative way to dynamically boot a trusted hypervisor or OS kernel. These

## 5.3 Background

---

platforms are able to dynamically reset their state without performing a complete reboot or initialisation [7, 103]. To enable the DRTM, the CPU, chipset and platform manufacturers had to accommodate many changes to the underlying hardware and software infrastructure.

However, a secure system still requires a harmonised collaboration between hardware and software components. The recent resurgence of interest and advances in virtualisation hardware development has created a window of opportunity for creative security solutions. Consequently, we propose to make full use of the existing protection mechanisms available in the new, hardware assisted, virtualisation chips at this early stage of hypervisor development.

## 5.3 Background

### 5.3.1 Trusted Computing Base

We provided a definition of TCB in Section 3.2.3, based on the ‘orange book’ [60]. In the following sections, we extend the previous definition and discuss improving the TCB in the context of hardware protection mechanisms.

Reducing the TCB is a well understood concept used to increase the trustworthiness and reliability of a computer system [12, 38, 114, 188, 189, 234, 243, 245]. The motivation behind reducing the code base of a system is twofold. Firstly, it is widely accepted that the reduction of code results in less errors and thus potential flaws that could compromise security [114, 188]. This is generally true if all code runs at the same privilege level. Secondly, evaluating or verifying code quality demands the use of small and well defined code blocks [46]. These code blocks should preferably be manually auditable. In reality, due to practicality and cost issues, verification of complex code such as OSs or hypervisors is very rare, but possible under some strict assumptions [144]. In 1987, Shockley et al. [243] described this approach as, “divide and conquer”, and argued a complex TCB can be divided into a subset of smaller components which can independently or incrementally be evaluated.

The work of Murray et al. [189] defines the TCB “as a set of code positions from which a privileged operation – one that can undermine the security of the system – may be invoked with arbitrary input.” The authors further concluded that all code

## 5.3 Background

---

which may undermine the system’s security should be trustworthy. The borders of the TCB also may become blurred, particularly when availability and robustness are paramount. A platform can contain code which is designed to detect, contain and recover from a failure. This code might not be security critical according to the previous discussion. However, this code provides the basis for an operational trusted platform. We therefore distinguish between ‘operational dependable code’ and ‘trusted code’. It is arguable whether operational code should be included in the TCB or not.

Further, it is difficult to evaluate the quality of a given code base. It is a common belief that the number of SLOC is a good measure of the quality of a TCB [114, 245]. This measure seems too coarse-grained and immediately raises the question of which lines of code to trust? Do all lines of code need to be trusted or is it sufficient to trust only code which performs sensitive operations? Moreover, it is also clear that poor interface design or poor code separation will not result in a more trustworthy system just because it has less code than any other system.

As discussed in the Section 5.4, the overall trustworthiness of a system can be elevated by separating and compartmentalising security critical functions, especially if those functions can be enforced by hardware.

### 5.3.2 Ring Protection

A generic overview on the ring protection scheme was given in Section 2.4.1.1. In this section we discuss how the different privilege levels are generally utilised by commodity OSs as well as by hypervisors.

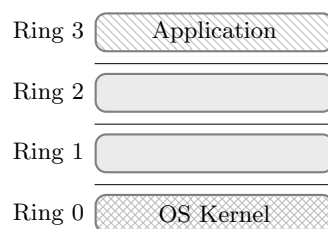


Figure 5.1: Traditional ring protection scheme.

Depending on which ring level code is being executed on, the code has access to different CPU functionality and features. Traditionally, an OS kernel – including

### 5.3 Background

---

device drivers – runs with the highest privileges in ring 0, with no access restriction at all. Applications are typically placed inside ring 3 at the least privileged level. Rings 1 and 2 are generally unused. Figure 5.1 illustrates the privileges in a non-virtualised context, whereas Figure 5.2 outlines the intended ring usage according to Grawrock [103]. Unfortunately, the monolithic design of commodity OS not only poses a security threat, but also turns out to be a technical hurdle for virtualisation.

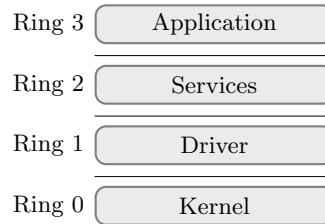


Figure 5.2: Intended ring use, according to Grawrock [103].

In a virtualised environment, a guest OS must not be allowed to run directly on hardware and interfere with the CPU state directly. Virtualising requires placing a hypervisor layer under the OS in order to trap privileged instructions. These instructions are then executed at a lower privilege level than they would be in a non-virtualised system. This mechanism is known as ring de-privileging or ring compression. Unfortunately, the x86 architecture contains several non-virtualisable instructions that could undermine the system’s stability and security [216]. Special safeguards have to be applied to those instructions and they have to be emulated by the hypervisor.

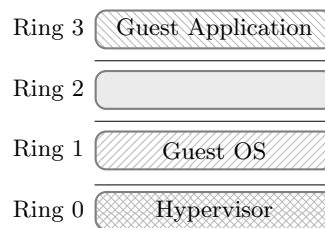


Figure 5.3: Ring deprivileging.

Figure 5.3 outlines the concept of ring de-privileging. In this scenario a hypervisor executes the guest OS at a lower privilege level, for instance ring 1. As we discussed in Section 2.4.2, ring de-privileging is not straightforward, as a hypervisor can be implemented in various ways. Technical details of and the challenges caused by ring de-privileging are beyond the scope of this thesis.

### 5.3 Background

---

To overcome the limitations of ring de-privileging and allow any unmodified guest OS to run without restrictions at its intended privilege level, CPU manufacturers introduced hardware support for virtualisation. The concept of hardware assisted virtualisation has already been discussed in Section 2.4.2. In the subsequent paragraphs we discuss the technical implementations of hardware assisted virtualisation which are relevant for the remainder of this thesis.

Hardware assisted virtualisation allows the creation of a Hardware Virtual Machine (HVM), under the control of a hypervisor and supported by hardware extensions to the CPU. HVM introduces a special mode of operation which allows the guest OS kernel to execute at its intended privilege level. Figure 5.4 illustrates the two different HVM modes available:

**Non-privileged mode** - This is a modified view of the CPU intended to accommodate virtualisation. In this mode OSs can now run at their intended privilege level. All privileged and some unprivileged CPU instruction will trap into the privileged mode.

**Privileged mode** - This mode remains the same as if running a native OS kernel; however a hypervisor is placed into the privileged mode to handle the traps from the non-privileged mode.

As outlined in Figure 5.4, the hypervisor in root mode also has four different privilege levels at its command. Throughout the literature, the privileged mode of operation is often referred to as ring -1. The transition between the two modes is controlled by hardware and the hypervisor. Transitions to the non-privileged guest kernel are referred to as *VMEntry* whereas transition to the privileged hypervisor are called *VMExit*s.

Because the author is more familiar with the Intel architecture, as well as having limited access to AMD documentation, we will focus on Intel terminology. The concepts and ideas are, potentially, applicable to any x86 architecture and AMD's virtualisation technology in particular.

### 5.4 Trusted Computing Base for HVMs

Previous efforts aimed at TCB reduction relied on software interfaces for segregation – wrappers, IPC, or trusted libraries for example. Instead of relying on software interfaces for code separation only, we want to discuss the extension of those concepts using existing hardware protection mechanisms. Hardware control mechanisms have the advantage of being self-contained and have a smaller attack surface, thus reducing the chance of subversion [235]. Often, hardware implementations tend to offer a performance benefit over software-only solutions.

With recent advances in hardware virtualisation support, the complete ring protection scheme also became available to hypervisors. Consequently, we argue that trusted code separation through software interfaces can be supported by hardware protection mechanisms. In addition, we want to discuss the opportunity to reconsider how the execution of hypervisor code can be separated to increase the overall trustworthiness of commodity systems.

#### 5.4.1 HVM Protection Rings

In Intel terminology the privileged mode is labelled VMX root mode and the unprivileged mode is called VMX non-root mode [270]. The Virtual-Machine eXtension (VMX), is a new hardware enhancement introduced to support virtualisation [120]. It is important to emphasise that the VMX root mode implements the protection scheme – described in Section 5.3.2 and illustrated in Figure 5.1 – whereas the VMX non-root mode can be regarded as a new mode of operation with reduced privileges. Both modes support all four rings. The guest OS is presented its expected ring model and additionally, the hypervisor is able to use multiple privilege levels [270]. In the following we use the terminology  $H0$  to  $H3$  for the rings in VMX root mode that are available to the hypervisor.

Throughout literature, the whole VMX root mode is often referred to as ring -1, implying that it is a new mode of operation added to the existing four rings. This is a common misconception as the VMX non-root mode is actually new. Figure 5.4 outlines the use of VMX root and VMX non-root mode.

## 5.4 Trusted Computing Base for HVMs

---

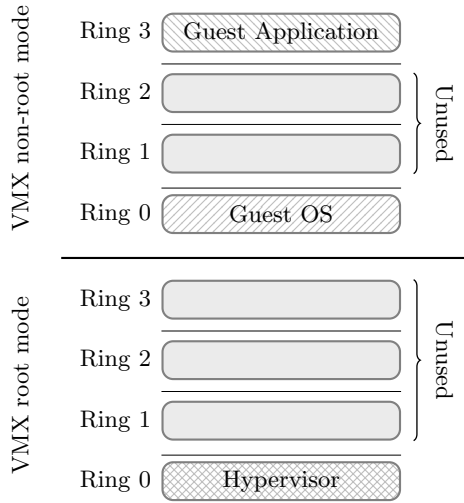


Figure 5.4: VMX root mode/non-root mode.

### 5.4.2 Code Separation

A more general approach to reducing the TCB is to separate trusted components from untrusted parts. Functionality which was previously part of the TCB can be executed outside the trusted context. This requires some form of interaction or collaboration between the two separated domains. To facilitate this, one well known approach is to implement a microkernel like IPC – for example via message passing over shared resources. Another approach is to use a layered system as described by Karger et al. [137] for the VAX architecture, in which each layer adds specific functions within a security kernel. Alternatively, trusted proxies or trusted wrappers can be used to decouple previously entangled components [114, 245]. The security gained results from well defined interfaces and limited interaction between the components on different trust levels. The addition of interfaces or processing capabilities also adds code to the TCB. This seems counterintuitive at first, however, adding IPC capabilities results in an overall decrease in the size of the TCB as other critical components can be outsourced.

As outlined in Section 5.3.1, disaggregation and code separation is beneficial as they lead to a smaller TCB. However, simply stripping a TCB of all non-security functionality seems a little naive. Furthermore, it is debatable what a “good” TCB consists of. For interfaces, according to Murray et al. [189], it rather depends on the size of the interfaces and the size of the TCB state space. Assuming the interfaces

## 5.4 Trusted Computing Base for HVMs

---

are secure by design and untrusted input is sufficiently sanitised, adding interface code to the TCB could increase the overall trustworthiness of a system.

We propose to add another parameter to the evaluation of the TCB – hardware separation. We can physically separate code, reduce the input sanitisation in the core TCB and therefore reduce the interface complexity. In the past there were only two places where code was executed – ring 0 and ring 3. As a result, any compromise of the most privileged context (ring 0), led to complete system compromise. Therefore, all code executed on ring 0 had to be trusted and included in the TCB, whereas code on ring 3 was generally regarded as untrusted. The introduction of a new hardware-based hypervisor mode has fundamentally altered this assumption.

To embrace the new hardware privileges, we would like to propose the following classification of TCB components. Our architectural model is illustrated in Figure 5.5.

### 5.4.2.1 Core hypervisor TCB

The core hypervisor TCB (hTCB) is located on ring 0 in VMX root mode *H0* and in essence remains the same as discussed in Section 5.3.2. However, the core hypervisor TCB consists of all security critical functionality which, if failing, would undermine the system’s security. Within the core hypervisor TCB are included those functions that are necessary to interact with the extended hTCB. This includes any means of implementing the interface as necessary and the associated input sanitising.

### 5.4.2.2 Extended hypervisor TCB

The extended hTCB is executed on either ring 1, ring 2 or ring 3 in VMX root mode, *H1*, *H2*, *H3*, and located outside the boundaries of the core hTCB. As depicted in Figure 5.5, potentially three different layers of extended hTCB can exist. Each of these extended TCBs will be represented by its own associated hardware protection ring. Each layer in this chain of extended hTCBs has to trust the layer beneath until the core hTCB is reached.

## 5.5 Usage Scenarios

---

### 5.4.2.3 Guest TCB

The guest TCB (gTCB) is the trusted codebase located in the VMX non-root mode. It contains the trusted code executed in the context of each VM (see Figure 5.5). Guest VMs have no knowledge of the functions implemented in the root mode, consequently the complete TCB of each VM is the sum of all TCBs beneath it. However, as VMExits trap to the hypervisor and thus the core hTCB, the extended hTCB is not always included in the overall TCB automatically. This will depend on which functionality has been implemented in the extended hTCB.

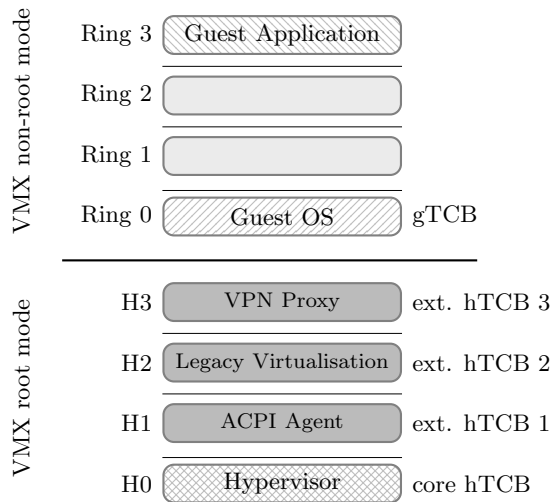


Figure 5.5: Conceptual overview.

## 5.5 Usage Scenarios

Bearing in mind that the VMX root mode operates in the same way as an unmodified CPU, we can use the ring model to construct a hypervisor system which consists of multiple components on different levels of privileges. Within the root mode, those components are isolated from each other by hardware mechanisms and not visible to any non-root mode code. Based on the intended ring usage as highlighted by Grawrock [103], we will discuss possible application on the four privilege levels. Most virtualisation management functions, for example handling VMExits and other vital emulation functions, have to remain in VMX root mode ring 0 (*H0*). However, other functionality could be outsourced to different privilege levels. To allow the hypervisor to communicate with outsourced modules on different rings,

## 5.5 Usage Scenarios

---

the hypervisor itself has to implement communication mechanisms, for instance via message passing or trusted libraries. This would preferably be something as simple as a traditional syscall interface for a H0 to H3 communication.

In the following we discuss four use-case scenarios which would greatly benefit from this approach: Inter VM Communication; Virtual Private Networks; Legacy Virtualisation; and Policy Control. More importantly, we outline how some of the virtualisation challenges described in the previous section could be mitigated by adopting a layered and hardware enforced hypervisor design.

### 5.5.1 Inter VM Communication

The isolation of VMs hosted on the same platform is one of the main properties of virtualisation. While isolation is important for security, it usually affects performance negatively. For instance, co-resident VMs are forced to communicate with each other using standard network interfaces and protocols as if they are resident on different physical machines. Those protocols were designed to operate on unreliable media and are not optimal for communication on the same platform. Therefore they inflict additional overhead in the form of integrity checks on multiple layers. Additionally, the network hardware used in VMs is often complicated when emulated in software.

Menon et al. [174] show that by carefully adapting existing protocols for inter VM communication, performance can be improved but is still not comparable to non-virtualised platforms. Consequently, solutions emerged trying to address this performance bottleneck. Many of the proposed solutions establish a shared memory link between VMs [140, 154, 279]. This link, however, weakens and potentially removes the isolation boundaries and assumes all VMs are equally trustworthy.

We envisage a proxy agent located on the previously vacant *H3* ring. This proxy can support multiple end-points and orchestrate the communication between VMs, operating in non-root mode, on different trust levels. Instead of using a management domain to do the emulation and setup of channels, this can be handled by the proxy agent instead. Firstly, this would de-privilege the management domain, stripping it of the above functions and, secondly, it would improve performance as the number of costly VMEntries and VMExits to and from the management domain would be reduced.

## 5.5 Usage Scenarios

---

Furthermore, the proxy agent is not part of the hypervisor itself – it is in that sense an application. It is not part of the core hTCB on  $H0$ , but part of the gTCB as seen by the VMs. The proxy on  $H3$  is isolated by hardware protection mechanisms from the hypervisor on  $H0$ . Therefore, from the hypervisor’s point of view, the proxy could be any untrusted code.

The major performance benefits however, are based on the fact that, once the CPU has entered the VMX root mode, the majority of the communication functionality can be handled in this mode without returning back to a VM and shifting data back and forth between different VMs. This essentially trades computationally expensive world transitions for less costly context switches. Nevertheless, new protocols are still necessary to circumvent the remaining performance bottlenecks, such as redundant integrity checks.

### 5.5.2 Virtual Private Networks

Virtual Private Networks (VPN), are self-contained private networks which rely on public untrusted networks for transport. VPN end-points protect the integrity and confidentiality of the private network’s data. Many VPN solutions are part of, or based on, commodity OSs such as Linux or Windows. For example, in IPsec-based systems the processing of network packets is typically implemented as part of the network stack. This has negative consequences for the TCB of the VPN end-point. Vulnerabilities in kernel or libraries can potentially be exploited and thus compromise the VPN. According to Singaravelu et al. [245] only a small proportion of code (<5%), in existing VPN end-point implementations is security relevant. Consequently, the authors propose to move the VPN functionality into a microkernel server to separate trusted from untrusted components. Also, Alkassar et al. [6], Schulz et al. [237], Berger et al. [32] and Cabuk et al. [43] have proposed the outsourcing of the VPN functionality into special service VMs. Rather than leaving the VPN management and security functions to the connection end-point, they propose to utilise a special VM to transparently connect VMs hosted on different physical platforms.

Instead of outsourcing the VPN functionality into a service VM or a microkernel server, we propose to place a VPN end-point proxy onto the VMX root mode  $H3$ . Figure 5.5 highlights the position of such a proxy agent. By doing so, the VPN functionality is not part of the guest’s kernel TCB nor part of the management domain’s

## 5.5 Usage Scenarios

---

TCB. Furthermore, it is separated from the hypervisor code as it is executed on a lower privilege level. Following the processing by the VPN proxy, any untrusted network stack could be used for transport. This resembles security properties similar to microkernels.

### 5.5.3 Legacy Virtualisation

Current research is focused on many aspects of virtualisation, such as performance enhancement, safe resource sharing and reducing the TCB. Future research will build on the foundations we lay today in the same way as we currently live with design decisions made in the past. Some of those past decisions have led to the necessity of complex software and hardware solutions, for instance retrofitting virtualisation into the x86 architecture. Today's hypervisors have to support legacy OSs in order to be usable for a mass market. In the same way future hypervisors are destined to support the execution of today's hypervisors, possibly even multiple copies of those. The second level, or nested virtualisation, is a known concept on mainframe computers, such as the IBM z/VM [161]. Nested virtualisation has also been explored by Mao et al. [48] to enhance grid security. The burden of overhead introduced by nested hypervisors became apparent as platform restrictions forced our Late Launch prototype – described in Chapter 7 – to resort to nested virtualisation.

There are multiple approaches imaginable to implement nested hypervisors on commodity platforms:

1. Software-based solutions could again resort to ring compression and use binary translation for the first level hypervisor. The scalability of ring compression would be very limited however.
2. Hypervisors could be made aware of nesting. Similar to para-virtualised OSs, there could be para-virtualised hypervisors. This could potentially work well for self-virtualising a hypervisor of the same kind or vendor. Unfortunately, achieving a heterogeneous hypervisor landscape demands well defined interfaces or standards and possibly access to the source code of a hypervisor. In a commercial world the likelihood of this happening is small.
3. Hardware assisted nested virtualisation of the hypervisor mode itself seems like the most promising approach. Hardware assisted nested virtualisation requires

## 5.5 Usage Scenarios

---

the hypervisor which is running closest to, and in control of the hardware to track the state changes and manage the virtualisation control structures. Due to the additional layers of virtualisation, the number of VMEntries and VMExits multiplies with each hypervisor and guest. Initially this will result in poor performance until hardware improvements can be made.

Hardware assisted nested virtualisation does not, however, implement execution flow control and assumes all hypervisors are equally trusted. Future applications might dictate the use of a more privileged control hypervisor running isolated from legacy hypervisors. Today, the first hypervisor to be installed is in charge of the physical platform and has to be trusted.

Alternatively, the unused protection modes in the VMX root mode could be used to support legacy hypervisors. As shown in Figure 5.5, a future hypervisor could be executed in *H0*, while legacy hypervisor functions or emulation is placed above in *H3* or even *H2* or *H1*. In this way, multiple virtualisation solutions – which already exist today – could be executed in parallel. This is potentially challenging to achieve as it might require standards, to avoid the need of retrofitting yet another hypervisor mode in hardware to support legacy hypervisors. Standardisation of a virtual PC interface is desirable so that one could install a generic hypervisor which executes different types of hypervisors and or OSs on top of it. Sadly, this is also quite unlikely to evolve in a commercial world.

### 5.5.4 vTPM Implementation

We have outlined challenges and possible solutions to virtualise a TPM in Section 4.2.6. While we identified a vTPM implementation based on hardware extensions as the most secure solution, such an extension will not be available in the foreseeable future. In the meantime, a para-virtualised approach based on hypervisor services offers a compromise between isolation guarantees and functionality. In the traditional hypervisor model there is no separation between services or drivers. Consequently, we propose to utilise the protection rings in VMX root mode for a vTPM implementation in order to differentiate between different security levels and to isolate code from non-TCB functions. A vTPM could be split up into a back-end part on *H1*, a service part on *H2* as well as a front-end part in the VMX non-root mode. While the back-end could keep track and maintain PCRs and keys with high confidence, the service part may contain world facing code capable of, for instance,

## 5.5 Usage Scenarios

---

providing guest measurements. From a functionality point of view this is similar to the design proposed by England et al. [74]. However, we propose to dissect their monolithic design and exploit the vacant VMX root mode rings to provide a layered design. This will allow us to separate TPM related code from less privileged code which must be contained in VMX root mode – for instance VPN code or legacy virtualisation.

### 5.5.5 ACPI

ACPI code is notoriously difficult to safeguard as highlighted in Section 4.2.4.2. Based on the ACPI design, power management needs to be handled by privileged system software. On commodity platforms, this is usually implemented by an OSPM, an AML interpreter and a device driver, which all hold kernel privileges. In the XEN model for instance, ACPI tables are interpreted by the management domain and thus hold the same privileges as the management domain. Since the ACPI tables are unconditionally executed and the interpreter, as well as the OSPM, runs with kernel privileges, this code cannot be trusted. As a result, Intel’s TXT extension treats a power management event as an attack and shuts down protected environments [103]. Depending on the protected environment this may result in a significant disruption of services.

While various safeguarding mechanisms have been discussed in Section 4.2.4.2, we believe separating privileged code from the OSPM, interpreter and driver greatly increases the isolation guarantees of the platform itself. As shown in Figure 5.5, the ACPI subsystem could be located on *H1*, effectively removing it from the management domain and isolating it. Unfortunately, this does not solve the ACPI issue in itself, but it may mitigate its effects. The ACPI would still be part of the gTCB as seen by the VMs, but the effects of a successful exploitation would be limited to the capabilities granted to *H1*. This approach could be used together with the previously discussed safeguarding mechanisms, for instance by cryptographically signing the tables – as proposed by Dufлот et al. [68].

## 5.5 Usage Scenarios

---

### 5.5.6 Policy Control

As a last example of how security may be enhanced through the use of the root mode, we consider the case of security access control mechanisms themselves.

Often implementations of access control mechanisms are split into two parts: the Policy Enforcement Point (PEP), and the security Policy Decision Point (PDP). For example, in the XEN sHype [229] access control framework, enforcement is carried out within the hypervisor itself but policy decision is handled within the management VM (domain 0). Obviously, this is not an ideal position to be in as we would like the security policy decisions to be as free from the risk of interference as possible. This is hard to guarantee if the decisions are handled within a general purpose OS such as Linux, rather than a trusted hypervisor.

The challenge here – and this is a general challenge when looking at reducing the TCB of a system – is that reducing code in the TCB is seen as beneficial but, from a security point of view, security decisions often need a policy context. In the case of hypervisors, maintaining context requires code which adds size and complexity and thus is usually kept out of the core of a hypervisor. The hypervisor only deals with enforcement mechanisms and actual security decisions are made by the code executed on layers above. If that decision making code is placed in a general purpose OS, then we have to question the confidence we can have in those decisions.

If we can hold sufficient context outside the hypervisor as well as outside any guest OSs or management domains, more assurance could be gained in the policy decisions. The VMX root mode is an excellent solution to host a policy decision end-point on *H1* to provide enough context for the hypervisor on *H0* to enforce it. The hypervisor-based policies provide enough information to impose on VM memory to enforce the presence and execution of specific applications, without executing the PDP in the VM's context. Consequently, more confidence could be gained in the overall security access control system. We use this property later in Section 7.4.6 in order to construct a layered PDP and PEP design.

### 5.6 Considerations

The architecture we propose has some similarities to the Kernel-based Virtual Machine (KVM) [146] model. In the KVM model, the Linux OS is executed in root mode with the kernel placed on  $H0$  and user applications, such as guest emulation, residing at  $H3$ . Guest OSs are executed in non-root mode at their intended privilege level, while the VM as such is a userspace task in VMX root mode. This does however, have similar trust implications as the XEN model since it too includes a general purpose OS in the TCB. Additionally, in a para-virtualised environment such as XEN, it is not trivial to separate the hypervisor and the management Domain 0 by hardware. Section 4.2.3 provided a discussion on the inclusion of a management VM into the TCB. This results from the direct hardware access privileges held by the Domain 0. Code which accesses physical resources needs to be contained by an IOMMU, or as discussed previously, executed in  $H1$ ,  $H2$  or  $H3$ .

#### 5.6.1 Performance

The goals of security and performance often appear incompatible. However, our approach has one very appealing aspect in this context: The transition between VMs and hypervisor via VMExits and VMEntries on current generation hardware still inflict a significant overhead [4]. We can trade the expensive VM-to-hypervisor transition against context (mode) switches in the VMX root mode. Context switches are computationally not cheap, but they are significantly faster than VM-to-hypervisor transitions on current generation chips. We are aware that this might change with further advances in hardware virtualisation technology, for instance hardware assisted page tables. However, a detailed performance analysis is beyond the scope of this thesis.

#### 5.6.2 Development Effort

There will undoubtedly be a development effort associated with utilising the VMX root mode protection mechanisms. The magnitude of changes necessary to the hypervisor and the amount of development effort for the hypervisor applications will certainly depend on the usage scenario.

## 5.7 Discussion

---

### 5.6.3 Device Sharing

Device drivers remain a difficult problem, because they have access to physical resources. In a virtualised environment, efficiently sharing physical resources often requires the inclusion of device drivers into the overall TCB. On systems without DMA protection, device drivers have potentially unlimited access to all physical memory and must be safeguarded. More importantly, past research has shown that device drivers are error prone [51]. DMA protection is becoming available on commodity systems in the form of an IOMMU. However, whilst an IOMMU allows the safe direct assignment of a device to a VM, it does not solve the issue of device sharing. The specifications defined by the Peripheral Component Interconnect Special Interest Group (PCI-SIG) [201], promise to mitigate the issue of device sharing, but this is not yet widely available.

## 5.7 Discussion

Apart from the previously discussed KVM, we are not aware of any other work which proposes the use of the CPU protection scheme in the VMX root mode, hence the application of those protection schemes for TCB code separation or reduction has not been previously debated. We have however, pointed out related work on reducing the TCB as well as extending existing hardware to protect the TCB in Section 4.4.

The assumption that a hypervisor is in sole charge of all hardware resources and the most privileged piece of code is paramount in order to express any trust assumptions. Moreover, the hypervisor might be subverted by malicious code, such as a root-kit, which presents emulated hardware to the hypervisor. Techniques, such as the Intel's TXT, promise to mitigate this issue by authorising code before execution. Unfortunately, Intel's TXT is fairly new technology and – as outlined in Section 4.2 – has multiple implementation issues.

As stressed in Section 4.2.4, most of the recent security issues with hardware virtualisation and Intel's TXT in particular are based on platform limitations such as ACPI and SMM. While it is difficult to safeguard the SMM without support by the platform manufacturer, we have seen previously that it is possible to limit the ACPI handler's effect on the TCB, by utilising vacant protection mechanisms.

## 5.8 Summary

---

As already outlined however, the assumption that the hypervisor is the most privileged code remains paramount for any sensible security assumptions.

While our proposal is not revolutionary, we believe that the current advances in hardware virtualisation technology have created a window of opportunity to learn from mistakes made in the past. As described by Grawrock [103], the intended privilege usage of the x86 protection scheme has never been implemented. This has had a profound impact on the TCB and security of commodity OSs which still affect us today. Redesigning legacy OSs is certainly possible but, as pointed out by Chen [50] et al., retrofitting security is difficult and potentially expensive.

## 5.8 Summary

In this chapter, we proposed full utilisation of the available hardware protection mechanisms to increase the overall trustworthiness of a virtualised system. We also extended the existing definition of TCB and applied the terminology to the new hardware protection mechanisms. Based on the requirement discussion in Section 4.1.2, we have demonstrated that our concept will help to increase the isolation requirement (4.1.2.1), by separating code on different trust levels, whilst at the same time reducing the overall software complexity (4.1.2.4). Moreover, the smaller code blocks help to increase the value gained from attesting to the TCB and thus help improve on the attestation requirement (4.1.2.2). Additionally, we discussed how the risk of policy decision interference (requirement 4.1.2.3), in a virtual environment can be reduced by using the available hardware capabilities. By exploiting the unused protection modes in virtualisation-enabled CPUs, we can rely on hardware protection, while at the same time are able to provide backward compatibility to existing legacy systems (requirement 4.1.2.5). We have therefore demonstrated how the hardware protection mechanisms could be utilised in various real-world examples in respect of the previously discussed requirements. Finally, we have pointed out some of the obvious limitations of our design.

# Chapter 6

## Trusted Virtual Disk Images

### Contents

---

<b>6.1</b>	<b>Introduction</b>	<b>122</b>
<b>6.2</b>	<b>Motivation</b>	<b>123</b>
<b>6.3</b>	<b>Security Requirements</b>	<b>124</b>
6.3.1	Threats	124
6.3.2	Design Goals	125
<b>6.4</b>	<b>Trusted Virtual Disk Images</b>	<b>126</b>
6.4.1	Assumptions	127
6.4.2	Ensuring Confidentiality	128
6.4.3	Integrity Protection	129
6.4.4	Creating Integrity Metrics	131
6.4.5	Policy Model	135
6.4.6	Software Enforcement	135
6.4.7	Metafile	136
<b>6.5</b>	<b>Life Cycle</b>	<b>139</b>
6.5.1	Initialisation	139
6.5.2	Deletion	139
6.5.3	Backup	140
6.5.4	Sparse Format	140
6.5.5	Migration	141
6.5.6	Snapshots	141
<b>6.6</b>	<b>Security Analysis</b>	<b>142</b>
<b>6.7</b>	<b>Considerations</b>	<b>145</b>

## 6.1 Introduction

---

6.7.1	Fragmentation . . . . .	145
6.7.2	Performance . . . . .	146
6.7.3	Swap-space . . . . .	146
<b>6.8</b>	<b>Usage Scenario . . . . .</b>	<b>146</b>
<b>6.9</b>	<b>Discussion . . . . .</b>	<b>147</b>
<b>6.10</b>	<b>Summary . . . . .</b>	<b>148</b>

---

*Gentlemen do not read each other's mail. (sic)*

– Henry Lewis Stimson

## 6.1 Introduction

While in Chapter 5 we addressed isolation on a hypervisor layer, in this chapter we will investigate how the services above the hypervisor layer could be improved. We therefore discuss how the trustworthiness of a storage layer in a virtual infrastructure could be improved.

In general, Virtual Disk Images (VDI) represent the hard drive of a VM encapsulated in a single entity. The disk image can contain a whole OS, data and/or applications. Often, this entity is one single file, which reduces the complexity of VDI handling to that of handling files or folders. For the VM this also means that it is irrelevant whether the disk image is located on the host's physical hard drive, mounted via a network share or hosted over the Internet. The location of the image and its access method is transparent to the VM.

The loose binding of VM and VDI poses a new set of security challenges – a VM cannot make any assumption about the integrity nor the confidentiality of its own hard drive content. As previously discussed [88, 93], one particular threat to VDIs is the fact that they can be copied without the legitimate owner's knowledge. If the image is copied or stolen, confidential data is at risk. Moreover, data could be altered deliberately or accidentally during storage or transit without the knowledge of the VM. To some extent this threat may be mitigated by cryptographically protecting the integrity of individual files within the guest OS. However, not all applications

## 6.2 Motivation

---

employ cryptographic technology and often assume that the execution environment itself is secure.

Other applications often rely on non-keyed hash functions or checksums to protect integrity, assuming that attackers do not have access to the integrity metric nor the generation mechanism. Depending on the specific circumstances, these may be reasonable assumptions for a non-virtualised OS or applications which run in a restricted and controlled environment. However, once virtualised, the VDI may leave this controlled environment, be copied and made available to an attacker to manipulate off-line. The traditional security concepts, which assume a static hard drive with limited or no physical access by the adversary, allow many non-cryptographic integrity protection mechanisms applied within the VM to be circumvented.

In this chapter we present a secure, flexible and transparent security architecture for VDIs. In our security concept, VMs transparently benefit from integrity as well as confidentiality assurance. We base our concepts on TC utilising the TPM to efficiently deliver integrity assurance to VDIs. This enables us to provide a secure and flexible Trusted Virtual Disk Image (TVDI) infrastructure to a broad number of platforms. Further, we allow a restrictive rule-set to be imposed by the TVDI owner and we enable the owner to retain control over the TVDI throughout its life-cycle.

## 6.2 Motivation

Disk encryption software in general – such as dm-crypt [64], TrueCrypt [268], FileVault [14] and Bitlocker [182] – is publicly available but does not provide integrity protection. Therefore, our work was mainly motivated by the fact that many VDIs reside on untrusted storage and are possibly subject to malicious or accidental modification. For instance, if a VDI is carried on a laptop device and left unattended, the device might be subject to an “evil maid attack”, in which alternative boot media is used to bypass standard access control. Moreover, VDIs might reside on shared local or remote storage in which a local administrator is not considered trustworthy. Such attacks may be mitigated by using TC and an RTM to assure a certain platform state. However, as discussed by Turpe et al. [269] attacks are still possible, even if the laptop is protected with full disk encryption. We will investigate a more dynamic and flexible security concept for VDIs in the following sections.

## 6.3 Security Requirements

---

Additionally, some hard disk encryption solutions are tightly tailored to a specific OS or use case, and do not meet the flexibility requirements of a virtualised infrastructure. For example, FileVault is only available on Apple's OSX and does not support full disk encryption. Bitlocker has support for a TPM and full disk encryption but is a commercial product which only targets Windows platforms. Moreover, the use of existing software-based disk encryption often restricts compatibility, transparency and manageability. None of the existing products have yet targeted the flexibility and transparency requirements of protecting disk images in a heterogenous virtual environment.

## 6.3 Security Requirements

### 6.3.1 Threats

VDIs are most often represented by a single large file or a set of files. A set of files is often used to overcome file size restrictions on legacy file systems – for instance FAT-32. The VDI is then exported by the hosting environment to the VM as a physical hard drive. The handling of a VDI is similar to the handling of any type of file, hence it can simply be copied, moved or altered much like any other file. Without appropriate protection mechanisms a VDI can be manipulated, injected with malicious code or leak sensitive data without the knowledge of the legitimate owner. We have classified the threats further into the following:

**Image manipulation** - A malicious party could alter sensitive data within the image or even inject code into a disk image without the knowledge of the legitimate owner or subsequent user. Image manipulation can take place even in the presence of encryption protection mechanisms.

**Information leakage** - Confidential information, such as corporate secrets and credentials, might be extracted from an image. Moreover, a disk image may contain several snapshots each of which may be reloaded by an attacker. This provides a good basis for further replay attacks, for example against security protocols [88].

**Image replacement** - With no data origin authentication an attacker may replace a file containing image data without the knowledge of the legitimate

## 6.3 Security Requirements

---

owner. This could be exploited to force the reuse of previously discarded encryption keys.

It is rather difficult to mitigate the threats mentioned above with traditional images while maintaining interoperability and backward compatibility. Generally, an image may be considered at risk if moved outside the secured borders of a data centre.

### 6.3.2 Design Goals

To mitigate the above threats, our three main security goals are:

1. Provide data confidentiality
2. Provide data integrity
3. Enforce a specific hypervisor and management VM for a particular image

Traditional OSs often lack the ability to sufficiently isolate their storage content from unauthorised access. We discussed, in Section 4.2.1, the manner in which most commodity OSs implement a DAC model, which imposes an all or nothing design. The DAC model makes it very difficult to protect VDIs from administrative accounts or malicious code acting on their behalf. The integrity and confidentiality protection of the VDI allows us to enhance the isolation requirement, discussed in Section 4.1.2.1.

We want a remote party to be able to gain confidence in both the hypervisor and the VDI used by a given VM in order to improve on the attestation requirement (4.1.2.2). As outlined in Section 3.2.4, a TC differentiates between owner, user and operator. Thus the image owner who provides the VDI for execution by the user or operator may impose policies on the image and limit the capabilities of the user and of the operator. Our goal is to allow the owner (provider) to retain control over the image once deployed and prevent modifications from local administrative accounts or malicious entities, in order to satisfy the policy requirement (4.1.2.3).

It is our goal to provide these security mechanisms transparently to the VM in order to maintain compatibility to existing legacy systems (requirement 4.1.2.5).

## 6.4 Trusted Virtual Disk Images

---

Our major concern is security but we are also anxious to keep any impact on implementation and performance to a minimum.

## 6.4 Trusted Virtual Disk Images

Our design of a TVDI is built upon the existing virtual disk (`block tap` or `blkmap`) driver currently integrated in XEN [282]. Figure 6.1 outlines the system design concept of a VDI in XEN, while Figure 6.2 depicts a more detailed implementation of our TVDI. The `block tvdi` is our module extension to the existing `block tap` driver and, together with the `Trust ctrl`, realises our TVDI concept. Our contribution is highlighted by a north-west line pattern in Figure 6.2. We incorporate our `block tvdi` module into XEN's existing `block tap` for the following reasons:

- XEN is open-source based and publicly available.
- An existing and well documented virtual disk driver exists.
- The existing driver offers support for portable user-level back-ends.
- Userspace tools and libraries can be used.
- XEN offers a good range of compatibility.

The `blkmap` driver is a modular userspace implementation of a virtual block device, which enables us to easily extend the `blkmap` with our proposed `block tvdi` module. Additionally, it allows us to use existing userspace tools and libraries, which minimises implementation overheads whilst preserving compatibility with existing installations.

In the XEN model, Domain 0 (Dom 0) represents the management and control domain and thus the most privileged domain. It manages virtual devices as well as administrative tasks such as suspension and migration of guest OS domains. In XEN terminology, guest OS domains are referred to as “user domains” or Dom Us. In the remainder of this chapter we adopt the XEN terminology to allow readers familiar with this terminology, to easily comprehend the details of our TVDI implementation.

XEN allows both an emulated access and a para-virtualised access to the disk interface. In the following sections we will however, focus on the para-virtualised

## 6.4 Trusted Virtual Disk Images

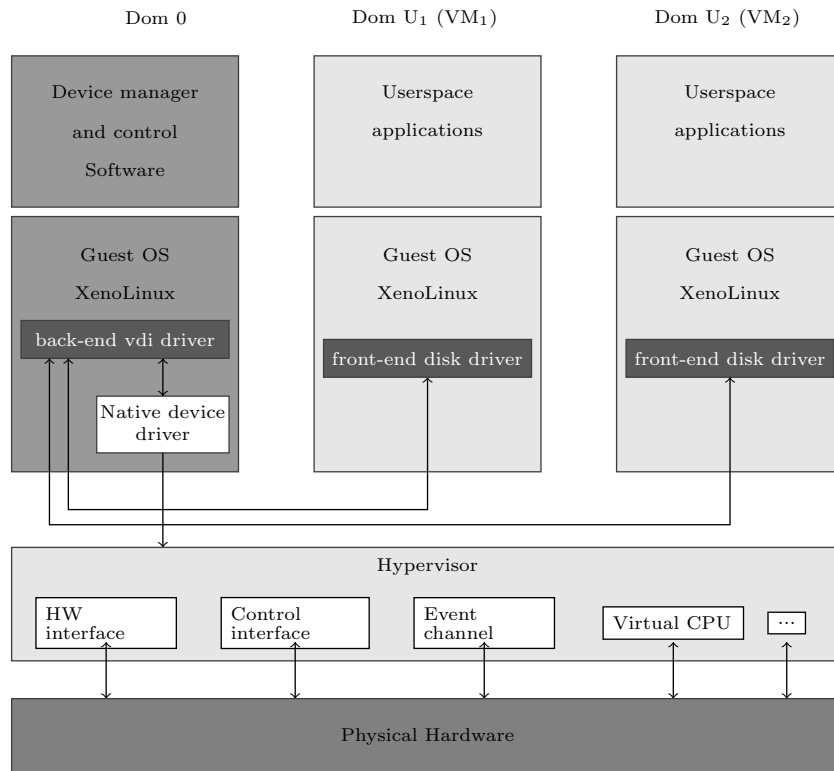


Figure 6.1: Sample VDI implementation in XEN, based on [25].

approach in favour of simplicity. Modifications are limited to the `block tvdi` module to ensure all existing Dom U remain compatible. Nevertheless, our concepts and designs are easily applicable to an emulated access and are by no means limited to XEN.

### 6.4.1 Assumptions

We based our TVDI design on existing code and on XEN [210] in particular and require the following assumptions to be fulfilled:

**Assumption 1** - Hardware virtualisation features and protected memory – as described in Section 2.4 – are available. This becomes necessary due to a shared address space configuration in the existing XEN back-end communication implementation.

## 6.4 Trusted Virtual Disk Images

---

**Assumption 2** - An integrity measurement architecture is present on the host and capable of measuring the userspace applications. We make use of the concepts of TC, and the TPM in particular, as described in Chapter 3. These will be used to report the system state of a hosting environment and additionally to perform sealing operations on a metafile. The VM does not necessarily need to be aware of the presence of a TPM as the VDI implementation on the hosting system is utilising the TPM's functionality.

**Assumption 3** - The hypervisor and a set of userspace applications are included in the chain of trust and the system is running in a pre-defined state. We assume that the hypervisor and the management domain is trusted to only execute code approved by the owner. Here we build upon our layered approach and rely on the hypervisor design described in Chapter 5, as well as on the DRTM and Trusted Management Domain concept discussed in Chapter 7.

**Assumption 4** - A trusted third party is existent and trust between a control application and the third party can be established. We utilise the trusted third party as a TVDI control entity, which is in charge of initialisation, migration, backup and deletion of a TVDI.

The actors, libraries and actions, summarised in Table 6.1, are involved in the following description of the TVDI framework.

### 6.4.2 Ensuring Confidentiality

Confidentiality for images is provided by encrypting each chunk of the image with a suitable encryption algorithm. The key for the image may be held in a sealed metafile. XEN already implements the QEMU [213] disk image format (qcow), which integrates 128-bit AES encryption [173]. The qcow disk image format is a versatile copy-on-write image format for the open-source emulator QEMU. However, in favour of flexibility we propose the use of *libcrypto* which offers a broader range of encryption algorithms.

Since the metafile itself is sealed to a TPM and a defined system state, only the system in a particular state may reveal any encryption keys stored in the file. In practice this allows the chunks and the metafile to reside on untrusted storage and traverse unsecured networks. Confidentiality can therefore be transparently pro-

## 6.4 Trusted Virtual Disk Images

---

Actors	Description
<code>Blktap</code>	Userspace application
<code>block tvdi</code>	Module extension to <code>Blktap</code>
<code>blktap ctrl</code>	Character control device
<code>trust ctrl</code>	Userspace application to manage trust
Libraries	Description
<code>libcrypto</code>	Cryptographic library
<code>libTPM</code>	TPM library
<code>libaio</code>	Asynchronous I/O library
Actions	Description
<code>hash()</code>	One-way hash function
<code>read()</code>	Function to read files
<code>write()</code>	Function to write files
<code>seal()</code>	TPM-based sealing function
<code>unseal()</code>	TPM-based unsealing function
<code>decrypt()</code>	Symmetric decryption function
<code>encrypt()</code>	Symmetric encryption function

Table 6.1: Actors, libraries and actions involved in TVDI operation.

vided to VMs while existing, potentially insecure, legacy storage and communication structures may be used.

### 6.4.3 Integrity Protection

In order to protect the integrity of a disk image, integrity metrics of the image have to be calculated, stored and subsequently checked. As mentioned in Section 6.3.1, a traditional VDI consists of a large file or files. It is difficult to measure the integrity of these files in a practical and timely manner.

Read-only images on one hand can be deployed with pre-calculated integrity metrics, but also require calculation of the image's integrity to be able to compare it to expected values. With a writable image on the other hand, every write operation performed on the image will result in different integrity metrics. Timeliness therefore becomes an important consideration if the image is likely to be updated frequently during execution of applications. To avoid a constant rehashing of the VDI we suggest compartmentalising the image, by splitting it into smaller chunks of a fixed length, and using a set of hash values to represent the integrity metrics of the whole disk image. Thus only the integrity of those elements which have been updated need

## 6.4 Trusted Virtual Disk Images

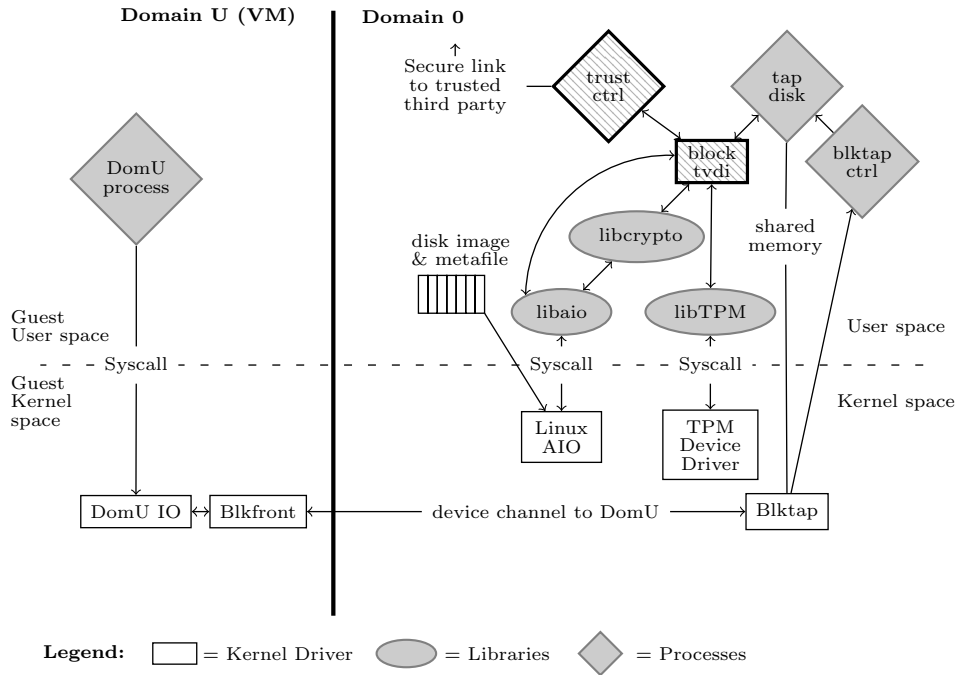


Figure 6.2: Sample TVDI implementation in XEN, based on [52].

to be recreated so we are able to perform integrity measurements in parallel. Figure 6.3 illustrates how such a disk image is constructed.

Rather than trading calculation time at the system’s start-up for calculation time during runtime by applying a Merkle hash tree [177] over the image, we decided to compartmentalise the image into independent chunks. While a Merkle hash tree would allow the disk image to be used immediately, it requires to check the tree structure during operation. Our proposal can be regarded as a flat hash tree with the depth of one layer. As a consequence, chunks can be treated separately. This allows us to reduce the number of hash operations necessary and, additionally, enables us to implement an efficient snapshot functionality as described in Section 6.5.6.

An alternative approach might be to separate read-only from writable sections of the image. However, we believe that splitting up an image into chunks has certain advantages over separating read-only from writable data on independent images. Firstly, we ensure full interoperability and compatibility with existing systems as well as support for legacy OSs which do not support execution from a read-only image. Secondly, we benefit from a performance gain through the parallel processing of only those chunks that need to be updated of necessity.

## 6.4 Trusted Virtual Disk Images

---

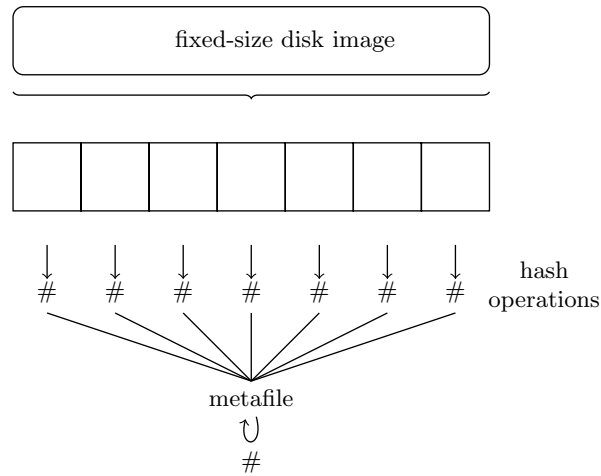


Figure 6.3: Overview of integrity construction.

### 6.4.4 Creating Integrity Metrics

We propose the use of a metafile – as described in Section 6.4.7 – to store the integrity metrics of each individual chunk of a VDI. Information about modification time and the corresponding chunk’s hash value are stored in this metafile. The metafile itself is protected by the TPM’s sealing capabilities and bound to a distinct platform in a specific configuration. The userspace and kernelspace, as parts part of the back-end driver, communicate via named pipes and shared memory [282]. Figure 6.4 outlines the detailed TVDI execution flow for the creation of integrity metrics as numbered below. The creation of integrity metrics is triggered by a write operation of a Dom U to its virtual block device and takes place as described in the following steps:

1. `block tvdi` calls the `unseal()` function in `libTPM` to unseal the metafile. The metafile is sealed to a CMK and a set of PCR values as described in Section 6.5.1. After being unsealed the metafile is kept in the main memory until the Dom U is finally paused or halted. The memory content itself is protected from malicious DMA or unprivileged access by employing an IOMMU.
2. `block tvdi` calls the `hash()` function in `libcrypto` to compute the integrity metrics of the metafile and compares the result against the expected value from the metafile. The metafile and its content are discussed in Section 6.4.7.

## 6.4 Trusted Virtual Disk Images

---

3. `block tvdi` maps the chunks as a block device for a Dom U – according to the mapping contained in the metafile. The mapping is described in more detail in Section 6.4.7.
4. The `block-frontend` in Dom U forwards the I/O write request to the `block tvdi` via a special character control device (`blktap ctrl`).
5. `block tvdi` looks up the chunk affected by the write request according to its mapping.
6. `block tvdi` calls the `read()` function in `libaio` to load the chunk’s data.
7. `block tvdi` calls the `decrypt()` function in `libcrypto` to decrypt the chunk and performs the requested write operation.
8. `block tvdi` calls the `encrypt()` function in `libcrypto` to encrypt the chunk and updates the integrity metrics of the chunk in the metafile.
9. `block tvdi` calls the `hash()` function in `libcrypto` to generate the new integrity metrics of the chunk.
10. `block tvdi` calls the `write()` function in `libaio` to write the changes to persistent storage.

This design allows transparent and concurrent operation for multiple VMs. Implementation overheads are minimal and limited to our `block tvdi` module and `trust ctrl` application. Userspace libraries such as `libcrypto` and `libaio` are widely available – `libTPM` is obtainable, for instance, via the TrouSerS software project [267]. We are aware that our concept will result in additional read/write operations that will impact performance but we consider this a fair compromise for the security assurance gained in return.

### 6.4.4.1 Checking Integrity Before Operation

Integrity checks of the TVDI can be performed all at once, thus before the VM is able to access the image. Before starting, resuming or migrating a VM the integrity of all chunks is checked against their pre-stored values from the metafile. Hashes of individual chunks are independent, which allows the hashing operations to be performed in parallel. In comparison to a traditional approach to hash a single

## 6.4 Trusted Virtual Disk Images

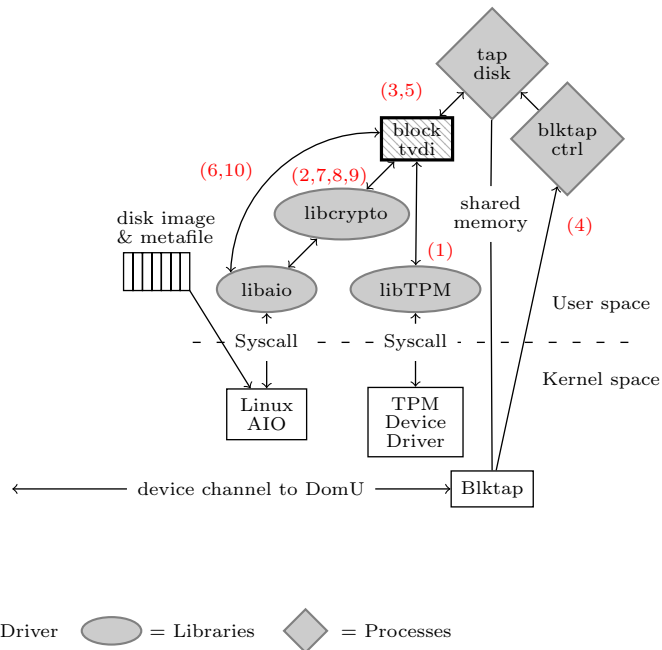


Figure 6.4: Creating integrity execution flow.

large file, our approach benefits from the increased speed gained by parallelisation. Moreover, as explained in the following section, it is not necessary to check an image before operation but it is possible to measure it during operation and even recover from integrity failures.

### 6.4.4.2 Checking Integrity During Operation

I/O read requests from Dom U are forwarded to the Dom 0 via the `blkctl ctrl`, as outlined in Section 6.4.4. It should be noted that steps 1-6 are the same. The following steps describe how the integrity of the TVDI's chunks are compared against their expected values in the metafile:

1. `block tvdi` calls `unseal()` function in `libTPM` to unseal the metafile.
2. `block tvdi` calls the `hash()` function in `libcrypto` to compute the integrity metrics of the metafile and finally compares the result against the expected value from the metafile.

## 6.4 Trusted Virtual Disk Images

---

3. `block tvdi` maps the chunks as a block device for a Dom U according to the mapping contained in the metafile.
4. The `block-frontend` in Dom U forwards the I/O read request to the `block tvdi` via a special character control device (`blktap ctrl`).
5. `block tvdi` looks up the chunk affected by the read request according to its mapping.
6. `block tvdi` calls the `read()` function in `libaio` to load the chunk's data.
7. `block tvdi` calls the `hash()` function in `libcrypto` to compute the integrity metrics of the chunk.
8. `block tvdi` compares the chunk's computed integrity metrics to the value contained in the metafile.
9. `block tvdi` performs the policy enforcement as described in Section 6.4.5.
10. `block tvdi` calls the `decrypt()` function in `libcrypto` to decrypt the chunk's data.

Assuming an enforcing policy, the `block tvdi` will act as the PEP and prevent further execution if the hash values do not match – for instance, by pausing the affected VM. Alternatively, a reporting only policy could be applied allowing the operation to proceed but reporting the incident. If the guest OS needs to be informed about an event, modification to the guest OS's disk driver or the virtual support application have to be implemented. Since the metafile is sealed to a specific set of host platform integrity metrics the following assumption is not unreasonable: the metafile would only be unsealed if the host was running a process to enforce the above policies.

A `trust control` process is required to take charge of reporting and assuring a third party that the TVDI implementation is correct and trustworthy. It will do so by measuring the integrity of each component of the TVDI system. The integrity values are stored in the platform's PCR registers. Attestation of their values can be made to a remote entity on request. The `trust control` also acts as the PDP of the TVDI and is further responsible for migrating the metafile and moving chunks to a different host in case a VM is moved.

## 6.4 Trusted Virtual Disk Images

---

### 6.4.4.3 Recovery From Integrity Failure

If an integrity check fails to validate, correct operation cannot be guaranteed. For instance, arbitrary changes to program code may render the VM un-operational. To mitigate this issue a VM may be set up to automatically recover from an integrity failure. This can be accomplished by returning to an earlier, trusted snapshot. A similar procedure to recover from integrity failures during bootstrapping has been proposed by Arbaugh et al. [16]. Unlike their proposition, we do not require a trusted repository as security is provided by the TVDI itself. Any storage repository holding copies of the last functional snapshot is sufficient. A snapshot repository has to be set up in advance and `block tvdi` has to be provided with its location. In the case of an integrity failure, `block tvdi` could then move the faulty chunks and obtain alternative chunks from the repository. To mitigate flooding or denial of service attacks affecting other VMs on the same host, defective chunks may be moved to a quarantine directory and the number of recovery attempts may be limited to a fixed amount per time span. Afterwards, `block tvdi` may proceed with one of the policies described in the following section.

### 6.4.5 Policy Model

Depending on the implementation, different policies to handle integrity failures are possible:

**Enforcing policy** - If an integrity check fails the `block tvdi` will deny further access to the TVDI. The `trust control` program in Dom 0 can pause the affected machine and report the incident.

**Reporting only policy** - The VM will be allowed further access to the TVDI, yet the incident will be reported. If the guest OS itself needs to be notified, additional modifications to the front-end driver are required.

### 6.4.6 Software Enforcement

To ensure a specific hypervisor and management domain is loaded we rely on the chain of trust as described previously. This allows the trusted third party to gain

## 6.4 Trusted Virtual Disk Images

---

confidence in the execution state of a platform via remote attestation. We will further discuss in Chapter 7 how trustworthy security services can be bootstrapped and trusted components can be protected.

### 6.4.7 Metafile

The metafile is central to our TVDI design. It holds the `EncryptionKey` and integrity metrics for each chunk, and needs special protection mechanisms. Therefore, we require the metafile to be confidential and tamper-evident. The metafile's confidentiality is ensured by protecting its content via the TPM's sealing mechanism when a corresponding VM is suspended or powered down. Tamper-evidence is guaranteed by checking the metafile's integrity against a pre-stored value in the metafile's header section. Moreover, the metafile is bundled together with the data chunks and thus both may reside on untrusted storage. The metafile should, however, be protected in such a manner that its content can only be revealed if the host system is in a pre-defined state. We therefore employ the DRTM or, alternatively, the SRTM as discussed in Section 3.2.6 and Section 3.2.7. As a consequence the metafile has to be maintained by the same entity which manages the TPM. This is currently Dom 0 but as discussed in Section 4.2.3 and Section 7.4.6, Dom 0 might be disaggregated and the TPM management might be handled by a different entity. Before a VM is powered up, the TVDI's metafile is loaded and checked as follows:

1. `block tvdi` calls `read()` function in *libaio* to read the metafile data from persistent storage.
2. `block tvdi` calls `unseal()` function in *libTPM* to unseal the metafile under the corresponding CMK and a set of PCR values. As explained in Section 6.5.1, the CMK is created during the initialisation phase.
3. `block tvdi` calls `hash()` function in *libcrypto* to compute the integrity of the metafile. The metafile itself carries its own integrity metrics in the header section. The value in the metafile's header is replaced with a fixed value, before the `hash()` function is called to ensure the value itself is not represented in the result. Afterwards, the result is compared against the integrity metrics taken from the metafile's header. Once the metafile is loaded and checked it is held in a protected part of memory and receives updates throughout the Dom U's lifetime.

## 6.4 Trusted Virtual Disk Images

---

Eventually, if a Dom U is powered down or suspended, `block tvdi` decommis-sions the metafile as described in the following:

1. `block tvdi` unmaps the Dom U's block device.
2. `block tvdi` substitutes the metafile's integrity metrics with a fixed value and calls the `hash()` function in *libcrypto* to compute the integrity metrics of the metafile. Finally, the result of this is placed in the metafile's header as its integrity metrics.
3. `block tvdi` calls the `seal()` function in *libTPM* to seal the metafile.
4. `block tvdi` calls the `write()` function in *libaio* to write the sealed metafile data to persistent storage.

Header property	Description
<code>NextFreeChunk</code>	Random number, next available chunk number
<code>EncryptionMethod</code>	Encryption method for all chunks
<code>EncryptionKey</code>	Symmetric encryption key for all chunks
<code>DigestMethod</code>	Hash algorithm of the metafile
<code>DigestValue</code>	Hash value of the metafile
<code>TimeStamp</code>	Timestamp value of the metafile
<code>SnapshotVersion</code>	Snapshot association
<code>ChunkSize</code>	Fixed value of the max size of a chunk
<code>ImageSize</code>	Fixed value of the max size of the TVDI
Chunk property	Description
<code>BlockAddress</code>	Mapping of the chunk
<code>ChunkPath</code>	Storage location of the chunk
<code>SnapshotVersion</code>	Snapshot status of the chunk
<code>DigestMethod</code>	Hash algorithm of the chunk
<code>DigestValue</code>	Hash value of the chunk

Table 6.2: Metafile properties.

### 6.4.7.1 Metafile Structure

In this section we will describe the metafile structure in detail. We decided against the obvious approach of mapping block addresses to chunk file names, in order to implement a snapshot capability (see Section 6.5.6). We propose to implement an unique random number (`NextFreeChunk`) as an addition to the chunk's basename

## 6.4 Trusted Virtual Disk Images

---

(for example, `chunk.[i]`). By doing so, no information about a chunk's source or allocation is revealed. As a result, the metafile reflects the block address mapping via the `BlockAddress` directive in the according section. Table 6.2 outlines the metafile properties and listing 6.1 illustrates a sample metafile.

We refer to the chunk's size via the `ChunkSize` directive in the header section of the metafile. A small chunk size would result in a constant, but avoidable, rehash if information within the chunk had changed. A large chunk size, on the other hand, would result in an increased execution time as the complete chunk needs to be hashed during runtime. To reduce additional read/write overhead and the amount of hash operations, we suggest using a fixed chunk size equal to the cache size of the underlying hard drive or file system. By doing so, the performance impact of multiple chunk read/write operations – for instance caused by fragmentation – can be further reduced.

```
<sampleImage>
<header>
...
<DigestMethod>SHA256</DigestMethod>
<DigestValue>894f435gd...fas32dag</DigestValue>
<EncryptionKey>3b23894f...fce3bc95</EncryptionKey>
<EncryptionMethod>AES</EncryptionMethod>
<ChunkSize>16777216</ChunkSize>
<ImageSize>536870912000</ImageSize>
<NextFreeChunk>123</NextFreeChunk>
<SnapshotVersion>2</SnapshotVersion>
<TimeStamp>68161</TimeStamp>
</header>
...
<chunk.122>
<SnapshotVersion>2</SnapshotVersion>
<BlockAddress>00040000</BlockAddress>
<ChunkPath>/sampleImage/chunk.122</ChunkPath>
<DigestMethod>SHA256</DigestMethod>
<DigestValue>dc460da4ad72c...6899d54ef98b5</DigestValue>
...
</chunk.122>
</sampleImage>
```

Listing 6.1: Sample metafile.

## 6.5 Life Cycle

Aside from the security goals described in Section 6.3.2, a worthwhile operation requires the TVDI to fulfil an additional set of features. In this section we discuss the typical life-cycle of a TVDI.

### 6.5.1 Initialisation

Creating a new TVDI only requires the initialisation of one metafile per disk image instance. Disk images can grow dynamically during their lifetime due to their sparse nature as will be discussed in 6.5.4. The sparse format allows the image to utilise disk space more efficiently by only saving allocated disk space and storing unallocated disk space in an abbreviated way. However, it is still necessary to specify the disk image's maximum capacity during initialisation as it is treated by the VM as a hard drive with fixed physical layout. Other parameters, such as encryption or hash algorithms, may be specified during generation or set to a default value. During the initialisation procedure a random `EncryptionKey` is generated and stored in the metafile. Depending on the usage scenario, the initialisation procedure itself can take place on the host, by a trusted third party, or by an IT department. Initialisation is always triggered by the image owner, however.

Initially, the metafile is sealed to the CMK created during initialisation as described in [261]. It is then migrated – by a Migration Authority (MA) – to its destination host. The destination host is subsequently able to use the metafile and proceed with normal operation as discussed in Section 6.4.4. Each chunk will be created dynamically by the `block tvdi` when a VM writes to the corresponding area of the image.

### 6.5.2 Deletion

Our main design goals stated in Section 6.3.2 included the maintenance of confidentiality and integrity throughout the disk image's life-cycle. Those characteristics should still be intact once the TVDI reaches the end of its life. The metafile is the digital equivalent to a key and thus needs to be kept secure throughout the TVDI's life-cycle. If the metafile and the `EncryptionKey` are securely deleted then this ef-

## 6.5 Life Cycle

---

fectively deletes the image. This design requires careful control over the distribution of the metafile and `EncryptionKey`. Ensuring that the metafile can only be unsealed on a particular platform gives some degree of control of the file's distribution.

### 6.5.3 Backup

An image is bound to a specific platform during operation. Recovering data from this platform is difficult if it fails while the image is bound to it. A trusted third party or the initial TVDI owner can, however, keep a backup of the cleartext metafile or of the `EncryptionKey` from the metafile. The metafile should never be stored in the clear, hence the TVDI owner should sufficiently protect any metafile copy – for instance, by sealing it to its own TPM. However, if a host fails during operation and the metafile needs to be recovered, integrity metrics may not be up-to-date. Transferring a metafile backup to a new host takes place as outlined in Section 6.5.5.

### 6.5.4 Sparse Format

TVDI utilises disk space more efficiently by saving disk content in a sparse manner. Instead of writing out allocated but empty disk space, TVDIs only store abbreviated information about those areas. Primarily it allows data to be saved more efficiently whilst at the same time allowing images to grow during operation.

Due to the encryption and sparse nature of the TVDI implementation, releasing free space is a challenging task. Once an address block is allocated and eventually released by a guest OS – for example via file deletion – the underlying TVDI implementation cannot reclaim that free space automatically. The guest OS does not actually delete free blocks, but instead removes the according entries in the file system's allocation table. Without modifying the overlying guest OS, garbage collection is far from trivial. However, this is an issue many VDIs face and a mechanism to reclaim allocated but unused storage becomes necessary. Our current approach to reclaim free space anticipates an offline garbage collection mechanism. Thus, an image is scanned by the `block tvdi` for free space while the VM is powered off and an exclusive access right can be guaranteed. This way no guest OS modifications are necessary and no additional performance overhead is inflicted while the VM is running.

## 6.5 Life Cycle

---

### 6.5.5 Migration

The TVDI we presented earlier allows an easy and secure offline migration mechanism – a migration while the VM is powered down.

One of the assumptions in Section 6.4.1 requires the hosting machine to be up and in a trustworthy state. This allows us to migrate the metafile’s protection key (CMK) to a different TPM. We utilise a trusted control (`trust ctrl`) process in Figure 6.2, to interact with the trusted third party (Migration Authority) and the back-end (`block tvdi`), implementation. The Migration Authority manages the migration process and may temporarily host the CMK. Once the TVDI files are accessible by the targeted machine, the CMK is migrated to the target TPM. In detail, the migration process of the metafile takes place as follows:

1. `trust ctrl` initiates a shutdown of the VM which is set to be migrated.
2. `block tvdi` finalises the integrity creation as discussed in Section 6.4.4 and finally seals the metafile as highlighted in Section 6.4.7.

In order to avoid attacks on the migration process, the `trust ctrl` application will only allow migration if the VM is not running.

3. `trust ctrl` temporarily migrates the CMK to the MA as described in [261]. The MA then decides which target machine the key will be migrated to. During this phase the target machine attests its state to the MA. The MA will then decide whether the remote state is trustworthy and safe to migrate the metafile to.
4. Once the CMK is migrated onto the target platform’s TPM, the destination `trust ctrl` application can unwrap the metafile.

Hence the disk image can be decrypted, checked and, finally, the target VM can be powered up. In a suspended state, memory content on the source machine could also be encrypted using the `EncryptionKey` provided in the metafile and migrated afterwards.

### 6.5.6 Snapshots

We believe that providing a snapshot feature is a very valuable and desirable property in the context of virtualisation, even though it results in more fragmentation:

## 6.6 Security Analysis

---

using a fixed chunk size, a snapshot will create fragments and waste a fixed amount of storage space with each chunk. However, we decided to implement block address mapping into the metafile in order to allow a snapshot feature. In a snapshot-less scenario a typical modification to a chunk would take place as outlined in Section 6.4.4: first read, update and hash the chunk, then write it to persistent storage. A snapshot however, is created as follows:

1. `block tvdi` treats every chunk as read-only after a snapshot request is made.
2. `block tvdi` increases the `SnapshotVersion` in the metafile's header section.
3. `block tvdi` allocates a new sequence number for every new chunk, based on `NextFreeChunk`.

A write operation to an already existing chunk will cause the creation of a new chunk file.

4. `block tvdi` writes the modified data to a new chunk.  
In a copy-on-write manner, a write operation causes `block tvdi` to read the chunk, perform the requested write and calculate the new integrity metrics.
5. `block tvdi` updates the `SnapshotVersion` in the metafile's chunk section.

The metafile keeps a record of the snapshot status via the `SnapshotVersion` directive in the header section. In addition, the metafile and the chunks currently in operation are written to the storage location in order to ensure a crash consistent state. This allows snapshots to be also taken whilst the TVDI is in use. Chunks which do not contain a `SnapshotVersion` have not been written to and therefore are valid for all snapshot revisions.

## 6.6 Security Analysis

In the foregoing sections we described how we achieved our initial security goals to provide data secrecy and integrity for TVDIs with regards to the security requirements discussed in Section 6.3.1. Our data confidentiality goal is met by ensuring that no data parts of a TVDI leave the trusted platform unencrypted and by protecting the metafile with the TC sealing mechanism. Our data integrity goal is met by maintaining trusted records of the integrity metrics of all data parts of a TVDI

## 6.6 Security Analysis

---

and by protecting those records with an additional integrity record of the metafile. The enforcement of a specific hypervisor is guaranteed by the TC's integrity and measurement architecture. The hypervisor is only able to unseal the metafile and trusted records if the configuration requirements are met. In the following sections we will demonstrate, how TVDIs will perform in response to the subsequently described attacks.

### Threat Model

In our threat model we assume an adversary who has access to the storage location of the TVDI. This attacker may read, modify, and inject arbitrary information into the TVDI and has control over the communication network. An attacker may also capture a genuine image and replay it at a later stage. We also assume that the trusted third party is considered trustworthy.

### Confidentiality Attack

The `EncryptionKey` for the TVDI is held within the metafile. Thus the confidentiality of the metafile is paramount. The metafile itself is protected by the TC sealing mechanism. Thus, to break confidentiality, an attacker would have to compromise the trusted environment or gain access to the TPM's private SRK or private CMK. The private part of the SRK never leaves the TPM, where the CMK might be stored outside the TPM, but protected by public-key encryption under the SRK. Hence an attacker must either break the public-key encryption which protects the CMK, or must be able to extract information from the TPM. While attacks on the TPM are technically possible [215], we consider them impractical. For backup or migration purposes, a copy of the metafile may however, be stored by the trusted third party or the TVDI owner. Here, the metafile should be again sealed by a TPM or protected accordingly. Consequently, if an attacker cannot recover the private part of the SRK or CMK, an attacker must be able to break the public-key algorithm. If the metafile cannot be recovered outside the trusted environment, confidentiality remains intact.

## 6.6 Security Analysis

---

### Integrity Attack

Legitimate updates of integrity metrics in the metafile can only be made if the system has access to the metafile as outlined earlier. Any unauthorised modification to chunks (step 8 in Section 6.4.4.2), or to metadata (step 3 in Section 6.4.7) will be uncovered by a mismatched checksum. Rolling back to a previous version of a chunk will have the right `EncryptionKey` but the wrong integrity metrics and can be detected. Additionally, swapping the order of the chunks – without swapping the integrity metrics in the metafile – will be identified by inconsistent checksums. Modifying integrity metrics in the metafile itself will be detected by an incorrect checksum of the metafile itself. Hence, the attacker must be able to successfully attack the hash algorithm by finding a useful collision. We consider that attacks on the hash algorithm itself will be unfeasible in the foreseeable future. An attacker must, therefore, successfully manipulate the cleartext metafile. However, as long as the trusted components remain intact, an attacker will fail to inject malicious integrity metrics into the metafile.

### Man-in-the-middle Attack

A man-in-the-middle attack is ineffective because no unencrypted data ever leaves the host, the VM or the trusted third party. An adversary may, however, present a malicious metafile to the host. In this case, the host will be unable to unseal the metafile as the host expects the metafile to be sealed to its own TPM. The TVDI implementation will therefore reject all non-valid metafiles.

### Replay Attack

We incorporate TPM-based timestamps into our metafile to prevent replay attacks. A metafile can either include a timestamp supplied by the TPM – via `TPM_GetTicks` to guarantee freshness – or, on initialisation of the metafile, instantiate a new TPM monotonic counter. Alternatively, the metafile’s integrity metrics could be signed via `TPM_TickStampBlock`. Together with the TPM’s internal tick counter, this attests to the time at which the metafile was unsealed by the TPM.

## 6.7 Considerations

---

### Availability Attack

An adversary who has control over the storage location or network may change, append data, or delete chunks and/or metadata. Alterations will be easily detected but modifications may render the VM un-operational. Also, an attacker may fake latency to disrupt or defer a normal operation. Such denial of service attacks are beyond the scope of our stated security goals. A more enhanced version of TVDI may make redundant chunks available to realise multiplexed storage – similar to a RAID array. This could mitigate availability attacks at the cost of inflicting additional write overhead.

## 6.7 Considerations

While our design offers many beneficial security attributes it raises different concerns that need to be addressed for every day usage. This section discusses several usage scenarios which require special consideration.

### 6.7.1 Fragmentation

Fragmentation poses a performance and storage space issue. TVDIs provide a hard drive implementation and therefore file handling is managed by the overlying file system structure. Different file systems implement various mechanisms to prevent fragmentation, however, the file system is not aware of how the disk implementation is handled by the TVDI. Any file could be spread across multiple chunks and invoke multiple hash, encryption and decryption operations if it is altered. Thus, too many fragments located throughout the disk-layout could inflict a performance overhead. The use of file systems which group blocks – such as UFS (Unix File System), or FFS (Fast File System) – promise to mitigate the fragmentation issue in this case. Snapshots, on the other hand, will create fragmentation since new chunks with redundant information are created for every snapshot version.

## 6.8 Usage Scenario

---

### 6.7.2 Performance

TVDI has been designed with security, rather than performance, in mind. Hash operations certainly will have an impact on performance. It will result in slightly higher disk I/O operations and in an increased memory and CPU usage – due to cryptographic operations. Nevertheless, by spreading chunks across multiple storage locations a RAID array-like feature could be achieved without inflicting additional computational overhead. Depending on the requirements of an application, the security gained could be considered more important than the performance impact.

### 6.7.3 Swap-space

Files that might change frequently, such as swap-space or temporary files, would cause a constant rehashing even though their integrity might not be of importance. To reduce the performance impact, those files could be out-sourced into separate, encryption-only TVDIs. For swap-space, this could be done by creating an encrypted disk image with a random encryption key each time a VM is powered up. Alternatively, encryption keys for swap-space could be included in the metafile.

## 6.8 Usage Scenario

The TVDI design presented in this chapter is best suited for VMs which are in need of transparent and trusted storage. In a virtualised infrastructure, TVDIs provide a high degree of control for the image owner, for instance the IT department. On one hand, TVDIs can be used to increase the confidence in locally stored images. On the other hand IT services may provide remote executable TVDIs to clients using untrusted existing storage infrastructure like standard network shares or storage. Moreover, for teleworkers those TVDIs might be made available via an untrusted network link, such as the internet.

The TVDI owner can gain confidence in the platform's status by using attestation and can control the execution of the image by employing the trust control process. Moreover, through the sealing process the owner can restrict the distribution and usage of a particular TVDI.

## 6.9 Discussion

---

Hence, applications which benefit most from TVDIs are special corporate management domains or thin appliances with the need for higher assurances.

## 6.9 Discussion

With the TVDI implementation presented in this chapter, we are able to deliver our primary goal of confidentiality and our secondary goal of integrity protection, transparently to VDIs. We are also able to ensure a TVDI is processed by a specific hypervisor and management VM, in order to meet our tertiary design goal. We therefore based our design on the foundation of TC and transparently incorporated virtualisation technology. Further, we are able to provide those security attributes while the underlying storage subsystem cannot guarantee any of these. Thus, data on a TVDI is protected as a whole, embracing log files and applications, as well as any part of the OS itself – including legacy OSs. Moreover, potential compromises of the storage subsystem do not effect the security of co-resident VMs. As a consequence the storage subsystem is removed from the TCB. If the TVDI implementation is located in a separate VM and decoupled from Dom 0, the security assurance of the platform is further increased.

Similar endeavours towards providing secure storage in a hostile environment, carried out by Suzaki et al. on OS circular, further underlines how important confidentiality and integrity are for virtual storage. OS circular is a framework for internet-based VDI distribution which Suzaki et al. demonstrated in 2007 [255]. Their approach was based on a one-to-many scheme and thus lacks support for any disk image updates made by the client. The client checks data integrity by means of a stackable virtual disk driver, based on an implementation of a trusted HTTP-FUSE CLOOP driver [255]. This drastically limits the OS support.

Terra [86] is a trusted hypervisor which partitions commodity platforms into high assurance VMs. The concept of attesting to integrity for a VDI has already been mentioned by the authors but has not been further elaborated in this paper. We believe, by aggregating write operations into larger groups, we can reduce the overhead of cryptographic operations and thus increase performance. More importantly, our design allows a flexible integration of a snapshot capability.

## 6.10 Summary

---

As stressed by Oberheide et al. [194], live migration of VMs poses a threat to integrity and confidentiality. Oberheide demonstrated how a migrating VM can be manipulated whilst traveling across an unsecured network. Live migration in general poses a difficult design challenge: the system's state and its current memory content has to be transferred to a different host in a fast and transparent, yet trusted, manner. Although TVDI addresses offline migration, it currently does not address the issue of live migration.

## 6.10 Summary

In this chapter we described the motivation, goals and principles behind our TVDI design. This design is based on the discussion of the security requirements and challenges for disk images in a virtual environment. Our TVDI concept is able to provide integrity and confidentiality to co-resident VMs in order to satisfy the isolation requirement (4.1.2.1) identified previously. Consequently, the storage location is removed from the TCB, which reduces overall complexity (requirement 4.1.2.4) and further increases isolation (requirement 4.1.2.1). To improve on the attestation requirement (4.1.2.2), we bound each metafile to a particular disk image and TPM. This binding allows a local or remote party to effectively gain confidence in the disk image as well as the virtual environment used by a particular VM. By allowing the disk image owner to control sealing and migration, we enable the owner to retain control and enforce policies on the disk image throughout its life-cycle (requirement 4.1.2.3). Additionally, our TVDI design can be transparently incorporated into existing virtual infrastructures to satisfy the compatibility requirement (4.1.2.5).

A detailed discussion of our design implementation, based on the XEN para-virtualisation model was followed by a discussion of the typical life-cycle of a TVDI. Subsequently, we provided a security analysis based on our threat model. Our TVDI currently requires compromises in certain areas. This does not limit the application scenarios but it does have an impact on usage considerations. This is an inherited problem as most VDIs which employ cryptography will suffer from limitations, such as the garbage collection issue. Nevertheless, our TVDI design is able to deliver confidentiality and integrity protection transparently to VMs, while allowing the TVDI owner to retain control over the image.

## Chapter 7

# LaLa: A Late Launch Application

### Contents

---

<b>7.1</b>	<b>Introduction</b>	<b>150</b>
<b>7.2</b>	<b>Motivation</b>	<b>151</b>
7.2.1	Design Goals	152
<b>7.3</b>	<b>Background</b>	<b>153</b>
7.3.1	Trusted Execution Technology	153
7.3.2	Launch Control Policies	156
7.3.3	OpenVZ	157
<b>7.4</b>	<b>Implementation Details</b>	<b>158</b>
7.4.1	Requirements and Limitations	160
7.4.2	Hardware Platform	160
7.4.3	Instant-on OS	161
7.4.4	The Late Launch Process	162
7.4.5	OVZ Virtual Machine	163
7.4.6	Trusted Management Domain	165
7.4.7	Environment Teardown	167
<b>7.5</b>	<b>Security Analysis</b>	<b>168</b>
<b>7.6</b>	<b>Discussion</b>	<b>172</b>
<b>7.7</b>	<b>Summary</b>	<b>174</b>

---

## 7.1 Introduction

---

*A computer lets you make more mistakes faster than any other human invention in human history . . . with the possible exception of handguns and tequila.*

– Mitch Ratcliffe

## 7.1 Introduction

In Chapter 5 we addressed the security guarantees of the hypervisor layer and in Chapter 6 investigated improvements to the storage layer on top of the hypervisor layer. In this chapter we build on the previous findings and investigate how the isolation of co-resident OSs can be improved with minimal impact on user experiences.

A traditional platform start-up procedure includes many components, such as the BIOS boot procedure, the boot loader and finally an OS. Modern OSs are becoming increasingly complex in functionality and size. This impacts not only the OS’s TCB and complexity but also the OS’s loading time; hence the start-up of a current OS can easily consume many minutes. This is undesirable for users who only want to perform simple or short operations such as browsing the web or checking emails. Additionally, the extra energy consumed by the unused functionality is wasted. This is certainly undesirable for stationary machines but even less desirable for mobile devices – the energy consumed by powering up a full-feature OS to perform just a simple task can reduce battery lifetime significantly.

Solutions, such as Splashtop Linux [62], Xandros Presto [291] or Hyperspace [206], which reduce complexity and start-up time already exist. They allow the user to boot quickly into a customised Linux OS with a reduced function set. Generally, such solutions are classified as instant-on OSs. However, most instant-on OSs force the user to make a decision of which OS to use. With our Late Launch (LaLa) prototype we enhance the user’s ability to use an instant-on OS, while powering up a full-featured OS, such as Windows, in the background. Our contribution is to combine the instant-on approach with our previously discussed efforts to improve the trusted virtual infrastructure. We therefore incorporate the improved hypervisor discussed in Chapter 5 and the trusted storage outlined in Chapter 6. Moreover,

## 7.2 Motivation

---

we make extensive use of newly available hardware protection capabilities, such as Intel's TXT extensions, to provide the user with trusted components.

The prototype described in Section 7.4 allows the use of an instant-on application (or appliance) to perform basic tasks such as web browsing, instant messaging or the use of custom made applications. The user can continue in this narrow mode of operation or start a fully functional OS in the background. Through this process, the user is still able to use the instant-on application initially presented to him even after the full OS has been launched. Our prototype enables the platform user to perform simple day-to-day tasks in a fast, comfortable and secure manner. We are able to ensure, through the new TXT features, that only hypervisors and VMs that comply with specific policies are executed. Thus our prototype combines the instant-on functionality with a trusted virtual infrastructure in a way which is transparent to the user. Additionally, we are able to preserve energy and extend battery life of mobile devices whilst, on the other hand, we are able to hide the long loading times of a modern OS without compromising the users' experiences.

At this early stage of development, we decided to exclude the instant-on system from the hardware protection boundaries in order to reduce the complexity of the implementation and to reduce overall loading times. The LaLa prototype was not designed to be a virtualisation silver bullet but to balance between security, manageability and user experience, based on existing technologies. However, as outlined in Section 7.4.3 in addition to the DRTM, the SRTM mechanism can be applied to the instant-on OS to include it in the trust boundaries.

## 7.2 Motivation

This work is mainly motivated by the desire to provide more trustworthy security services to end-users by incorporating virtualisation and state of the art trust technology with regard to the requirements outlined in Chapter 4. We discussed previously ways to improve the trustworthiness of the hypervisor and the trusted virtual storage in Chapter 5 and Chapter 6 respectively. The work described in this chapter builds upon those layers to improve the overall trustworthiness of the virtual infrastructure. We aim to further strengthen the guarantees of the virtual infrastructure as well as improving user experiences.

## 7.2 Motivation

---

The objective of this work is to separate legacy from trustworthy (attestable) applications in a manner which does not compromise user experiences. We also want to offer an alternative to hardware sleep and resume as a way of avoiding long OS boot times and improving user experiences. More importantly this allows us to avoid cold-boot attacks [106]. As hibernating or sleeping machines may contain key material in memory and/or in persistent storage, those attacks can be prevented by using an instant-on solution rather than a sleep and resume approach. We want to provide the user with simple interfaces for their daily tasks that run outside the scope of any protected environments instead of encrypting and decrypting the memory content of sleeping machines. Additionally our approach allows us to reduce the complexity and increase the isolation between untrusted and trusted applications.

We want to provide a trusted management framework through early hardware assisted policy enforcement as well as policy enforcement throughout the platform's life-cycle. To achieve this, we securely bootstrap security services, such as a hypervisor capable of enforcing policies, and a trusted management domain as an entry point for sophisticated policy decisions.

This work is also motivated by the desire to enhance the user's experience by allowing them to interact with their system in a productive fashion as early as possible from power-on. As part of this desire, we want to support more flexible client-device use models where a user may choose to operate within a restricted environment. A user might initially operate in a thin-client desktop whilst retaining the ability to flexibly and dynamically make a transition to thicker-client functionality when needed, in a way that can be verified and attested to.

### 7.2.1 Design Goals

To address the above requirements, our main security goals are:

- Provide more trustworthy security services (trusted compartments).
- Separate legacy from trusted components.
- Provide strong hardware-based policy enforcement.
- Utilise recent advances in hardware-based trust technology to increase the isolation guarantees.

## 7.3 Background

---

Additionally, we want to reduce long OS loading times to enhance user experiences and preserve energy on mobile platforms.

Platforms which benefit most from our prototype are mobile platforms with limited energy capacity and the requirement for trusted manageability. The scenarios we have in mind include some where an on-the-go worker requires timely access to their productivity tools whilst, simultaneously, allowing an enterprise to gain confidence in a user's platform before allowing them access to corporate assets. On the one hand, a user is able to perform his everyday tasks in an energy efficient way, whilst being isolated from the rest of the system. On the other hand, corporate policies can be maintained and updated by the trusted security services.

## 7.3 Background

The following sections describe the existing tools and technologies in detail, which form the technological basis for our Late Launch prototype described in Section 7.4. The subsequent sections are a more detailed explanation of the background discussion in Section 3.2.7. A more broader explanation about the Late Launch concept and Intel's TXT in particular can be found in [103].

### 7.3.1 Trusted Execution Technology

Trusted Execution Technology (TXT) [103], formerly named LaGrande, is a set of hardware extensions introduced by Intel to defend against software attacks. Intel's extensions are good examples of how TC and virtualisation are merging – hardware virtualisation support in the CPU and chipset are extended by a TPM to provide trusted and sealed storage, as well as supporting platform attestation. The TXT platform complements the static RTM by allowing a dynamic creation of protected components via a dynamic RTM. Rather than measuring components in a traditional consecutive chain, TXT allows the platform to establish a measured environment at any given time. This *Late Launch* can occur at any time during runtime and thus puts the root of trust dynamically into the platform's hardware. Finally, after the launch procedure, a *Measured Launch Environment* (MLE) is established. The MLE can act as the trusted base for an OS kernel or hypervisor. The MLE can be any type of code with arbitrary functionality, however, for the remainder of this

### 7.3 Background

thesis we assume the MLE implements the hypervisor functionality. Considered in a broader context, the MLE is the piece of software which utilises Intel’s Safer Mode eXtension (SMX) [121]. The SMX represents the extension to the CPU in the form of new CPU instructions (GETSEC) and the authenticated code execution. In order to guarantee a safe environment, the MLE relies heavily on the underlying hardware support of the SMX, trusted initialisation and policy enforcement as described in the subsequent paragraphs.

A Late Launch occurs in different phases. Firstly, the MLE code and a special authentication module (SINIT Authenticated Code module) are loaded into unprotected memory regions. This is illustrated by step (1) in Figure 7.1. The SINIT AC Module (ACM) is a platform and processor specific module responsible for the initialisation of the platform, the measurement of the environment, and the policy enforcement. The ACM is cryptographically signed by Intel and must be verified using a pre-supplied public verification key contained in the ACM. A hash of this public key is securely stored in the platform’s chipset and used to verify the integrity of the public key.

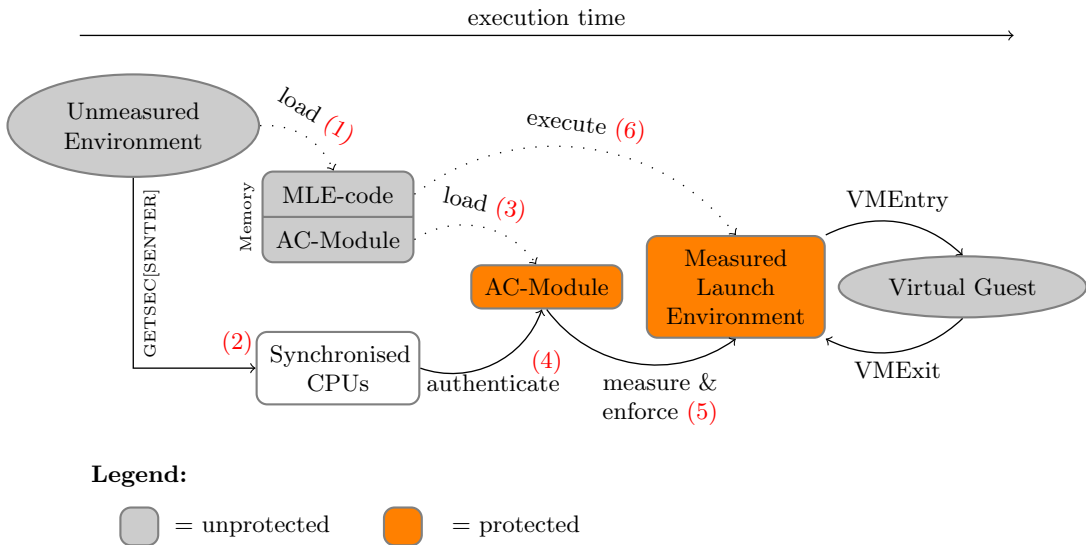


Figure 7.1: TXT initialisation phase.

### 7.3 Background

---

Secondly, a new CPU instruction – GETSEC[SENDER] – is invoked to make the transition to the MLE. The GETSEC[SENDER] is provided by the new SMX extensions and initialises the hardware with the goal of bringing the chipset and processor into a known, as well as protected, state. For instance, interrupts, DMA access and debugging capabilities are disabled during this phase – see [103] for a complete description of the protection mechanisms. The GETSEC[SENDER] instruction causes all but the Initiating Logical Processor (ILP) to enter a waitstate (Step (2) in Figure 7.1.). All of the remaining operations are subsequently carried out by the ILP.

The ACM is then loaded into a special protected execution area within the ILP, and its signature is verified using the public verification key described above (step (3) and (4) in Figure 7.1). This methodology is used by Intel to verify the integrity of the public key provided in the header of the ACM. If it is successfully verified, the ACM is executed and finalises the platform initialisation as well as protecting the MLE memory regions. Next, the ACM measures and verifies the integrity of the MLE in memory (step (5)), and if the integrity metric of the MLE conforms to the platform policies described in Section 7.3.2, it launches the MLE (step (6)). Once control is passed on to the MLE, platform configuration is finalised by waking up the remaining processors as well as re-enabling interrupts.

In a virtualised environment, the MLE acts as a hypervisor and is consequently responsible for managing every VMExit and VMEntry operation in order to maintain the measured environment [121]. Eventually, to tear down an MLE, a GETSEC[SEXIT] command is issued. It essentially operates in reverse to the GETSEC[SENDER] command: on a multi-core platform, all processors are synchronised. Remaining secrets, such as encryption keys which might have been used during the execution of the MLE, are encrypted. Finally, the memory is scrubbed to prevent any leakage. After the GETSEC[SEXIT], the platform continues with normal operation. The MLE might pass platform control to system software previously executed in a VM.

In this way, the hardware extensions provide the special protected environment necessary for code to be executed and policy to be enforced. Together with the platform policy, which will be described in the following section, the system is able to determine whether the MLE meets a pre-defined and trusted state or not. For a more detailed description of the TXT extension refer to [103, 121].

## 7.3 Background

### 7.3.2 Launch Control Policies

Launch Control Policies (LCPs) [103] are the central control mechanism of the TXT security concept. The LCPs are used to deliver the Policy Decision Point where the ACM provides the Policy Enforcement Point. The LCPs together with the ACM ensure only authorised MLEs, which meet a set of pre-defined criteria, can be launched. LCPs are kept inside the TPM's NV memory to protect them from unauthorised access and modification, as well as to persist after power cycles. As the TPM NV storage is limited – typically 4096 bytes – the LCPs only contains one aggregated integrity metric. However, this could be the integrity metric of an externally stored list of acceptable policies. As shown in Figure 7.2, the policies are loaded either from the TPM (LCP\_POLICY) or from external storage (LCP\_POLICY\_DATA) and are then enforced by the ACM. If all requirements of the active policies are fulfilled the platform proceeds to launch the MLE. Modifying the LCPs requires the TPM owner to present his authentication credentials. The TXT extensions define two different sets of policies:

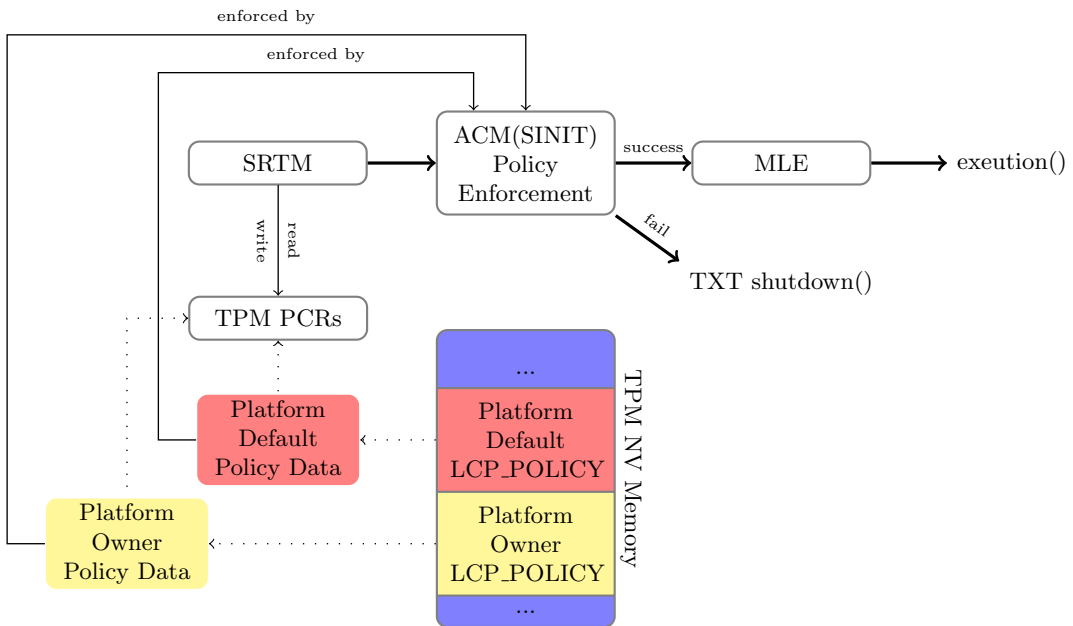


Figure 7.2: TXT policy flow, based on [121].

**Supplier or Policy Default (PD) policy** - The default policies are provided by the platform supplier and represent the standard policies. Nevertheless, the owner has the possibility to bypass the default policies.

## 7.3 Background

---

**Platform Owner (PO) policy** - It is vital to understand that the platform owner is different to the platform user. Here the TXT concept follows the TCG's design principles [264] and differentiates between platform owner and user. The platform owner can be the same entity as the user, but it could also be an IT department or the company itself. The policy decision and deployment is controlled by the platform owner.

If no policy has been defined the platform allows any MLE with any configuration to be launched. Both sets of policies, PD and PO, can contain a list of valid MLE measurements (currently SHA-1) as well as a list of valid platform configuration measurements. The pre-stored MLE measurements are compared against the measurements of the pending MLE (step (6) in Figure 7.1). The platform configuration value is then checked against a policy defined set of PCRs.

The policy enforcement model is a powerful tool for any platform supplier as it enables them to define their policies by default. The platform owner can also decide, by using the PO, whether the PD will be enforced or not. Moreover, keeping a pro-active list of trusted environments, and enforcing the execution with low-level system software offers, potentially, greater assurance compared to retrofitting policy enforcement.

### 7.3.3 OpenVZ

We will discuss OpenVZ and the requirement for migration in greater detail in Section 7.4.5 and so we will only provide the required background information in this section. The OpenVZ (OVZ) extensions [198] are able to fully encapsulate the running state of a VM, save it, and restore it on a different OVZ system. The state of a saved VM is also referred to as a snapshot. A snapshot can be transferred to, and resumed by, a different host in a process known as migration. In a live migration process, the snapshot is transparently moved to a remote host with minimal time between the interruption on the source host and resumption on the target host. This process is independent of the network state and the application state so it is not noticeable to users. The process does still inflict some downtime on the migration target but it is usually performed within tens of milliseconds. By default, an OVZ migration is carried out using the platform's networking facilities and divided into the following steps:

## 7.4 Implementation Details

---

1. OVZ synchronises the source and destination VM via the ‘rsync’ software application. This first-pass rsync performs an initial synchronisation of the source and destination VM.
2. Once synchronised, the source VM is frozen and a snapshot is created. This snapshot is then transferred via SSH Secure Copy (SCP) to the destination VM.
3. The third step performs a second-pass rsync in which the remaining files that have changed – between the first-pass rsync and the creation of the snapshot – are transferred.
4. Finally, the destination VM is resumed using the transferred snapshot.

As a basis for the live migration process, we decided to use OVZ over other migration solutions such as ZAP [200] or Vserver [61]. OVZ offers certain advantages over competing solutions, for instance more recent patches as well as a sophisticated snapshot and live migration functionality. OVZ differs from machine virtualisation technologies, such as XEN, as it implements OS-level virtualisation. VMs hosted by OVZ share the same kernel and address space and additionally are limited to Linux. On one hand, OS-level virtualisation offers less isolation and less legacy support but, on the other, it implements a fast and flexible migration functionality. We do not consider this to be a shortcoming as we do not rely on the OS-level virtualisation isolation. We rely on the isolation guarantees of the underlying hypervisor and the hardware in particular.

## 7.4 Implementation Details

In the previous sections we introduced the available technological building blocks. The following sections demonstrate how the available technology can be applied in order to realise an instant-on trusted virtual architecture. Moreover, we will discuss the design goals and concepts of our prototype in detail, before describing implementation specific details and limitations.

## 7.4 Implementation Details

---

The execution flow of our LaLa prototype is devised in three main steps:

1. Start measured hypervisor and Trusted Management Domain.

Initially we run an instant-on application with its supporting OS directly on bare-metal. We then allow transition to a state where the original instant-on supporting OS is replaced on the bare-metal hardware by a hypervisor. The transition to a hypervisor-based environment is performed via a transparent measured launch. This measured launch brings the system into a defined state according to the pre-defined policies. Our late launch module will be discussed in detail in Section 7.4.4 and highlighted by Figure 7.3. The Trusted Management Domain will be described in Section 7.4.6.

2. Migrate Instant-on system into special VM.

An important point to note is that we do not migrate the original instant-on OS itself to a VM. Instead, we snapshot and capture the state of the application running in the initial instant-on bare-metal OS. We migrate the instant-on applications, which are already abstracted from the physical hardware, not the full instant-on OS which is bound to the physical hardware. Section 7.4.5 describes the migration process which is outlined in Figure 7.4.

3. Launch full-feature OS.

Throughout the migration process, the user is able to make full use of the instant-on application initially presented to him. The start of the measured hypervisor, as well as the migration process, is transparent to the user. After the migration is completed, the user is able to switch between a VM containing the migrated instant-on application and a VM containing a full-featured OS. The final step is illustrated in Figure 7.5.

The work demonstrated in the following sections was part of the research internship I carried out at the Hewlett-Packard Labs in Bristol, United Kingdom. The findings of this work provided the basis for a publication [89], as well as a patent application [56]. The prototype, as well as the source code, remain with Hewlett-Packard Labs for further development.

## 7.4 Implementation Details

---

### 7.4.1 Requirements and Limitations

We have based our prototype on Intel’s TXT concept for three main reasons. Firstly, the platforms which were available to us are all equipped with TXT. Secondly, at the stage of implementation, Intel’s concepts seemed slightly more elaborated and more suited to our needs than, for instance, AMD’s Secure Virtual Machine. This is mainly based on the presence of the ACM, which removes the burden of initialising the platform as well as allowing policy enforcement. Both these tasks have to be developed separately for AMD platforms. Thirdly, documentation [121], alongside a proof-of-concept implementation named ‘tboot’ [128], are publicly available, which allowed a more timely development.

Intel’s and AMD’s concepts are comparable but not compatible. The concepts discussed in this chapter are by no means limited to Intel and our prototype could be ported to support different platforms. The hardware platform must, however, be able to support the DRTM concept in order to perform a Late Launch.

We tried to keep implementation work of the prototype to a minimum and re-use as much existing code as possible. Hence, the instant-on OS, the migration VM, as well as the hypervisor and management domain are all based on open-source and Linux. The existing software has been modified to fit our needs – this will be described in the following sections. Our implementation could also be realised with proprietary software.

To perform the measured launch, our kernel module is loosely based on Intel’s trusted boot prototype, tboot [128]. Currently it requires compiling against a 2.6.18 Linux kernel due to the implementation decisions we made.

### 7.4.2 Hardware Platform

Our initial prototype is based on the Hewlett-Packard 6930p Montevino / Calpella based notebook as well as the Hewlett-Packard DC7800 and DC7900 desktop platforms. Key features required are Intel Virtualization Technology for x86 (VT-x) [120], Intel Virtualization Technology for Directed I/O (VT-d) [1] and Intel Safer Mode Extensions (SMX) [121].

## 7.4 Implementation Details

---

### 7.4.3 Instant-on OS

During the first stage of powering on the platform an initial instant-on system is booted. The instant-on system consists of the instant-on OS as well as a set of demo applications. In our demo application we use Damn Small Linux (DSL) [57] as the basis for our instant-on OS. The kernel has been modified to include the OpenVZ patches as described in the following section. Any other Linux-based instant-on OS with a 2.6.18 kernel could also be used. We decided to use DSL for the following reasons:

- Very small in size and memory usage.
- Fast boot-up.
- Highly customisable.

We further customised DSL to fit our needs. Unnecessary functionality has been stripped and the system has been tightly tailored to the hardware being used to minimise software complexity and to reduce boot time. The functionality provided by the instant-on OS could be freely adjusted to any requirements necessary.

In this early stage of prototyping, we decided against measuring the integrity of the instant-on system. This decision allowed us to speed up the boot process itself as creating the measurements for the SRTM takes up a considerable amount of time. We regard the instant-on OS as the private VM which does not fall under policy enforcement. Consequently, we consider the instant-on OS as being untrusted at this stage. Depending on the use-case scenario, requirements may dictate the instant-on to be a trusted OS, for instance to host a corporate thin client or similar appliance. Future versions of LaLa, which go beyond the proof-of-concept stage, might allow us to additionally create an SRTM to include the instant-on OS in the trust chain. However, allowing the initial trusted application to persist or migrate after installing a hypervisor is a challenging task and beyond the scope of this thesis.

Nevertheless, the instant-on OS offers some security benefits as the system software is a minimised Linux OS and resides in a self-contained, read-only partition. Therefore, application crashes or compromises during runtime are limited to a single session, while the hardware isolation prevents any infiltration of co-resident VMs.

## 7.4 Implementation Details

Unfortunately, a useable instant-on system must still contain a Linux kernel, hardware drivers and a set of applications, which still constitute a large TCB. Nevertheless, the instant-on system could be a pre-installed virtual appliance, comprised of a pre-configured security kernel to provide stronger isolation and potentially offering more attestable properties.

### 7.4.4 The Late Launch Process

The DRTM, or Late Launch, allows the concurrent existence of legacy and trusted software without breaking the requirement to maintain compatibility. As outlined in Section 7.3.1, Late Launch can put a platform into a defined state without interfering with existing software already running on the platform. It is tightly coupled to the platform support for virtualisation and allows verifiable start-up of trusted software, such as a hypervisor. We use this functionality and extend our instant-on OS with a custom kernel module to exploit this feature. The initialisation and measurement of the platform is an atomic operation orchestrated by the ACM. Initially the instant-on OS is run on the bare-metal hardware, then the Late Launch is triggered to transition to a hypervisor-based environment. Step (1) in Figure 7.3 highlights the position of the Late Launch in the execution flow of our prototype. The Late Launch is triggered by our kernel module and processed in the stages described in Section 7.3.1:

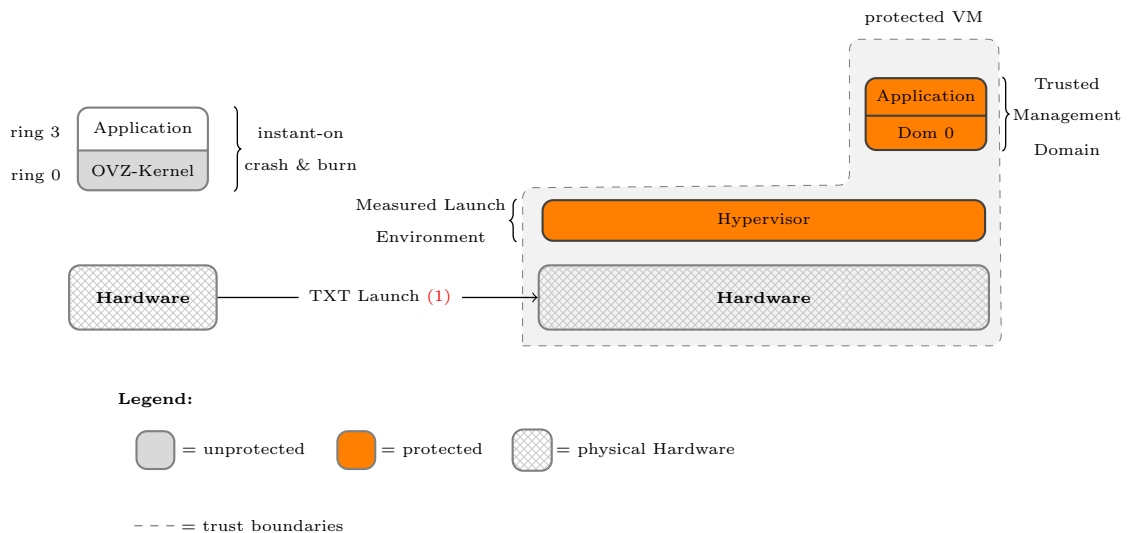


Figure 7.3: Step (1), measured launch.

## 7.4 Implementation Details

---

First of all the MLE is loaded into the system’s main memory. In our LaLa prototype the MLE is a version of the XEN hypervisor. Alongside the hypervisor code, the previously described ACM is required to verify and initialise the platform as has been described in Section 7.3.1. The ACM is cryptographically signed by Intel and created for each specific chipset individually. The ACM comes as a binary file and may be obtained from the tboot project website [128].

Once the MLE and ACM are placed into memory (step (1) in Figure 7.1), our modified kernel module executes the special CPU instruction (GETSEC[SENTER]), which triggers the Late Launch (step (2) in Figure 7.1). During the Late Launch procedure the ACM is loaded and authenticated (step (3) and (4) in Figure 7.1). Afterwards, the ACM measures and checks the integrity of the hypervisor in memory and, if it conforms to the platform policies, it is launched – step (5) and (6) in Figure 7.1. During normal execution the hypervisor itself and any potential secrets are protected, by the TXT hardware extension, from illegitimate access – for instance – unauthorised DMA. Finally, the hypervisor is responsible for dispatching every VM operation in order to maintain the measured environment.

### 7.4.5 OVZ Virtual Machine

The main goal of our design is to provide the migration of the instant-on system into a VM, as illustrated by step (2) in Figure 7.4. The migration of the instant-on OS is necessary to provide a smooth transition for the user and to overcome the fact that hardware interfaces will change due to the installation of a hypervisor. A hypervisor presents a different view of the existing hardware to the guest OS running on top of it. The different view on the hardware interfaces is outlined by the virtual HW in Figure 7.4. Handling the graphics hardware interface in particular is difficult and currently there is no hardware support available. In order to be able to efficiently share available resources and, more importantly, to enable the user to switch between VMs, the hardware view has to be synchronised. We do so by migrating the initial instant-on system into an untrusted VM, which has been specially created for this purpose. The illustration in Figure 7.4 shows the different view on physical and virtual hardware as well as illustrating, in step (2), the OVZ migration flow described in Section 7.3.3.

In addition to our main goals, we have two further objectives. Firstly, we want to be able to present the user with the same set of applications and interfaces through-

## 7.4 Implementation Details

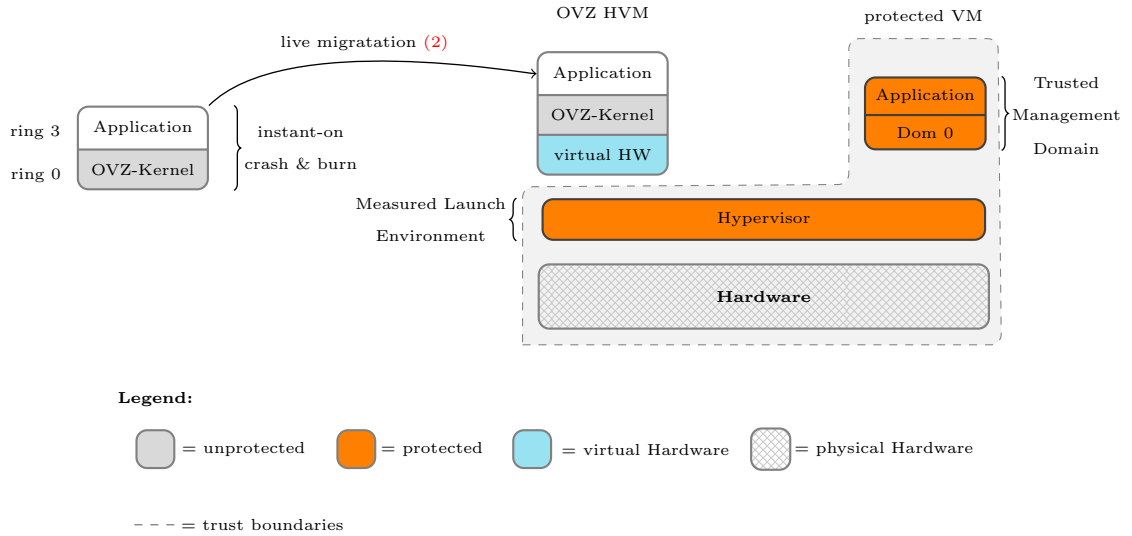


Figure 7.4: Step (2), live migration process.

out the platform operation. Secondly, we want to hide OS loading times. Once the LaLa has established the protected environment – as described in Section 7.4.4 – a special VM is spawned transparently in the background. This VM is an XEN hardware assisted VM (HVM), which itself contains an OVZ kernel (OVZ HVM in Figure 7.4). The HVM operates as the future container for the instant-on system. Once the OVZ migration process is complete, any number and combination of VMs might be spawned. In our prototype it is a full-featured Windows OS – see step (3) in Figure 7.5.

The OVZ VM in our prototype contains a stripped-down version of a Debian Linux distribution [58] including the OVZ patches to support live migration. This Linux system is optimised for a fast boot-up and only serves the purpose of hosting the instant-on system. The OVZ VM is the target VM for the instant-on application after migration. It has been previously outlined that the instant-on OS is considered untrusted. Consequently, in our current prototype, the OVZ VM is also considered untrusted and outside the protection boundaries. The trust boundaries of our Late Launch concept are highlighted by a broken line in Figure 7.5.

The thin OVZ abstraction layer offers less isolation than, for example, a HVM but also less overhead. This is necessary for the seamless live migration we require. The migration process could be optimised to increase performance and reduce latency by omitting SCP and by performing the migration via shared memory. This

## 7.4 Implementation Details

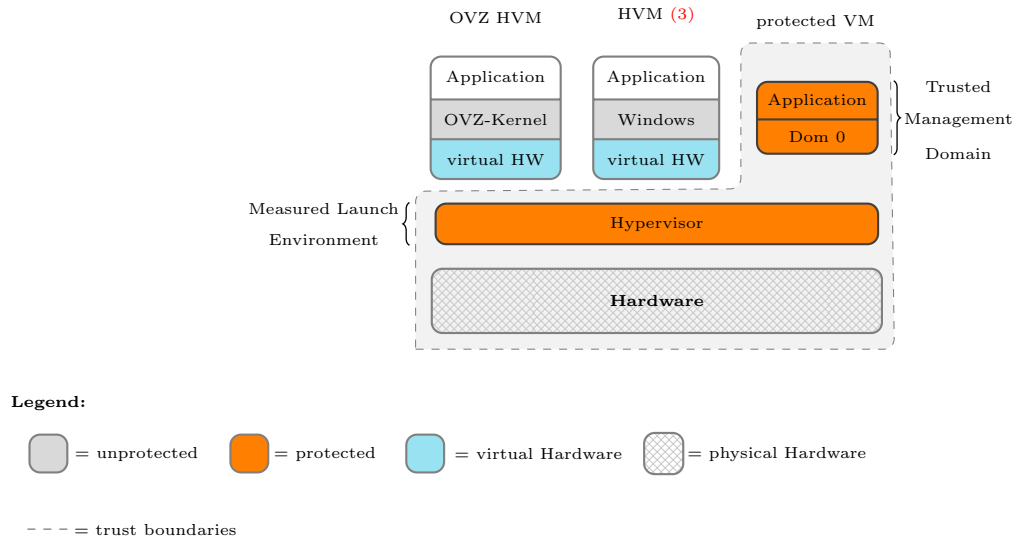


Figure 7.5: Step (3), launch full-feature OS.

would, however, certainly compromise the isolation boundaries of the virtualisation layer. Consequently, we opted for a more transparent, yet slower network migration approach.

### 7.4.6 Trusted Management Domain

We have included the Trusted Management Domain (TMD) in the TXT policy enforcing mechanism in order to be able to provide a trusted management component. Our TMD design goals are therefore:

- Manage the platform’s physical and virtual resources.
- Remove complexity and reduce the TMD’s TCB.
- Provide trusted management capabilities for the platform.
- Provide an entry point for more complex policy decisions.

The TMD is contained within the protection boundaries of the TXT hardware, which means that the integrity of the TMD will be checked against the pre-defined LCP before it can be launched. In this way we can ensure that only well-known

## 7.4 Implementation Details

---

configurations of TMDs and hypervisors can be executed. Consequently, we consider the TMD alongside the hypervisor as a trusted component. The trust boundaries are also illustrated by a dashed line in Figure 7.5.

First and foremost, the TMD serves the purpose of controlling the physical hardware on the platform as well as delegating resources to VMs. The TMD is similar to XEN’s Domain 0 concept [25]. Our approach differs, however, as we do not give the platform’s user the possibility to directly interact with the TMD. Furthermore, we include device drivers in the TMD and customise it to match only a specific platform and chipset. This customisation allows us to minimise the amount of software and drivers necessary and, thus, to reduce the TCB. Additionally, Intel’s ACM code is designed to work on a specific chipset family, which allows us to make assumptions about the hardware platform being used. We utilise this and tailor the kernel of our TMD to fit a specific platform family that matches the ACM. By doing so, the code base of the TMD kernel can be further reduced to only the set of drivers necessary for a specific set of hardware. We consider this to be an advantage as we do not sacrifice much transparency but gain security through reduced complexity in return.

As discussed in Section 7.3.2, the LCP policies play a critical role as they determine which hypervisor and management domain can be launched. They therefore represent a “white” list of pre-defined and trusted configurations. The LCP describe the acceptable integrity metrics of the hypervisor code; the code of the management domain; as well as a valid state of the platform, reflected by PCR values. At this early stage of the platform’s boot process policy capabilities are limited to integrity metrics over binaries and configuration files, thus we refer to them as simple policies (first-level PDP). As described in Section 7.3.2 and outlined in Figure 7.6, the LCPs provide the Policy Decision Point while the ACM acts as the first-level Policy Enforcement Point. Figure 7.6 illustrates the hierarchical layout and position of complex policies in relation to simple – LCP-based – policies. In our prototype we re-use existing code from Intel’s TXT tboot reference implementation [128] to define and install platform policies in the TPM.

The TMD does however, offer a strong basis to host more complex policies. As we discussed earlier, LCPs only offer simple policies in the form of integrity metrics over binaries and configuration files. We argued in Section 5.5.6 that a multi-layered hypervisor on different rings (H0/H1) can provide strong policy decisions and enforcement but, it only can provide a limited context for those decisions. Consequently, we regard hypervisor-based policies as limited policies (second-level



## 7.5 Security Analysis

---

environment as failing to do so will delay the next start-up until the memory is completely cleared of any remaining secrets. This allows us to protect the environment against cold-boot attacks as discussed in Section 7.2.

## 7.5 Security Analysis

In the previous sections, we discussed how we achieve our security goals, outlined in Section 7.2.1. We meet our goal of separating legacy from trusted components by allowing the parallel execution of multiple VMs. Legacy applications can be executed unhindered, while trusted components are protected by policy enforcement and strong isolation guarantees. Our goal to deliver a more trustworthy security service is met by incorporating a Trusted Management Domain into the policy enforcement mechanism and protecting it from unauthorised access.

### Existing Attacks on TXT

Although TXT has only been recently introduced attacks have already been carried out on this technology. In February 2009, Wojtczuk and Rutkowska [287] presented a proof-of-concept attack against the TXT extensions. The attack is mainly based on the fact that there exists a more privileged mode of operation on all Intel platforms – the SMM. As described in Section 4.2.4, the SMM code runs on a more privileged level than the hypervisor and is not designed to be accessible from any of the unprivileged levels on top. We also have outlined the current platform limitations in Section 4.2.4 and, as discussed by Grawrock [103], the SMM should be shielded by an STM. Currently there is no known implementation of an STM available. This allows SMM attacks to be carried out on some platforms. We consider this a shortcoming in the current implementation state of TXT rather than a fundamental design flaw. We assume that this will be addressed in future TXT implementations and that the security architecture as such remains intact.

In late 2009, Wojtczuk’s and Rutkowska’s attack was followed by an alternative approach to circumvent the TXT protection [289]. This alternative was based on a software bug in the ACM which tricked the chipset into protecting the wrong memory regions from DMA [289]. As a result a malicious local user could exploit this vulnerability for privilege escalation. Intel addressed the issue and released a

## 7.5 Security Analysis

---

fixed ACM version in December 2009 [126]. Newer chipsets require a matching ACM and therefore are likely to be unaffected by this bug. However, existing platforms will still be allowed to execute vulnerable ACM versions as there are no mechanisms to revoke existing ACMs. Consequently, the LCPs must be set accordingly to prevent vulnerable ACMs from being used.

TXT is designed to defend against software attacks and cannot prevent software compromises caused by software bugs. It is unfortunate that the previous bug was carried by the ACM, which is responsible for a secure platform initialisation and therefore must be inherently trusted. Similarly, if an MLE is vulnerable to a remote exploit TXT might, at best, be able to mitigate the effects. For instance, the LCPs can prevent known vulnerable software from being launched, while the hardware extension can isolate compromises to a certain extent.

### Threat Model

In terms of hardware, we consider an adversary who can perform simple hardware-based attacks, for instance attaching a debugger to slow system buses. We assume, however, that the adversary is unable to carry out sophisticated attacks, such as high-speed bus snooping or similar. We follow here the assumptions made by the TCG on hardware-based attacks [261]. An adversary might compromise a DMA capable device to directly access arbitrary memory. To mitigate DMA attacks we utilise the protection strategies outlined by Grawrock [103]. Our threat model at this stage excludes covert channels and timing channels between cooperating VMs.

### Isolation Attack

An adversary may succeed in compromising any of the VMs on the platform, for instance by exploiting a software vulnerability, such as a buffer overflow, in an application running inside a VM. This could affect the instant-on OS, the private VM or any other VM and is difficult to prevent. However, the LCPs can be selected so that only hardened VMs, without known vulnerabilities, are being used. Therefore, if the VM does not contain an exploitable vulnerability, the attacker must first succeed in circumventing the policy enforcement. Nevertheless, the trusted hardware and the trusted hypervisor offer a certain confidence in the isolation of co-resident VMs.

## 7.5 Security Analysis

---

Should one of the VMs be compromised the breach can be contained and access to any of the other VMs is prevented.

### **First-level Policy Attack**

A local user could try to circumvent the policies enforced by the platform owner, for instance by starting a hypervisor that is different from the pre-defined hypervisor environment. We rely on the first-level policy enforcement explained in Section 7.3.1 to prevent any circumvention. In order to bypass the first-level policy enforcement mechanism, an attacker would have to bypass the ACM enforcement (PEP) or must succeed in manipulating the LCPs (PDP) in the TPM. Earlier in this section we discussed the fact that the circumvention of the ACM is possible by, for example, exploiting a software bug in the ACM or by abusing the SMM. We consider this to be a shortcoming in the current state of TXT and therefore regard the concept as intact and a circumvention of TXT as infeasible in the foreseeable future.

A local adversary can boot into a special OS using an alternative boot medium, for instance CD/USB media, in order to subvert the instant-on OS. The instant-on system is outside the trust boundaries and hence not protected in any form. The local adversary, however, might try to gain access to the protected VM and possible secrets. We rely on the TVDI mechanism discussed in Chapter 6 to ensure integrity and confidentiality of security sensitive VMs and the TMD. For a successful attack on the TVDI disk encryption mechanism the attacker must succeed in gaining access to the TVDI metafile. While the system is powered down the metafile is protected by the TPM. During operation an attacker must successfully gain access to the TVDI's encryption key in system memory, thus the attacker must be able to successfully carry out an isolation attack.

### **Second-level Policy Attack**

In order to circumvent the second-level policy enforcement an attacker must successfully compromise the hypervisor (PEP) on  $H0$ , or the hypervisor (PDP) on  $H1$ . The hypervisor operates on simple hardware interfaces with little interaction with the outside world, hence its attack surface is very limited. Nevertheless, an attacker might be able to exploit a software vulnerability within the hypervisor and success-

## 7.5 Security Analysis

---

fully compromise either layer to bypass the enforcement or modify the policies. We therefore rely on the first-level policy enforcement to ensure only trusted hypervisors without known vulnerabilities are being executed. If the trusted hypervisor remains uncompromised an attacker must successfully modify the policies themselves. Policies, however, are protected by a hash in the TPM's NV memory. Therefore, in order to modify the policies, an attacker must successfully compromise the TPM's NV memory or gain access to the owner's authorisation secret.

### Third-level Policy Attack

As the policy enforcement is not carried out by the TMD, but by the hypervisor layer below, an attacker must successfully modify the active policy decision within the TMD or modify the stored policies. To subvert the TMD an attacker must either carry out an isolation attack or subvert any of the trusted layers beneath. As the TMD is the most complex piece of software, it potentially possesses the most vulnerabilities. We therefore rely on the previous policy enforcement mechanisms to ensure only TMDs without known vulnerabilities are being executed and, additionally, rely on the second-level policy enforcement to ensure the TMD executes a trusted PDP. In order to successfully modify the stored policies the attacker must again compromise the TPM's NV memory or gain access to the owner's authorisation secret.

### Availability Attack

We do not consider a denial of service attack from a local attacker. A local attacker could always simply power down the platform to disrupt services. However, a remote adversary in possession of ring 0 privileges could invoke the GETSEC[SENDER] instruction to enter a measured environment. As GETSEC[SENDER] will fail if the platform is already in a measured environment or in an error state, this would prevent the legitimate user from executing it. A remote adversary would first need to compromise the instant-on system and successfully gain ring 0 privileges. If the platform already has executed a Late Launch GETSEC[SENDER] is no longer available for a remote attacker until the environment has been torn down. To forcibly exit the measured environment an adversary must first overcome the previously described protection methods and gain full control over the platform.

## 7.6 Discussion

Our key contribution in this chapter can be summarised as follows:

- Provide a novel way to combine existing trust and virtualisation technology to improve on the trusted virtual infrastructure in a secure yet user-friendly way.
- Increase isolation and separate legacy from trustworthy (attestable) applications in a manner which does not compromise user experiences.
- Provide trusted security services and policy enforcement mechanisms in the form of a Trusted Management Domain.

There are currently many commercial and non-commercial instant-on solutions available, such as Splashtop Linux [62], Xandros Presto [291], Phoenix Hyperspace [206] and many more. Solutions such as Splashtop Linux or Xandros Presto, on the one hand, only offer support to either an instant-on or a full-feature OS. Hyperspace, on the other hand, is able to deliver a hybrid approach allowing the use of a full-feature OS alongside an instant-on system but it is lacking any trusted components. We certainly share the goal of providing an instant-on approach with minimal boot time, similar to all instant-on solutions. We distinguish our work however, with the ability to transparently launch all necessary components in the background and, more importantly, by providing trusted security services. This also allows us to reduce OS loading time, even after a cold boot, and to minimise the overhead of providing security services. In addition, the concept of migrating a live instant-on system into a set of nested virtualisation technologies, to maintain the user experience and transparency, is unique to our design and differentiates ours from all other approaches.

As discussed in Section 7.4.3, the instant-on OS is currently untrusted. We do not consider this to be a shortcoming in our design because the instant-on system is treated as a private VM and we provide a trusted and policy enforced VM after the system has been fully booted. Nevertheless, in some cases it might be desired to include the instant-on OS into the chain of trust. In our current model this requires the instant-on OS or applications to be migrated into a special HVM, whilst maintaining the confidentiality and integrity. With the present state of available hardware and software this is a challenging task. It seems more appropriate to follow our model but, instead of maintaining the trust while migrating, sacrificing

## 7.6 Discussion

---

usability by discarding the initial instant-on session and launching a trusted VM after the Late Launch.

We have extended the concept of a trusted hypervisor – discussed in Chapter 5 – by including the TMD within the trust boundaries. This allows the TMD to act as a policy decision point where the trusted hypervisor might not hold sufficient context. In contrast to the hypervisor-based policy decisions – discussed in Section 5.5.6 – policies based in the TMD allow for a much broader and more complex policy model. The policy context within the hypervisor is limited and thus, for sophisticated policies, the TMD offers a more pragmatic, yet not so strong, contextual relationship. Consequently, the TMD is a strong entry point for a wide range of security management functionality.

We are able to enhance system security and trusted manageability of our prototype in respect of the requirements discussed in Section 4.1.2. The work by McCune et al. [168] to reduce the TCB is similar to ours in the goals and the technology used. It is important to note however, that our objective was not to reduce the TCB to its bare minimum, but to balance between usability, manageability and trusted components. Also Qubes OS [224] is closely related to our design but it is an early stage of development and many of security design features are not yet fully implemented. Moreover, our design implements a layered hypervisor design and further implements strong confidentiality and integrity protection for even remotely executed VMs. Consequently, we rely on the concepts discussed in Chapter 5 and Chapter 6 in order to be able to provide trusted components. Here, our work ideologically overlaps with the OpenTC project [197] and many other TC projects: we aim to reduce system-related threats, vulnerabilities and errors as well as reducing the overall attack surface. At the same time, we want to preserve usability and provide trustworthy components as an entry point for trusted management functionality.

The main security benefit results from the segregation and isolation of untrusted and trusted VM and the containment of the latter. Moreover, the hardware supported policy enforcement mechanism allows us to make strong assumptions about the state of the hardware and software configuration. Furthermore, the Late Launch concept enables us to freely explore any combination of trusted and untrusted VM on the same physical platform in order to address any legacy concerns.

## 7.7 Summary

In this chapter we have provided the motivation and goals for our novel Late Launch scheme. We also outlined the technological building blocks available to us and have demonstrated how we can exploit these to realise our prototype trusted virtual architecture. Our concept has been described in detail. In the security discussion we considered past and present attacks on the TXT platform. Moreover, we have outlined our adversary model and analysed a number of attacks that an adversary could carry out.

Our Late Launch concept preserves existing user experiences by allowing a parallel execution of legacy OSs and secure components in order to satisfy the requirement of compatibility (4.1.2.5). The segregation of trusted and untrusted components further reduces the overall complexity (requirement 4.1.2.4) as well as allowing a more fine-grained attestation of trusted components (requirement 4.1.2.2). Furthermore, the adaption of the TMD to a specific platform minimises its TCB, further reduces the complexity (requirement 4.1.2.4) and thus increases the isolation guarantees (requirement 4.1.2.1). We integrate Intel's TXT technology for hardware protection to strengthen the isolation guarantees (requirement 4.1.2.1) and further incorporate the TXT's policy enforcement mechanisms to ensure only trusted components are used (requirement 4.1.2.3).

## Part IV

# Conclusion

# Chapter 8

## Summary and Conclusion

### Contents

---

<b>8.1</b>	<b>Summary</b>	<b>176</b>
<b>8.2</b>	<b>Conclusion</b>	<b>180</b>
<b>8.3</b>	<b>Directions for Future Work</b>	<b>181</b>
8.3.1	Runtime Integrity	181
8.3.2	Inter VM Communication	183

---

*There are known knowns. These are things we know that we know. There are known unknowns. That is to say, there are things that we now know we don't know. But there are also unknown unknowns. These are things we do not know we don't know.*

– Donald Rumsfeld

### 8.1 Summary

In this thesis we have examined how Trusted Computing and virtualisation technology can be united to move towards a trusted virtualisation infrastructure. We investigated the vulnerabilities and strengths of each technology and outlined how they could be combined and improved to increase the overall trustworthiness of

## 8.1 Summary

---

a virtualised platform. We also investigated weaknesses of current x86 commodity platforms, which could undermine the existing trust assumptions, and we then showed how their impact could be mitigated. We found that trusted virtualisation needs more protection mechanisms and guarantees than the sum of its building blocks can provide. We identified the following requirements for a sensible trusted virtual environment:

**Isolation requirement (4.1.2.1):** We determined that the process isolation provided by commodity OSs is insufficient for ensuring the integrity and confidentiality of security sensitive applications.

**Attestation requirement (4.1.2.2):** We have pointed out how current hypervisors lack the ability to cryptographically identify themselves and thus require more precise attestability.

**Policy requirement (4.1.2.3):** We require a trusted virtualisation layer to implement policy enforcement, as well as being able to make reliable policy decisions.

**Complexity requirement (4.1.2.4):** The complexity of modern commodity x86 platforms must be reduced as the complexity is potentially the only property a platform can attest to.

**Compatibility requirement (4.1.2.5):** Compatibility and the support for unmodified legacy applications is an important property to satisfy in order to maintain user experiences.

We discussed these requirements in part II and identified in Section 4.3 the three key issues we addressed in part III of this thesis:

- **Increase the trustworthiness of the hypervisor layer.**
- **Deliver efficient, trusted and attestable storage for VMs.**
- **Provide trusted separation of co-resident VMs, and provide trusted security services.**

To improve on the trusted virtual infrastructure we adopted a layered approach and addressed the above requirements on key levels of abstraction – the hypervisor, the storage and the VM space. In the following paragraphs the contribution of this thesis with regards to the requirements and goals identified above are summarised:

## 8.1 Summary

---

### Separating Trusted Computing Base with Hardware

In Chapter 5 we examined how the TCB of a hypervisor could be reduced (requirement 4.1.2.4) to increase the value gained from attesting (requirement 4.1.2.2) to a virtual platform's properties. The assumption that a hypervisor is in exclusive control of all hardware resources and the most privileged piece of code is paramount in order to express any trust assumptions about upper layers. Both are essential requirements for a trusted virtualisation layer as we identified in Chapter 4. To improve the isolation properties (requirement 4.1.2.1) and the value gained from attesting to these, we proposed to fully utilise the newly available and currently unused hardware protection mechanisms on virtualisation-enabled CPUs. Moreover, the use of virtualisation allows us to provide backward compatibility to existing legacy systems (requirement 4.1.2.5) at the same time.

We reassessed the definition of TCB and discussed the application of our scheme to current platform weaknesses, which we identified in part II. For instance, most recent security issues related to virtualisation are based on hardware limitations such as ACPI and SMM as well as software's inability to sufficiently isolate processes from each other. While it is difficult to safeguard the SMM without support from the platform manufacturer, we demonstrated in Chapter 5 that it is possible to increase isolation and to limit the ACPI handler's effect on the TCB.

We believe that the current advances in hardware virtualisation technology have created a window of opportunity for creative security solutions. For example, the intended x86 protection scheme has never been implemented – forcing many security solutions to be erroneously and expensively retrofitted. Dissecting the hypervisor into a layered and hardware-protected entity further helps to reduce the hypervisor's complexity, while increasing its isolation guarantees and the value derived from attesting to these properties.

Our approach, of a layered TCB, also allowed us to improve on the policy (4.1.2.3) requirement by increasing the confidence in policy decisions. Separating and maintaining the policy decision context within the hypervisor enabled us to ensure that the decisions originating from within this setting are free from any interferences.

## 8.1 Summary

---

### Trusted Virtual Disk Images

In Chapter 6 we built on the hypervisor improvements described in Chapter 5. Chapter 6 discussed our TVDI design, which makes use of the TPM and recent advances in hardware virtualisation technology to transparently provide integrity and confidentiality to VMs. Hence, our TVDI proposal allows us to provide trusted storage seamlessly to guest OSs. The creation of a bound metafile per disk image enables a local or remote party to gain confidence in the disk image used by a particular VM (requirement 4.1.2.2). Therefore, a VM can attest to its disk image while a remote party can enforce the usage of a particular image, hypervisor or a VM in a particular state (requirement 4.1.2.3). As outlined in Chapter 4, attestation and policy control are both essential requirements in a trusted virtualisation infrastructure. Additionally, our TVDI design can be transparently incorporated into a virtual environment to satisfy the compatibility (4.1.2.5) requirement.

We are able to provide those security attributes even if the underlying storage subsystem cannot guarantee any of these. Thus, data inside a TVDI is protected as a whole, including log files, applications and legacy OSs. Moreover, potential compromises of the storage subsystem do not effect the security of co-resident VMs (requirement 4.1.2.1). As a result, the storage system is removed from the TCB, which reduces complexity (requirement 4.1.2.4) and further increases the isolation.

### LaLa: A Late Launch Application

In Chapter 7 we exploited the newly available DRTM and, together with the building blocks investigated in the preceding chapters, aimed to construct a more trustworthy virtual infrastructure. We therefore incorporated the trusted hypervisor established in Chapter 5 and the trusted storage discussed in Chapter 6. Chapter 7 introduced our unique LaLa concept that combines the latest hardware virtualisation and trust technologies to segregate OSs on different trust levels. Our approach preserves existing user experiences and maintains compatibility by allowing a parallel execution of legacy OSs and secure components (requirement 4.1.2.5). We discussed, in Chapter 4, the fact that the lack of legacy support is a major obstacle for the adoption of more secure OSs and hence we consider maintaining compatibility and preserving user experiences as an important requirement.

## 8.2 Conclusion

---

The segregation of multiple personae on a single platform helps to reduce the complexity (requirement 4.1.2.4) and increase the isolation (requirement 4.1.2.1) of today's OSs. Modern OSs have grown uncontrollably in size and functionality thus they contribute to a large TCB. We have seen in Chapter 4 that the value of attesting to such a large TCB is debatable. By segregating, for instance TMD and private VMs, we reduced the overall complexity and increased the meaning of attesting to trusted components (requirement 4.1.2.2).

We also devised the concept of a TMD, which is included within the trust boundaries, in order to be able to make policy decisions and enforce those decisions with our trusted hypervisor. Further, we incorporated Intel's TXT policy enforcement mechanism to enforce a specific trusted hypervisor and TMD. It enables the platform to attest to its hypervisor and TMD (requirement 4.1.2.3). Consequently, the TMD is a strong entry point for security services and a wide range of management functionality where the hypervisor cannot maintain sufficient context to make according policy decisions.

## 8.2 Conclusion

The current design and complexity of commodity x86 platforms seriously impacts the security of such systems. We have seen that some researchers and manufacturers turned to MAC-based systems in order to contain potential application compromises. Other research indicates that virtualisation and its isolation properties are better suited for containment of rogue software. While this is certainly true to some extent, we have discussed various shortcomings in current virtualisation technology and considered the improvement of virtualisation by applying trusted technology in order to move towards a more trustworthy virtual infrastructure.

We found that a sensible trusted virtualisation layer requires more protection guarantees than simply the combination of the Trusted Computing and virtualisation building blocks. Security requires a holistic approach and, while it is relatively simple to improve on a particular security issue, it requires more effort to create an overall secure system. This thesis offers an insight into current shortcomings of trusted virtualisation and indicates how some of those shortcomings could be overcome.

### 8.3 Directions for Future Work

---

We have mainly focused on low-level platform properties as there are many good reasons to place security on a lower level. For instance, any level of security can be compromised if an attacker manages to compromise the layer below. To evaluate the security properties of a given layer it must be ensured that this layer cannot be bypassed. Hence, there must be assurances that the layer on which security principles are built is trustworthy and cannot be compromised. Virtualisation's hardware interfaces consist of a simple structure which can help to evaluate the security properties to a higher level of assurance. The generic hardware interfaces also permit a great degree of portability while hiding implementation details from developers. Additionally, simple low-level hardware interfaces potentially offer greater performance over more complex, high-level interfaces, as the implementation can be platform-specific and optimised for a particular use case.

In this thesis we extended the current perception of what trust in a virtualised infrastructure means from a technical perspective and how it can be improved. We challenged the view of current trusted virtualisation research as well as the view of the future of commodity virtualisation platforms themselves. We investigated how we can move towards trusted virtualisation with the current technology available to us. This thesis proposed several concepts which when combined, allow us to move towards trustworthy virtualisation.

### 8.3 Directions for Future Work

Hardware-based virtualisation and Trusted Computing technology, such as the DRTM, are recent hardware advances and are likely to evolve over time. While we heavily use and depend on those technologies, there are many aspects which need further exploration. The issues identified in this thesis are of a technical nature. Nevertheless, many more issues are based on usability, commercial and even political and social issues. However, we consider that two technical aspects are particularly worthy of further exploration.

#### 8.3.1 Runtime Integrity

The fact that almost all modern commodity platforms implement a 'Von Neumann' architecture – a single memory storage structure that mixes data and code – allows

### 8.3 Directions for Future Work

---

many programming errors to be exploited. Mistakes, such as buffer overflows, can be used to execute malicious code on data areas and therefore trivially translate into security vulnerabilities. Other architectures, for instance the ‘Harvard’ architecture, in which data and code is separated are rarely found in commodity platforms. This is mostly due to performance and efficiency benefits of the ‘Von Neumann’ architecture. Moreover, as pointed out by Karger et al. [136], most security issues identified 40 years ago in Multics can still be found in the OSs of today.

While the DRTM can ensure no untrusted hypervisor code is loaded – thus providing loadtime integrity – the hypervisor’s integrity also needs to be protected during runtime. The hypervisor should be strictly prevented from executing data as well as unauthorised code in memory, for instance, code or data located in a VM’s memory context. ‘Harvard-like’ features such as the *No eXecute* (NX) bit have already been implemented by Intel [124] and AMD [39] for non-virtualised systems but these need further adaption to match the requirements of a virtualised platform. Ensuring runtime integrity is critical in the presence of potentially exploitable software vulnerabilities. Even though the code base of a hypervisor is relatively small compared to traditional OS kernels and therefore has potentially fewer software bugs, a study of the National Vulnerability Database [192] reveals that common hypervisors are not free of exploitable vulnerabilities. The concepts and methods discussed in this thesis improve the loadtime integrity as well as the isolation of co-resident VMs but they cannot prevent the exploitation of software vulnerabilities within the trusted hypervisors.

In recent years virtualisation has been utilised as a tool to inspect the integrity of guest OSs but little consideration has been given to the integrity protection of the hypervisor itself. Livewire [87], SecVisor [238] and HookSafe [281] are a few examples of guest OS integrity protection. However, they assume a trusted hypervisor is present on which they faithfully rely to inspect the memory integrity of VMs. To protect the integrity of VMs, it is therefore necessary to provide integrity protection for the hypervisor. Early efforts to provide hypervisor runtime integrity include DeepWatch [41]. DeepWatch is a proof-of-concept design of a chipset-based rootkit detection system, embedded into the chipset’s firmware. It is capable of scanning, detecting and possibly sanitising a virtualisation-based rootkit. Unfortunately, the signature-based approach has certain disadvantages. Firstly, the malware could adapt easily by changing its signature and secondly, the low-level position of the scanner renders maintenance of its database difficult.

### 8.3 Directions for Future Work

---

Recently, HyperSentry [20] has been proposed which uses the SMM to inspect the runtime integrity of a hypervisor. While HyperSentry is periodically invoked to ensure the hypervisor’s integrity is still intact, HyperSafe [280] promises to ensure the runtime integrity and control-flow integrity in the presence of exploitable software vulnerabilities. It is however more important to preserve the integrity of low-level code, such as the hypervisor and SMM, rather than detecting modifications afterwards.

Further investigation and development into integrity protection will greatly enhance the security guarantees of trusted virtual platforms. Our future work will include the research into existing hardware-based memory protection strategies – such as the NX bit – and how those strategies could be applied to a virtual context. Furthermore, we want to examine how the existing hardware could be extended with special hypervisor protection mechanisms. This extension could be achieved with an NX bit dedicated to the hypervisor context. Since hardware vendors are increasingly willing to dedicate chip real estate to security functions this is a concept worth exploring.

#### 8.3.2 Inter VM Communication

One of the outstanding properties of virtualisation is its ability to isolate co-resident OSs on the same physical platform. While isolation is an important property from a security perspective, co-resident VMs often need to communicate and exchange a considerable amount of data. Despite ever growing accomplishments to improve VM performance, research has shown that network virtualisation performance is still poor [175]. As demonstrated by Menon et al. [174], performance can be improved if carefully designed. The resulting improvement, however, is still not comparable to inter-process communication on a non-virtual platform [295]. Currently many efforts are undertaken to improve network virtualisation performance as well as inter VM communication. One example of IVMC would be two processes on the same physical machine – in different VMs – which want to exchange data in some form. The processes in the VM have to communicate via the standard network interface as if they did not share the same physical host. This clearly inflicts unnecessary performance penalties on both VMs: data has to be encapsulated, addressed, transmitted and checked via the network stack as well as the virtualisation layer.

### 8.3 Directions for Future Work

---

The cost and improvement of I/O virtualisation has received a lot of attention [30, 221]. Compared to CPU or memory virtualisation, I/O device virtualisation is still considered costly and presents a performance bottleneck. Though the secure and efficient I/O hardware sharing is a popular topic in academic and commercial research only minor attention is given to secure and efficient IVMC itself. To address the issue of IVMC performance many solutions propose the use of shared memory to tunnel the isolation boundaries. Shared memory seems like the obvious solution to address the performance aspect but it has certain drawbacks in terms of security, handling and transparency [95]. Also, even if shared memory is being used, the performance of different implementations varies considerably [95].

With the ever increasing amount of VMs simultaneously hosted on a single platform further exploration of the security and performance properties of IVMC is necessary. We have already discussed in Section 5.5.1 the possibility of using the existing hardware protection capabilities to host an IVMC proxy. Nevertheless, realising efficient IVMC without profound guest OS modification remains a difficult task. Our future work will therefore investigate the feasibility of an IVMC enabled, guest OS network proxy and para-virtualised network protocol. The proxy will be placed beneath the guest's network layer and will be capable of using the IVMC proxy as outlined in Section 5.5.1. Since most VMs implement a para-virtualised network interface card to avoid performance penalties, the additional installation of a para-virtualised network protocol and proxy are not unreasonable assumptions. By modifying the network protocol, multiple integrity checks could be avoided and performance can be improved. The proxy will help to maintain compatibility and transparency by applying transparent and efficient routing decisions. We believe that such a design will offer the best compromise in terms of performance, compatibility and manageability to improve the trusted virtual infrastructure as we move towards trustworthy virtualisation.

# Bibliography

- [1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 10(03):179–192, August 2006.
- [2] ACPI. Advanced Configuration and Power Interface Specification. <http://www.acpi.info/spec.htm>, accessed June 2009.
- [3] R. J. Adair, R. U. Bayles, L. W. Comeau, and R. J. Creasy. A Virtual Machine System for the 360/40. Technical Report Report No. 320-2007, IBM Corporation Cambridge Scientific Center, May 1966.
- [4] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, ACM, October 2006.
- [5] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 1–10, New York, NY, USA, ACM, October 2006.
- [6] A. Alkassar, M. Scheibel, C. Stüble, A.-R. Sadeghi, and W. Marcel. Security Architecture for Device Encryption and VPN. In *ISSE '06: Information Security Solution Europe*, October 2006.
- [7] AMD. AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual. Technical Report 33047 rev. 3.01, AMD Corporation, May 2005.
- [8] AMD. AMD I/O Virtualization Technology (IOMMU) Specification 1.2. Technical report, AMD Corporation, February 2007.

- [9] AMD. AMD64 Architecture Programmer's Manual, Volume 2: System Programming, Rev. 3.12. Technical Report 24593 rev. 3.14, AMD Corporation, September 2007.
- [10] AMD. AMD-V Nested Paging. White Paper 1.0, AMD Corporation, July 2008.
- [11] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, Bedford, MA, October 1972.
- [12] M. J. Anderson, M. Moffie, and C. I. Dalton. Towards Trustworthy Virtualisation Environments: Xen Library OS Security Service Infrastructure. Technical report, Hewlett-Packard Laboratories, April 2007.
- [13] R. Anderson. Cryptography and competition policy: Issues with 'Trusted Computing'. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 3–10, New York, NY, USA, ACM, July 2003.
- [14] Apple. Mac OS X Security. [http://images.apple.com/macosx/security/docs/MacOSX\\_Security\\_TB.pdf](http://images.apple.com/macosx/security/docs/MacOSX_Security_TB.pdf), accessed April 2010.
- [15] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 65, Washington, DC, USA, IEEE Computer Society, May 1997.
- [16] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, and J. M. Smith. Automated Recovery In a Secure Bootstrap Process. In *Proceedings of Network and Distributed System Security Symposium*, pages 155–167. Internet Society, March 1998.
- [17] I. Arce. Ghost in the Virtual Machine. *IEEE Security and Privacy*, 5(4):68–71, July 2007.
- [18] K. Ashcraft and D. R. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 143, Washington, DC, USA, IEEE Computer Society, May 2002.
- [19] C. R. Attanasio, P. W. Markstein, and R. J. Phillips. Penetrating an operating system: a study of VM/370 integrity. *IBM Systems Journal*, 15(1):102–116, March 1976.

- [20] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *CCS '10: Proceedings of the 17th ACM Conference on Computer and Communications Security*, Chicago, Illinois, USA, ACM, October 2010.
- [21] B. Balacheff, D. Chan, L. Chen, S. Pearson, and G. Proudler. How can you trust a computing platform? In *Proceedings of Information Security Solutions Europe*, Spain, September 2000.
- [22] S. Balfe. *Secure Payment Architectures and Other Applications of Trusted Computing*. PhD thesis, Royal Holloway University of London, <http://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-10.pdf>, February 2009.
- [23] S. Balfe, E. Gallery, K. G. Paterson, and C. J. Mitchell. Challenges for trusted computing. Technical Report RHUL-MA-2008-14, Department of Mathematics, Royal Holloway, University of London, February 2008.
- [24] S. Balfe, A. D. Lakhani, and K. G. Paterson. Trusted Computing: Providing Security for Peer-to-Peer Networks. In *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*, pages 117–124, Washington, DC, USA, IEEE Computer Society, August 2005.
- [25] 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. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, ACM, October 2003.
- [26] D. E. Bell. Looking Back at the Bell-La Padula Model. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 337–351, Washington, DC, USA, IEEE Computer Society, December 2005.
- [27] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report 2547, MITRE, March 1973.
- [28] D. E. Bell and L. J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report ESD-TR-75-306, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, Bedford, MA, March 1975.
- [29] S. M. Bellovin. Computer security—an end state? *Communications of the ACM*, 44(3):131–132, March 2001.

- [30] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. van Doorn, J. Nakajima, A. Mallick, and E. Wahlig. Utilizing IOMMUs for Virtualization in Linux and Xen. In *OLS '06: The 2006 Ottawa Linux Symposium*, pages 71–86. Linux Symposium Inc, July 2006.
- [31] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: virtualizing the trusted platform module. In *USENIX-SS '06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, USENIX Association, August 2006.
- [32] S. Berger, R. Cáceres, D. Pendarakis, R. Sailer, E. Valdez, R. Perez, W. Schildhauer, and D. Srinivasan. TVDc: managing security in the trusted virtual datacenter. *ACM SIGOPS Operating Systems Review*, 42(1):40–47, January 2008.
- [33] K. Biba, S. Ames, E. Burke, P. Karger, W. Price, R. Schell, and S. W.L. A Preliminary Specification of a Multics Security Kernel. In *ACM Computer Science Conference*, page 16, Washington, DC, USA, ACM, February 1975.
- [34] Bochs. bochs: The Open Source IA-32 Emulation Project. <http://bochs.sourceforge.net>, accessed December 2009.
- [35] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1):104–123, March 1989.
- [36] S. Bratus, P. C. Johnson, A. Ramaswamy, S. W. Smith, and M. E. Locasto. The cake is a lie: privilege rings as a policy resource. In *VMSec '09: Proceedings of the 1st ACM workshop on Virtual machine security*, pages 33–38, New York, NY, USA, ACM, November 2009.
- [37] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, New York, NY, USA, ACM, February 2004.
- [38] D. Brumley and D. Song. Privtrans: automatically partitioning programs for privilege separation. In *SSYM '04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, USENIX Association, August 2004.

- [39] R. Brunner. Enhanced Virus Protection in AMD Opteron and AMD Athlon 64 Processors. [http://www.amd.com/us-en/assets/content\\_type/DownloadableAssets/dwamd\\_AMD64\\_WinHEC04\\_pubfinal.pdf](http://www.amd.com/us-en/assets/content_type/DownloadableAssets/dwamd_AMD64_WinHEC04_pubfinal.pdf), May 2004.
- [40] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 143–156, New York, NY, USA, ACM, November 1997.
- [41] Y. Bulygin and D. Samyde. Chipset Based Approach To Detect Virtualization Malware a.k.a. DeepWatch. <http://www.mnm-team.org/pub/Fopras/frit08/PDF-Version/frit08.pdf>, accessed August 2008.
- [42] Bundesamt für Sicherheit in der Informationstechnik. Certification report for processor resource/system manager (PR/SM) for the IBM system z10 EC GA1. Technical Report BSI-DSZ-CC-0460-2008, Bundesamt für Sicherheit in der Informationstechnik, October 2008.
- [43] S. Cabuk, C. I. Dalton, H. Ramasamy, and M. Schunter. Towards automated provisioning of secure virtualized networks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 235–245, New York, NY, USA, ACM, October 2007.
- [44] L. Campo-Giralte, R. Jimenez-Peris, and M. Patino-Martinez. PolyVaccine: Protecting Web Servers against Zero-Day, Polymorphic and Metamorphic Exploits. *IEEE Symposium on Reliable Distributed Systems*, pages 91–99, September 2009.
- [45] M. Carpenter, T. Liston, and E. Skoudis. Hiding Virtualization from Attackers and Malware. *IEEE Security and Privacy*, 5(3):62–65, May 2007.
- [46] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 385–395, Washington, DC, USA, IEEE Computer Society, May 2003.
- [47] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn. *A practical guide to trusted computing*. IBM Press, January 2008.
- [48] H. Chen, J. Chen, W. Mao, and F. Yan. Daonity - Grid security from two levels of virtualization. *Information Security Technical Report*, 12(3):123–138, May 2007.

- [49] P. M. Chen and B. D. Noble. When Virtual Is Better Than Real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, IEEE Computer Society, May 2001.
- [50] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review*, 42(2):2–13, March 2008.
- [51] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88, New York, NY, USA, ACM, October 2001.
- [52] Citrix Systems. blktap - Xen Wiki. <http://wiki.xensource.com/xenwiki/blktap>, accessed June 2008.
- [53] Codeweaver. WineHQ - Run Windows applications on Linux, BSD and MAC OS X. <http://www.winehq.org/>, accessed March 2010.
- [54] J. R. Crandall, G. Wassermann, D. A. S. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Temporal search: detecting hidden malware timebombs with virtual machines. In *ASPLOS XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 25–36, New York, NY, USA, ACM, October 2006.
- [55] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Research Devison*, 25(5):483–490, September 1981.
- [56] C. I. Dalton and C. Gebhardt. Booting a Computing Device. Patent application, October 2009. Application number 200903635.
- [57] Damn Small Linux. Damn Small Linux. <http://damnsmalllinux.org/>, accessed January 2009.
- [58] Debian. Debian. <http://www.debian.org/>, accessed January 2009.
- [59] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 15–24, Washington, DC, USA, IEEE Computer Society, March 2003.

- [60] Department of Defense. *Trusted Computer System Evaluation Criteria*, volume DoD 5200.28-STD. Department of Defense, December 1985.
- [61] B. des Ligneris. Virtualization of Linux Based Computers: The Linux-VServer Project. In *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 340–346, Washington, DC, USA, IEEE Computer Society, May 2005.
- [62] DeviceVM. Splashtop. <http://www.splashtop.com>, accessed January 2009.
- [63] S. W. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. Patent, May 2002. US 6397242.
- [64] Dm-crypt. dm-crypt: a device-mapper crypto target. <http://www.saout.de/misc/dm-crypt/>, accessed December 2009.
- [65] L. Duflot, D. Etiemble, and O. Grumelard. Using CPU System Management Mode to Circumvent Operating System Security Functions. <http://www.ssi.gouv.fr/fr/sciences/fichiers/lti/cansecwest2006-duflot-paper.pdf>, accessed April 2007.
- [66] L. Duflot, D. Etiemble, and O. Grumelard. Security Issues Related to Pentium System Management Mode. <http://www.cansecwest.com/slides06/csw06-duflot.ppt>, published April 2006.
- [67] L. Duflot, O. Grumelard, O. Levillain, and B. Morin. ACPI and SMI handlers: some limits to trusted computing. *Journal in Computer Virology*, November 2009.
- [68] L. Duflot, O. Levillain, and B. Morin. ACPI: Design Principles and Concerns. In *Trusted Computing: Second International Conference on Trusted Computing*, Lecture Notes in Computer Science, pages 14–28. Springer, February 2009.
- [69] L. Duflot, O. Levillain, B. Morin, and O. Grumelard. Getting into the SM-RAM: SMM Reloaded. <http://cansecwest.com/csw09/csw09-duflot.pdf>, published 2009.
- [70] L. Duflot, Y.-A. Perez, G. Valadon, and O. Levillain. Can you still trust your network card? <http://www.ssi.gouv.fr/IMG/pdf/csw-trustnetworkcard.pdf>, accessed September 2010.

- [71] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *Computer*, 34(10):57–66, October 2001.
- [72] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic Binary Translation and Optimization. *IEEE Transactions on Computers*, 50(6):529–548, June 2001.
- [73] S. Embleton, S. Sparks, and C. Zou. SMM Rootkits: A New Breed of OS Independent Malware. In *SecureComm 2008*, Istanbul, Turkey, ACM, September 2008.
- [74] P. England. Practical Techniques for Operating System Attestation. In *Trust '08: Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies*, pages 1–13, Berlin, Heidelberg, Springer, August 2008.
- [75] P. England and J. Loeser. Para-Virtualized TPM Sharing. In *Trust '08: Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies*, pages 119–132, Berlin, Heidelberg, Springer, August 2008.
- [76] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, ACM, December 1995.
- [77] European Multilaterally Secure Computing Base. European Multilaterally Secure Computing Base. <http://www.emscb.de>, accessed December 2010.
- [78] European Multilaterally Secure Computing Base. Turaya. <http://www.emscb.com/content/pages/turaya.htm>, accessed December 2010.
- [79] P. Ferrie. Attacks on Virtual Machine Emulators. [http://www.symantec.com/avcenter/reference/Virtual\\_Machine\\_Threats.pdf](http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf), published December 2006.
- [80] P. Ferrie, T. Ptacek, and N. Lawson. Don't Tell Joanna, The Virtualized Rootkit Is Dead. Black Hat USA [https://www.blackhat.com/presentations/bh-usa-07/Ptacek\\_Goldsmith\\_and\\_Lawson/Presentation/bh-usa-07-ptacek\\_goldsmith\\_and\\_lawson.pdf](https://www.blackhat.com/presentations/bh-usa-07/Ptacek_Goldsmith_and_Lawson/Presentation/bh-usa-07-ptacek_goldsmith_and_lawson.pdf), published July 2007.

- [81] B. Frantz, N. Hardy, J. Jonekait, and C. Landau. GNOSIS: A Prototype Operating System for the 1990's. In *Proceedings of SHARE 52*, pages 3–17, Chicago, Illinois, USA, SHARE Inc., March 1979.
- [82] K. Fraser. XEN Hypervisor Lines Of Code. <http://lists.xensource.com/archives/html/xen-devel/2008-09/msg00359.html>, accessed September 2008.
- [83] K. Fraser, H. Steven, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, October 2004.
- [84] E. Gallery. *Authorisation Issues for Mobile Code in Mobile Systems*. PhD thesis, Royal Holloway, University of London, <http://www.ma.rhul.ac.uk/static/techrep/2007/RHUL-MA-2007-3.pdf>, May 2007.
- [85] E. Gallery and C. J. Mitchell. Trusted computing: Security and applications. *Cryptologia*, 33(3):217–245, July 2009.
- [86] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 193–206, New York, NY, USA, ACM, October 2003.
- [87] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed System Security Symposium*, pages 191–206, San Diego, CA, USA, The Internet Society, February 2003.
- [88] T. Garfinkel and M. Rosenblum. When virtual is harder than real: security challenges in virtual machine based computing environments. In *HOTOS '05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 20–20, Berkeley, CA, USA, USENIX Association, June 2005.
- [89] C. Gebhardt and C. I. Dalton. LaLa: A Late Launch Application. In *The Fourth Annual Workshop on Scalable Trusted Computing*, Chicago, Illinois, USA, ACM, November 2009.
- [90] C. Gebhardt, C. I. Dalton, and R. Brown. Preventing hypervisor-based rootkits with Trusted Execution Technology. *Elsevier Network Security Newsletter*, 11:7–11, November 2008.

- [91] C. Gebhardt, C. I. Dalton, and A. Tomlinson. Separating Hypervisor Trusted Computing Base Supported by Hardware. In *The Fifth Annual Workshop on Scalable Trusted Computing*, Chicago, Illinois, USA, ACM, October 2010.
- [92] C. Gebhardt and A. Tomlinson. Secure Virtual Disk Images for Grid Computing. In *APTC '08: Proceedings of the 2008 Third Asia-Pacific Trusted Infrastructure Technologies Conference*, pages 19–29, Washington, DC, USA, IEEE Computer Society, November 2008.
- [93] C. Gebhardt and A. Tomlinson. Security Considerations for Virtualization. Technical Report RHUL-MA-2008-16, Department of Mathematics, Royal Holloway, University of London, April 2008.
- [94] C. Gebhardt and A. Tomlinson. Trusted Virtual Disk Images. In *2nd Conference on the Future of Trust in Computing*, Berlin, Vieweg & Teubner, July 2008.
- [95] C. Gebhardt and A. Tomlinson. Challenges for Inter Virtual Machine Communication. Technical Report RHUL-MA-2010-12, Department of Mathematics, Royal Holloway, University of London, August 2010.
- [96] Geeknet. Security-enhanced Linux. <http://selinux.sourceforge.net/>, accessed February 2009.
- [97] V. D. Gligor. Analysis of the Hardware Verification of the Honeywell SCOMP. *IEEE Symposium on Security and Privacy*, page 32, April 1985.
- [98] B. D. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in Retrospect. *IEEE Symposium on Security and Privacy*, page 13, May 1984.
- [99] B. D. Gold, R. R. Linde, M. Schaefer, and J. F. Scheid. VM/370 security retrofit program. In *ACM '77: Proceedings of the 1977 annual conference*, pages 411–418, New York, NY, USA, ACM, January 1977.
- [100] R. P. Goldberg. A Survey of Virtual Machine Research. In *Computer*, volume 7, pages 34–43. Honeywell Information Systems and Harvard University, IEEE Computer Society, June 1974.
- [101] D. Gollmann. *Computer security*. John Wiley & Sons, Inc., New York, NY, USA, July 1999.
- [102] D. Gollmann. Why Trust is Bad for Security. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 157(3):3–9, May 2006.

- [103] D. Grawrock. *Dynamics of a Trusted Platform*. Intel Press, February 2009.
- [104] Grsecurity. grsecurity. <http://www.grsecurity.net/>, accessed March 2010.
- [105] V. Haldar, D. Chandra, and M. Franz. Semantic Remote Attestation - A Virtual Machine directed approach to Trusted Computing. In *VM '04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 3–3, Berkeley, CA, USA, USENIX Association, May 2004.
- [106] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, May 2009.
- [107] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *HOTOS '05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, Berkeley, CA, USA, USENIX Association, June 2005.
- [108] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of  $\mu$ -kernel-based systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 66–77, New York, NY, USA, ACM, October 1997.
- [109] H. Härtig, J. Löser, F. Mehnert, L. Reuther, and M. P. and Alexander Warg. An I/O Architecture for Microkernel-Based Operating Systems. Technical report, Dresden University of Technology Department of Computer Science, 01062 Dresden, Germany, July 2003.
- [110] J. Heasman. Implementing and Detecting an ACPI BIOS Rootkit. Black Hat USA <http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf>, published January 2006.
- [111] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, July 2007.
- [112] G. Heiser, V. Uhlig, and J. LeVasseur. Are virtual-machine monitors microkernels done right? *ACM SIGOPS Operating Systems Review*, 40(1):95–99, January 2006.
- [113] C. Helmuth, A. Warg, and N. Feske. Mikro-SINA—Hands-on Experiences with the Nizza Security Architecture. In *D.A.CH Security 2005*, March 2005.

- [114] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *EW 11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 22, New York, NY, USA, ACM, September 2004.
- [115] J. Hoopes. *Virtualization for Security: Including Sandboxing, Disaster Recovery, High Availability, Forensic Analysis, and Honeypotting*. Syngress Publishing, January 2009.
- [116] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, Redmond, WA, USA, October 2005.
- [117] IBM. IBM System z. <http://www.ibm.com/systems/z/>, accessed January 2010.
- [118] IEEE P1363 working group. IEEE P1363: Standard Specifications For Public-Key Cryptography. <http://grouper.ieee.org/groups/1363/index.html>, accessed June 2010.
- [119] Intel. Intel Low Pin Count (LPC) Interface Specification. Technical Report Revision 1.1, Intel Corporation, August 2002.
- [120] Intel. Intel Virtualization Technology Specification for the IA-32 Intel Architecture. Technical Report C97063-002, Intel Corporation, April 2005.
- [121] Intel. Intel Trusted Execution Technology Measured Launched Environment Developer's Guide. Technical report, Intel Corporation, June 2008.
- [122] Intel. Architecture Guide: Intel Active Management Technology. <http://software.intel.com/en-us/articles/architecture-guide-intel-active-management-technology/>, accessed June 2009.
- [123] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. Technical Report 253665-033 US, Intel Corporation, December 2009.
- [124] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3a: System Programming Guide Part 1. Technical Report 253668-033 US, Intel Corporation, December 2009.

- [125] Intel. Intel's Advanced Encryption Standard (AES) Instructions Set. Technical Report Rev. 2.0, Intel Corporation, April 2009.
- [126] Intel. SINIT misconfiguration allows for Privilege Escalation. <http://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00021&languageid=en-fr>, accessed December 2009.
- [127] Intel. Intel Q45 Express Chipset Overview. <http://www.intel.com/products/desktop/chipsets/q45/q45-overview.htm>, accessed March 2010.
- [128] Intel. Trusted Boot. <http://tboot.sourceforge.net/>, accessed February 2009.
- [129] N. Itoi, W. A. Arbaugh, S. J. Pollack, and D. M. Reeves. Personal Secure Booting. In *ACISP '01: Proceedings of the 6th Australasian Conference on Information Security and Privacy*, pages 130–144, London, UK, Springer, January 2001.
- [130] T. Jaeger, R. Sailer, and Y. Sreenivasan. Managing the risk of covert information flows in virtual machine systems. In V. Lotz and B. M. Thuraisingham, editors, *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 81–90. ACM, June 2007.
- [131] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, March 2000.
- [132] P. A. Karger. Multi-Level Security Requirements for Hypervisors. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 267–275, Washington, DC, USA, IEEE Computer Society, December 2005.
- [133] P. A. Karger. Securing virtual machine monitors: what is needed? In *ASI-ACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 1–2, New York, NY, USA, ACM, March 2009.
- [134] P. A. Karger, U. Roger, and R. R. Schell. Multics security evaluation: Vulnerability analysis. Technical report, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, Bedford, MA, June 1974.
- [135] P. A. Karger and D. R. Safford. I/O for Virtual Machine Monitors: Security and Performance Issues. *IEEE Security and Privacy*, 6(5):16–23, September 2008.

- [136] P. A. Karger and R. R. Schell. Thirty Years Later: Lessons from the Multics Security Evaluation. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, page 119, Washington, DC, USA, IEEE Computer Society, September 2002.
- [137] P. A. Karger, M. E. Zurko, D. W. Benin, A. H. Mason, and C. E. Kahn. A VMM Security Kernel for the VAX Architecture. In *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 2–19. IEEE Computer Society, May 1990.
- [138] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A Retrospective on the VAX VMM Security Kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, November 1991.
- [139] B. Kauer. OSLO: improving the security of trusted computing. In *SS '07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–9, Berkeley, CA, USA, USENIX Association, August 2007.
- [140] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim. Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 11–20, New York, NY, USA, ACM, March 2008.
- [141] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 314–327, Washington, DC, USA, IEEE Computer Society, May 2006.
- [142] J. Kirch. Virtual Machine Security Guidelines. [http://www.cisecurity.org/tools2/vm/CIS\\_VM\\_Benchmark\\_v1.0.pdf](http://www.cisecurity.org/tools2/vm/CIS_VM_Benchmark_v1.0.pdf), accessed September 2007.
- [143] N. Kiyancilar. A Survey of Virtualization Techniques Focusing on Secure On-Demand Cluster Computing. *The Computing Research Repository (CoRR)*, November 2005.
- [144] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, New York, NY, USA, ACM, October 2009.

- [145] K. Kortchinsky. Honey-VMware patch. <http://honeynet.rstack.org/tools/vmpatch.c>, accessed November 2007.
- [146] KVM. Kernel-based Virtual Machine. <http://www.linux-kvm.org>, accessed January 2010.
- [147] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.
- [148] B. W. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.
- [149] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for Protecting Critical Secrets in Microprocessors. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 2–13, Washington, DC, USA, IEEE Computer Society, June 2005.
- [150] J. Lepreau, B. Ford, and M. Hibler. The persistent relevance of the local operating system to global applications. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 133–140, New York, NY, USA, ACM, September 1996.
- [151] A. H. Y. Leung. *Securing Mobile Ubiquitous Services using Trusted Computing*. PhD thesis, Royal Holloway University of London, July 2009.
- [152] A. H. Y. Leung, L. Chen, and C. J. Mitchell. On a Possible Privacy Flaw in Direct Anonymous Attestation (DAA). In *Trust '08: Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies*, pages 179–190, Berlin, Heidelberg, Springer, August 2008.
- [153] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI '04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, Berkeley, CA, USA, USENIX Association, December 2004.
- [154] D. Li, H. Jin, Y. Shao, and X. Liao. A High-efficient Inter-Domain Data Transferring System for Virtual Machines. In *ICUIMC '09: Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*, pages 385–390, New York, NY, USA, ACM, January 2009.

- [155] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, November 2000.
- [156] J. Liedtke. On micro-kernel construction. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 237–250, New York, NY, USA, ACM, December 1995.
- [157] T. A. Linden. Operating System Structures to Support Security and Reliable Software. *ACM Computing Surveys (CSUR)*, 8(4):409–445, December 1976.
- [158] S. B. Lipner. A comment on the confinement problem. In *SOSP '75: Proceedings of the fifth ACM symposium on Operating systems principles*, pages 192–196, New York, NY, USA, ACM, November 1975.
- [159] T. Liston and E. Skoudis. On the Cutting Edge: Thwarting Virtual Machine Detection. [http://handlers.sans.org/tliston/ThwartingVMDetection\\_Liston\\_Skoudis.pdf](http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf), published July 2006.
- [160] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314. National Security Agency, November 1998.
- [161] S. Loveland, E. M. Dow, F. LeFevre, D. Beyer, and P. F. Chan. Leveraging virtualization to optimize high-availability system configurations. *IBM Systems Journal*, 47(4):591–604, October 2008.
- [162] J. Lyle and A. Martin. On the Feasibility of Remote Attestation for Web Services. *IEEE International Conference on Computational Science and Engineering*, 3:283–288, August 2009.
- [163] S. E. Madnick and J. J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. *Proceedings of the workshop on virtual computer systems*, pages 210–224, March 1973.
- [164] MAME. MAME - Multiple Arcade Machine Emulator. <http://mamedev.org>, accessed December 2009.
- [165] W. Mao, A. Martin, H. Jin, and H. Zhang. Innovations for Grid Security from Trusted Computing. In *Fourteenth International Workshop on Security Protocols*, LNCS. Springer, March 2006.

- [166] A. Martin. The Ten Page Introduction to Trusted Computing. Technical Report CS-RR-08-11, Oxford University Computing Laboratory, Parks Road, Oxford, UK, November 2008.
- [167] J. M. McCune. *Reducing the Trusted Computing Base for Applications on Commodity Systems*. PhD thesis, School of Electrical and Computer Engineering Carnegie Mellon University, January 2009.
- [168] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. *ACM SIGOPS Operating Systems Review*, 42(4):315–328, April 2008.
- [169] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. How Low Can You Go? Recommendations for Hardware-Supported Minimal TCB Code Execution. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 14–25, New York, NY, USA, ACM, March 2008.
- [170] J. M. McCune, A. Perrig, A. Seshadri, and L. van Doorn. Turtles All The Way Down: Research Challenges in User-Based Attestation. In *HOTSEC '07: Proceedings of the 2nd USENIX workshop on Hot topics in security*, pages 1–5, Berkeley, CA, USA, USENIX Association, August 2007.
- [171] J. McDermott and M. Kang. An Open-Source High-Robustness Virtual Machine Monitor. In *The 22st Annual Computer Security Applications Conference*. The 22st Annual Computer Security Applications Conference, IEEE Computer Society, December 2006.
- [172] D. H. McKnight and N. L. Chervany. The meanings of trust. Technical report, University of Minnesota, November 1996.
- [173] M. McLoughlin. The QCOW Image Format. <http://www.gnome.org/~markmc/qcow-image-format.html>, accessed April 2008.
- [174] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, Berkeley, CA, USA, USENIX Association, June 2006.
- [175] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference*

- on *Virtual execution environments*, pages 13–23, New York, NY, USA, ACM, June 2005.
- [176] A. Menon, S. Schubert, and W. Zwaenepoel. TwinDrivers: Semi-Automatic Derivation of Fast and Safe Hypervisor Network Drivers from Guest OS Drivers. In *ASPLOS XIV: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 301–312, New York, NY, USA, ACM, March 2009.
- [177] R. C. Merkle. Protocols for Public Key Cryptosystems. *Security and Privacy*, pages 122–134, April 1980.
- [178] R. Meushaw and D. Simard. NetTop: Commercial Technology in High Assurance Applications. *National Security Agency Tech Trend Notes*, 9:3–10, Fall 2000.
- [179] R. A. Meyer and L. H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9(3):199–218, September 1970.
- [180] Microsoft. Microsoft Acquires Connectix Virtual Machine Technology. <http://www.microsoft.com/presspass/press/2003/feb03/02-19PartitionPR.msp>, published June 2003.
- [181] Microsoft. Virtualization: The Architectural Foundation for Dynamic IT. <http://www.microsoft.com/business/dsi/virtualization.msp>, published June 2006.
- [182] Microsoft. BitLocker Drive Encryption Technical Overview. <http://technet.microsoft.com/en-us/library/cc732774.aspx>, accessed April 2009.
- [183] J. S. Miller and S. Ragsdale. *The Common Language Infrastructure Annotated Standard*. Addison-Wesley, Boston, MA, USA, November 2003.
- [184] K. Miller and M. Pegah. Virtualization: virtually at the desktop. In *SIGUCCS '07: Proceedings of the 35th annual ACM SIGUCCS conference on User services*, pages 255–260, New York, NY, USA, ACM, October 2007.
- [185] S. C. Misra and V. C. Bhavsar. Relationships between selected software measures and latent bug-density: guidelines for improving quality. In *ICCSA '03: Proceedings of the 2003 international conference on Computational science and its applications*, pages 724–732, Berlin, Heidelberg, Springer, January 2003.
- [186] C. Mitchell, editor. *Trusted Computing (Professional Applications of Computing)*. IEEE Computer Society, Piscataway, NJ, USA, November 2005.

- [187] C. Mundie, P. de Vries, P. Haynes, and M. Corwine. Trustworthy Computing. White paper, Microsoft, October 2002.
- [188] D. G. Murray and S. Hand. Privilege separation made easy: trusting small libraries not big processes. In *EUROSEC '08: Proceedings of the 1st European Workshop on System Security*, pages 40–46, New York, NY, USA, ACM, March 2008.
- [189] D. G. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 151–160, New York, NY, USA, ACM, March 2008.
- [190] National Computer Security Center. Rainbow Series. <http://csrc.nist.gov/publications/secpubs/rainbow>, published July 1987.
- [191] National Institute of Standards and Technology. FIPS PUB 180-1: Secure Hash Standard. Technical report, NIST, April 1995.
- [192] National Institute of Standards and Technology. National Vulnerability Database. <http://nvd.nist.gov/home.cfm>, accessed August 2010.
- [193] P. G. Neumann, L. Robinson, K. N. Levitt, R. S. Boyer, and A. R. Saxena. A Provably Secure Operating System. Technical Report M79-225, Stanford Research Institute, Menlo Park, CA 94025, June 1975.
- [194] J. Oberheide, E. Cooke, and F. Jahanian. Exploiting Live Virtual Machine Migration. Black Hat USA <http://www.blackhat.com/presentations/bh-dc-08/Oberheide/Presentation/bh-dc-08-oberheide.pdf>, published February 2008.
- [195] A. A. Omella. Methods for Virtual Machine Detection. <http://www.s21sec.com/descargas/vmware-eng.pdf>, published June 2006.
- [196] R. W. O'Neill. Experience using a time-shared multi-programming system with dynamic address relocation hardware. In *AFIPS '67: Proceedings of the spring joint computer conference*, pages 611–621, New York, NY, USA, ACM, April 1967.
- [197] OpenTC. OpenTC Project. <http://opentc.net/>, accessed January 2009.
- [198] OpenVZ. OpenVZ OS-level virtualisation. <http://www.openvz.org/>, accessed February 2009.

- [199] T. Ormandy. An Emperical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. <http://taviso.decsystem.org/virtsec.pdf>, accessed April 2007.
- [200] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: a system for migrating computing environments. *ACM SIGOPS Operating Systems Review*, 36(SI):361–376, December 2002.
- [201] PCI-SIG. Single Root I/O Virtualization and Sharing Specification. Technical Report 1.0, PCI-SIG, September 2007.
- [202] J. Pearsall. *The Concise Oxford Dictionary*, volume 10. Oxford University Press, 1999.
- [203] S. Pearson, editor. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall PTR, Upper Saddle River, NJ, USA, August 2002.
- [204] M. Peinado, Y. Chen, P. England, and J. Manferdelli. NGSCB: A Trusted Open System. In *Proceedings of 9th Australasian Conference on Information Security and Privacy ACISP*, pages 86–97. Springer, June 2004.
- [205] M. Peter, H. Schild, A. Lackorzynski, and A. Warg. Virtual machines jailed: virtualization in systems with small trusted computing bases. In *VDTs '09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, pages 18–23, New York, NY, USA, ACM, March 2009.
- [206] Phoenix. HyperSpace. <http://www.hyperspace.com/>, accessed February 2009.
- [207] Phoenix. Phoenix Technologies Announces Major Milestone - First Customers for Phoenix FailSafe(TM) and HyperCore(TM). <http://investor.phoenix.com/releasedetail.cfm?ReleaseID=319319>, accessed May 2010.
- [208] D. Plaquin, S. Cabuk, C. I. Dalton, D. Kuhlmann, P. Grete, C. Weinhold, A. Böttcher, D. Murray, T. Hong, and M. Winandy. TPM Virtualisation Architecture document. Technical Report IST-027635, OpenTC, May 2009.
- [209] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, July 1974.
- [210] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick. Xen 3.0 and the Art of Virtualization. In *Proceedings of Linux Symposium 2005*. ACM, July 2005.

- [211] S. L. Presti. A Tree of Trust rooted in Extended Trusted Computing. In *Proceedings of the 2nd conference on Advances in Computer Security and Forensics (ACSF)*, Liverpool, UK, Liverpool John Moores University, July 2007.
- [212] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 4–4, Berkeley, CA, USA, USENIX Association, April 2007.
- [213] QEMU. QEMU - Open Source Processor Emulator. <http://www.qemu.org/>, accessed December 2009.
- [214] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
- [215] J. Robertson. Supergeek pulls off ‘near impossible’ crypto chip hack. [http://www.nzherald.co.nz/technology/news/article.cfm?c\\_id=5&objectid=10625082&pnum=0](http://www.nzherald.co.nz/technology/news/article.cfm?c_id=5&objectid=10625082&pnum=0), published February 2010.
- [216] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium’s ability to support a secure virtual machine monitor. In *SSYM '00: Proceedings of the 9th conference on USENIX Security Symposium*, page 10, Berkeley, CA, USA, USENIX Association, August 2000.
- [217] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and virtue. In *HOTOS '07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, USENIX Association, May 2007.
- [218] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *Computer*, 38(5):39–47, May 2005.
- [219] M. Rosenblum and M. Varadarajan. SimOS: A Fast Operating System Simulation Environment. Technical report, Stanford University, Stanford, CA, USA, July 1994.
- [220] RSA. PKCS #1 v2.1: RSA Cryptography Standard. Technical report, RSA Laboratories, June 2002.
- [221] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, July 2008.

- [222] J. Rutkowska. Subverting Vista Kernel For Fun And Profit. Black Hat USA <http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf>, published July 2006.
- [223] J. Rutkowska and R. Wojtczuk. Preventing and Detecting Xen Hypervisor Subversions. Black Hat USA <http://invisiblethingslab.com/resources/bh08/part2-full.pdf>, published August 2008.
- [224] J. Rutkowska and R. Wojtczuk. Qubes OS Architecture. <http://qubes-os.org/files/doc/arch-spec-0.3.pdf>, accessed September 2010.
- [225] A.-R. Sadeghi, M. Selhorst, C. Stübke, C. Wachsmann, and M. Winandy. TCG inside?: a note on TPM specification compliance. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 47–56, New York, NY, USA, ACM, November 2006.
- [226] A.-R. Sadeghi, C. Stübke, and M. Winandy. Property-Based TPM Virtualization. In *ISC '08: Proceedings of the 11th international conference on Information Security*, pages 1–16, Berlin, Heidelberg, Springer, September 2008.
- [227] R. Sahita, U. Warriar, and P. Dewan. Dynamic Software Application Protection. [http://blogs.intel.com/research/trusted%20dynamic%20launch-flyer-rls\\_pss001.pdf](http://blogs.intel.com/research/trusted%20dynamic%20launch-flyer-rls_pss001.pdf), April 2009.
- [228] R. Sailer, T. Jaeger, J. L. Griffin, S. Berger, L. van Doorn, R. Perez, and E. Valdez. Building a General-Purpose Secure Virtual Machine Monitor. 2005 RC23537 (W0502-132), IBM Research Division, February 2005.
- [229] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype - Secure Hypervisor. Technical Report RC23511, IBM Research Division, February 2005.
- [230] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *SSYM '04: Proceedings of the 13th conference on USENIX Security Symposium*, Berkeley, CA, USA, USENIX Association, August 2004.
- [231] P. H. Salus. *A quarter century of UNIX*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, June 1994.
- [232] J. Satran, L. Shalev, M. Ben-Yehuda, and Z. Machulsky. Scalable I/O - A Well-Architected Way to Do Scalable, Secure and Virtualized I/O. In *Workshop on I/O Virtualization*. USENIX Association, April 2008.

- [233] V. Scarlata, C. Rozas, M. Wiseman, D. Grawrock, and C. Vishik. TPM Virtualization: Building a General Framework. In N. Pohlmann and H. Reimer, editors, *Trusted Computing*, pages 43–56. Vieweg & Teubner, May 2008.
- [234] M. Schaefer and R. R. Schell. Toward an Understanding of Extensible Architectures for Evaluated Trusted Computer System Products. *IEEE Symposium on Security and Privacy*, page 41, May 1984.
- [235] R. R. Schell and M. Thompson. Platform Security: What is Lacking? Technical report, Elsevier Science, January 2000.
- [236] S. Schoen. EFF Comments on TCG Design, Implementation and Usage Principles. [http://www.eff.org/files/filenode/trustedcomputing/20041004\\_eff\\_comments\\_tcg\\_principles.pdf](http://www.eff.org/files/filenode/trustedcomputing/20041004_eff_comments_tcg_principles.pdf), published October 2004.
- [237] S. Schulz and A.-R. Sadeghi. Secure VPNs for Trusted Computing Environments. In *Trust '09: Proceedings of the 2nd International Conference on Trusted Computing*, pages 197–216, Berlin, Heidelberg, Springer, February 2009.
- [238] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350, New York, NY, USA, ACM, October 2007.
- [239] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, New York, NY, USA, ACM, December 1999.
- [240] H. Sharangpani and K. Arora. Itanium Processor Microarchitecture. *IEEE Micro*, 20(5):24–43, September 2000.
- [241] A. Shieh, D. Williams, E. G. Sirer, and F. B. Schneider. Nexus: a new operating system for trustworthy computing. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–9, New York, NY, USA, ACM, October 2005.
- [242] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: a thin hypervisor for enforcing I/O device security. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international*

- conference on Virtual execution environments*, pages 121–130, New York, NY, USA, ACM, March 2009.
- [243] W. R. Shockley and R. R. Schell. TCB subsets for incremental evaluation. In *Third AIAA Conference on Computer Security*, pages 131–139. IEEE Computer Society, December 1987.
- [244] H. Shrobe, T. Knight, and A. deHon. TIARA: Trust Management, Intrusion-tolerance, Accountability, and Reconstitution Architecture. Technical Report MIT-CSAIL-TR-2007-028, MIT, May 2007.
- [245] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 161–174, New York, NY, USA, ACM, April 2006.
- [246] J. E. Smith and R. Nair. The Architecture of Virtual Machines. *Computer*, 38:32–38, May 2005.
- [247] E. R. Sparks. A Security Assessment of Trusted Platform Modules. Technical Report TR2007-597, Department of Computer Science Dartmouth College, June 2007.
- [248] K. E. Stewart, J. W. Humphries, and T. R. Andel. Developing a virtualization platform for courses in networking, systems administration and cyber security education. In *SpringSim '09: Proceedings of the 2009 Spring Simulation Multiconference*, pages 1–7, San Diego, CA, USA, Society for Computer Simulation International, March 2009.
- [249] C. Strachey. Time Sharing in Large Fast Computers. In *International Conference on Information Processing*, volume paper B. 2. 1, pages 336–341. Proceedings of the International Conference on Information Processing, UNESCO, June 1959.
- [250] M. Strasser. A Software-based TPM Emulator for Linux. Master's thesis, Department of Computer Science Swiss Federal Institute of Technology Zurich, September 2004.
- [251] C. Stübli. PERSEUS trustworthy computing framework. <http://www.perseus-os.org/>, accessed February 2007.
- [252] F. Stumpf and C. Eckert. Enhancing Trusted Platform Modules with Hardware-Based Virtualization Techniques. In *SECURWARE '08: Proceed-*

- ings of the 2008 Second International Conference on Emerging Security Information, Systems and Technologies*, pages 1–9, Washington, DC, USA, IEEE Computer Society, September 2008.
- [253] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, USENIX Association, June 2001.
- [254] E. G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ICS ’03: Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171, New York, NY, USA, ACM, June 2003.
- [255] K. Suzaki, T. Yagi, K. Iijima, and N. A. Quynh. OS circular: internet client for reference. In *LISA ’07: Proceedings of the 21st conference on 21st Large Installation System Administration Conference*, pages 1–12, Berkeley, CA, USA, USENIX Association, November 2007.
- [256] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers. Nooks: an architecture for reliable device drivers. In *EW 10: Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107, New York, NY, USA, ACM, September 2002.
- [257] SWsoft. Top Ten Considerations For Choosing A Server Virtualization Technology. <http://www.swsoft.com/en/products/virtuozzo/lib/wp>, accessed February 2009.
- [258] A. S. Tanenbaum. *Modern Operating Systems*, volume 3. Prentice Hall Press, Upper Saddle River, NJ, USA, December 2007.
- [259] TCG. TCG PC Client Specific TPM Interface Specification (TIS). Technical Report Version 1.2 Revision 1.0, Trusted Computing Group, July 2005.
- [260] TCG. TCG Specification Architecture Overview. Technical Report Revision 1.3, Trusted Computing Group, March 2007.
- [261] TCG. TPM Main, Part 1 Design Principles. TCG Specification Version 1.2 Revision 103, Trusted Computing Group, Portland, OR, USA, July 2007.
- [262] TCG. TPM Main, Part 2 TPM Data Structures. TCG Specification Version 1.2 Revision 103, Trusted Computing Group, Portland, OR, USA, July 2007.

- [263] TCG. TPM Main, Part 3 Commands. TCG Specification Version 1.2 Revision 103, Trusted Computing Group, Portland, OR, USA, July 2007.
- [264] TCG. Trusted Computing Group. <https://www.trustedcomputinggroup.org>, accessed December 2008.
- [265] TCG. Summary Of Features Under Consideration For The Next Generation Of TPM. Technical report, Trusted Computing Group, published April 2009.
- [266] A. Tereshkin and R. Wojtczuk. Introducing Ring -3 Rootkits. Black Hat USA <http://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf>, published July 2009.
- [267] TrouSerS. TrouSerS - The open-source TCG Software Stack. <http://trousers.sourceforge.net>, accessed October 2008.
- [268] TrueCrypt Developers Association. TrueCrypt - Free open-source disk encryption software for Windows 7/Vista/XP, Mac OS X, and Linux. <http://www.truecrypt.org/>, accessed December 2009.
- [269] S. TÜRPE, A. Poller, J. Steffan, J.-P. Stotz, and J. Trukenmüller. Attacking the BitLocker Boot Process. In *Trust '09: Proceedings of the 2nd International Conference on Trusted Computing*, pages 183–196, Berlin, Heidelberg, Springer, February 2009.
- [270] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38:48–56, May 2005.
- [271] M. Varian. VM and the VM Community: Past, Present, and Future. Technical report, Office of Computing and Information Technology Princeton University, August 1997.
- [272] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., New York, NY, USA, January 1998.
- [273] Virtuatech. Virtuatech Simics. <http://www.virtutech.com>, accessed December 2009.
- [274] VMware. VMware Media Resource Center - Company Milestones. <http://www.vmware.com/company/mediaresource/milestones.html>, accessed December 2009.

- [275] VMware. Timekeeping in VMware Virtual Machines. [http://www.vmware.com/pdf/vmware\\_timekeeping.pdf](http://www.vmware.com/pdf/vmware_timekeeping.pdf), accessed May 2010.
- [276] VMware. Reducing Server Total Cost of Ownership with VMware Virtualization Software. <http://www.vmware.com/pdf/TCO.pdf>, published 2006.
- [277] VMware. Reduce Energy Costs and Go Green with VMware Green IT Solutions and FalconStor Software. <http://www.falconstor.com/?tk=3Z706C06E38B3461D584BED243BD55DA>, published September 2008.
- [278] C. A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, December 2002.
- [279] J. Wang, K.-L. Wright, and K. Gopalan. XenLoop: a transparent high performance inter-vm network loopback. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 109–118, New York, NY, USA, ACM, June 2008.
- [280] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. *IEEE Symposium on Security and Privacy*, pages 380–395, May 2010.
- [281] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 545–554, New York, NY, USA, ACM, November 2009.
- [282] A. Warfield and J. Chesterfield. Blktap Userspace Tools + Library. <http://lxr.xensource.com/lxr/source/tools/blktap/README>, published June 2006.
- [283] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical Report 02-02-01, University of Washington, February 2002.
- [284] J. Whitmore, A. Bensoussan, P. Green, D. Hunt, A. Kobziar, and J. Stern. Design for MULTICS Security Enhancements. Technical Report ESD-TR-74-176, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, Bedford, MA, December 1973.
- [285] P. Willmann, S. Rixner, and A. L. Cox. Protection Strategies for Direct Access to Virtualized I/O Devices. In *ATC '08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 15–28, Berkeley, CA, USA, USENIX Association, June 2008.

- [286] R. Wojtczuk. Subverting the Xen hypervisor. Black Hat USA [http://invisiblethingslab.com/bh08/papers/part1-subverting\\_xen.pdf](http://invisiblethingslab.com/bh08/papers/part1-subverting_xen.pdf), published August 2008.
- [287] R. Wojtczuk and J. Rutkowska. Attacking Intel Trusted Execution Technology. Black Hat USA <http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf>, published February 2009.
- [288] R. Wojtczuk and J. Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning. [http://invisiblethingslab.com/resources/misc09/smm\\_cache\\_fun.pdf](http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf), published March 2009.
- [289] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Another Way to Circumvent Intel Trusted Execution Technology. <http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf>, published December 2009.
- [290] R. Wojtczuk and A. Tereshkin. Attacking Intel BIOS. Black Hat USA <http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf>, published July 2009.
- [291] Xandros. Presto. <http://www.prestomypc.com/>, accessed January 2009.
- [292] C. J. Young. Extended architecture and Hypervisor performance. In *Proceedings of the workshop on virtual computer systems*, pages 177–183, New York, NY, USA, ACM, March 1973.
- [293] Y. Yu, F. Guo, S. Nanda, L.-c. Lam, and T.-c. Chiueh. A feather-weight virtual machine for windows applications. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 24–34, New York, NY, USA, ACM, June 2006.
- [294] M. Yung. Trusted Computing Platforms: The Good, the Bad, and the Ugly. In R. N. Wright, editor, *Financial Cryptography*, volume 2742 of *Lecture Notes in Computer Science*, pages 250–254. Springer, February 2003.
- [295] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. XenSocket: A High-Throughput Interdomain Transport for Virtual Machines. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 184–203, New York, NY, USA, Springer, November 2007.
- [296] Q. Zhong and N. Edwards. Security Risk Control of COTS-based Applications. Technical Report HPL-97-108, Hewlett-Packard Laboratories, September 1997.

[297] D. A. D. Zovi. Hardware Virtualization Rootkits. Black Hat USA <http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf>, August published 2006.