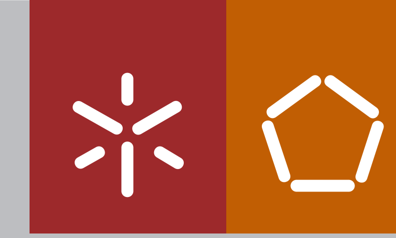


Universidade do Minho
Escola de Engenharia

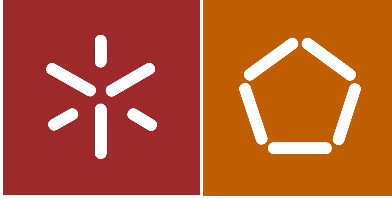
Sandro Emanuel Salgado Pinto

Secure and Safe Virtualization-based Framework for Embedded Systems Development

março de 2017

Sandro Emanuel Salgado Pinto
Secure and Safe Virtualization-based Framework for Embedded Systems Development
UMinho | 2017

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR



Universidade do Minho
Escola de Engenharia

Sandro Emanuel Salgado Pinto

Secure and Safe Virtualization-based Framework for Embedded Systems Development

Tese de Doutoramento
Programa Doutoral em
Engenharia Electrónica e de Computadores (PDEEC)

Trabalho efetuado sob a orientação do
Professor Doutor Jorge Miguel Nunes dos Santos Cabral
Professor Doutor Adriano José Conceição Tavares

março de 2017

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, ________________________________

Full name: Sandro Emanuel Salgado Pinto

Signature: Sandro Emanuel Salgado Pinto ______________________

Acknowledgments

This journey has been long. Sometimes exciting and rewarding, sometimes difficult and stressful. However, as everything in life, it is this multitude of feelings that makes the process of discovering into an unknown path worthwhile. There are many people that I want to thank. Without their time, expertise, patience and support, I am pretty sure that I could not have succeeded.

To Dr. Adriano Tavares, my mentor, advisor, long-time professor and friend, thanks for the meetings, the calls, the advices and the reviews, and, above all, thanks for always caring about me. I know that I have a friend for life. To my advisor Dr. Jorge Cabral thanks for guiding me all these years, and thanks for always understanding me even on those days where I was mentally sick.

Thanks to my PhD and lab colleagues, who helped me in many ways. Tiago A. Gomes, Filipe Salgado, Filipe Alves, Nuno Cardoso, and Paulo Garcia: thanks for all the wonderful time we spent together, and above all, thanks for explaining me what I was not able to understand.

Special thanks to Tiago M. Gomes and Jorge Pereira: you definitely were the biggest supporters during this journey. Thanks for helping me in all possible ways, and, above all, for sharing all those thousands kilometers we traveled together. From Canada to Germany, from Luxembourg to Turkey, and from South Korea, Japan and China to Thailand, let me just say: "Obrigado"!

Thanks to the master students under my supervision, who were always open-minded to consider and evaluate even my out-of-the-box ideas. Daniel Oliveira, Carlos Fernandes, Diogo Lima, Raphael Gonçalves, André Oliveira, Eduardo Mendes, João Silva, and José Martins: thanks for having the patience to listen me, for following my advices, for answering my bazillion questions and for reviewing my code.

Thanks to Fundação para a Ciência e Tecnologia for financing my Ph.D (grant

SFRH/BD/91530/2012). Thanks to the ESRG group of the University of Minho for being such a fantastic place to work. Thanks to the MES group of the Asian Institute of Technology, Thailand, for hosting me twice. Special thanks to Professor Mongkol Ekpanyapong for making them both possible: "*Din daen haeng roy yim*". Thanks to Dr. Sergio Montenegro and the AIT group of the University of Würzburg for hosting me in Germany: "*Dankeschön*".

Thanks to family and friends for all the rest. Naming all of you would require twice as many pages as the ones already written. Thanks to my close group of friends for always being there for me: you definitely know who you are.

Special thanks to my mom, dad and grandmother. Thanks for being the best parents in the world, and thanks for supporting me unconditionally. I would need billions of words to truly express my gratitude, but if I had to choose one I would just say: "Amo-vos"!

Finally, to the love of my life, Bárbara, sorry and thanks. Sorry for not being always there for you, especially on those months of oceanic distance. Sorry for every single tear that you dropped, and sorry for not hug and cuddle you when you most need. Sorry for not always understand you, and thanks for, apart all, you still support and love me so hard: "És a mulher da minha vida"!

"A todos, Muito Obrigado"!

Sandro Pinto

Guimarães, March 7th, 2017.

"Genius is 1% inspiration, 99% perspiration"

- Thomas Edison

Abstract

The Internet of Things (IoT) is here. Billions of smart, connected devices are proliferating at rapid pace in our key infrastructures, generating, processing and exchanging vast amounts of security-critical and privacy-sensitive data. This strong connectivity of IoT environments demands for a holistic, end-to-end security approach, addressing security and privacy risks across different abstraction levels: device, communications, cloud, and lifecycle managment.

Security at the device level is being misconstrued as the addition of features in a late stage of the system development. Several software-based approaches such as microkernels, and virtualization have been used, but it is proven, per se, they fail in providing the desired security level. As a step towards the correct operation of these devices, it is imperative to extend them with new security-oriented technologies which guarantee security from the outset.

This thesis aims to conceive and design a novel security and safety architecture for virtualized systems by 1) evaluating which technologies are key enablers for scalable and secure virtualization, 2) designing and implementing a fully-featured virtualization environment providing hardware isolation 3) investigating which "hard entities" can extend virtualization to guarantee the security requirements dictated by confidentiality, integrity, and availability, and 4) simplifying system configurability and integration through a design ecosystem supported by a domain-specific language.

The developed artefacts demonstrate: 1) why ARM TrustZone is nowadays a reference technology for security, 2) how TrustZone can be adequately exploited for virtualization in different use-cases, 3) why the secure boot process, trusted execution environment and other hardware trust anchors are essential to establish and guarantee a complete root and chain of trust, and 4) how a domain-specific language enables easy design, integration and customization of a secure virtualized system assisted by the above mentioned building blocks.

Resumo

Vivemos na era da Internet das Coisas (IoT). Biliões de dispositivos inteligentes começam a proliferar nas nossas infraestruturas chave, levando ao processamento de avolumadas quantidades de dados privados e sensíveis. Esta forte conectividade inerente ao conceito IoT necessita de uma abordagem holística, em que os riscos de privacidade e segurança são abordados nas diferentes camadas de abstração: dispositivo, comunicações, nuvem e ciclo de vida.

A segurança ao nível dos dispositivos tem sido erradamente assegurada pela inclusão de funcionalidades numa fase tardia do desenvolvimento. Têm sido utilizadas diversas abordagens de software, incluindo a virtualização, mas está provado que estas não conseguem garantir o nível de segurança desejado. De forma a garantir a correta operação dos dispositivos, é fundamental complementar os mesmos com novas tecnologias que promovem a segurança desde os primeiros estágios de desenvolvimento.

Esta tese propõe, assim, o desenvolvimento de uma solução arquitetural inovadora para sistemas virtualizados seguros, contemplando 1) a avaliação de tecnologias chave que promovam tal realização, 2) a implementação de uma solução de virtualização garantindo isolamento por hardware, 3) a identificação de componentes que integrados permitirão complementar a virtualização para garantir os requisitos de segurança, e 4) a simplificação do processo de configuração e integração da solução através de um ecossistema suportado por uma linguagem de domínio específico.

Os artefactos desenvolvidos demonstram: 1) o porquê da tecnologia ARM TrustZone ser uma tecnologia de referência para a segurança, 2) a efetividade desta tecnologia quando utilizada em diferentes domínios, 3) o porquê do processo seguro de inicialização, juntamente com um ambiente de execução seguro e outros componentes de hardware, serem essenciais para estabelecer uma cadeia de confiança, e 4) a viabilidade em utilizar uma linguagem de um domínio específico para configurar e integrar um ambiente virtualizado suportado pelos artefactos supramencionados.

Contents

Acknowledgments **iii**

Abstract **vii**

Resumo **ix**

Acronyms **xix**

1 Introduction **3**

1.1 Motivation 5
1.2 Problem Statement 7
1.3 Scope 10
1.4 Research Questions and Methodology 12
1.5 State-of-the-Art 13
1.5.1 Software-based Isolation and Virtualization 14
1.5.2 Hardware-based Virtualization 15
1.5.3 Secure Processor Architectures 17
1.5.4 Hardware Security Modules 21
1.5.5 Secure Silicon Against Insider Attacks 23
1.6 Conclusions 26
1.7 Thesis Structure 27
1.8 List of Publications 28
1.9 Summary 31
References 31

2 Research Platform and Tools **43**

2.1 Platform Requirements 45
2.2 ARM Architecture Overview 47
2.2.1 ARMv7-A Architecture 48

2.3 TrustZone: The ARM Security Extensions 53
2.3.1 TrustZone: Hardware Component 54
2.3.2 TrustZone: Software Component 55
2.4 The Zynq Device 56
2.4.1 Security 58
2.4.2 Zynq-7000 Family 60
2.4.3 Development Boards 61
2.5 Operating System Stacks 63
2.5.1 Real-Time Operating Systems 63
2.5.2 General-Purpose Operating Systems 66
2.6 Benchmarks 67
2.6.1 Thread-Metric 67
2.6.2 LMBench 67
2.7 Summary 68
References 69

3 LTZVisor: TrustZone is the Key 73
3.1 LTZVisor: Objectives 75
3.2 LTZVisor: Design 76
3.2.1 Design Principles 76
3.2.2 General Architecture 77
3.3 LTZVisor: Implementation 78
3.3.1 Virtual CPU 78
3.3.2 Scheduler 79
3.3.3 Memory Partition 80
3.3.4 MMU and Cache Management 81
3.3.5 Device Partition 82
3.3.6 Interrupt Management 82
3.3.7 Time Management 84
3.4 LTZVisor: Execution Flow 84
3.5 Evaluation 86
3.5.1 Memory Footprint 86
3.5.2 Performance 87
3.6 Discussion 95
3.7 Summary 98
References 99

4 TZVisor: Beyond TrustZone Support 105

4.1 TZVisor: Objectives . . . 107
4.2 TZVisor: General Architecture . . . 107
4.3 TZVisor: Implementation . . . 109
4.3.1 Guest Management . . . 109
4.3.2 Scheduler . . . 111
4.3.3 Memory Partition . . . 112
4.3.4 MMU and Cache Management . . . 114
4.3.5 Device Management . . . 116
4.3.6 Interrupt Management . . . 118
4.3.7 Time Management . . . 120
4.3.8 Inter-VM Communication . . . 121
4.4 Aerospace Safety-Critical Use Case . . . 123
4.4.1 Implementation . . . 124
4.4.2 Execution Flow . . . 126
4.4.3 Evaluation . . . 127
4.5 Industrial Mixed-Critical Use Case . . . 134
4.5.1 Implementation . . . 135
4.5.2 Execution Flow . . . 136
4.5.3 Evaluation . . . 138
4.6 Discussion . . . 144
4.7 Summary . . . 147
References . . . 147

5 T-TZVisor: No Safety without Security 153
5.1 T-TZVisor: Objectives . . . 155
5.2 T-TZVisor: General Architecture . . . 155
5.3 T-TZVisor: Implementation . . . 157
5.3.1 Trusted Storage . . . 157
5.3.2 Trusted Boot . . . 160
5.3.3 Trusted TZvisor . . . 163
5.3.4 Trusted RTOS . . . 164
5.3.5 TrustZone-aware GPOS . . . 166
5.4 Evaluation . . . 169
5.4.1 Real-time . . . 170
5.4.2 Security Analysis . . . 171
5.4.3 Experimental Validation . . . 173
5.5 Discussion . . . 176
5.6 Summary . . . 179

References . . . 180

6 Design Automation: It's Not Just about Technology 185

6.1 Motivation . . . 187
6.2 Domain-Specific Languages . . . 188
6.2.1 Related Work . . . 189
6.2.2 The Domain-Specific Development Process . . . 190
6.2.3 The Domain-Specific Development Benefits . . . 191
6.2.4 DSL Implementation Approaches . . . 192
6.3 EL: The Elaboration Language and Workflow . . . 193
6.3.1 EL Workflow . . . 196
6.4 vEL: a VMM-assisted DSL . . . 200
6.5 Discussion . . . 204
6.6 Summary . . . 205
References . . . 206

7 Conclusion and Future Work 211

7.1 Summary and Conclusions . . . 212
7.2 Contributions . . . 214
7.3 Limitations . . . 215
7.4 Research Roadmap . . . 216

List of Figures

1.1 Generic end-to-end IoT model from things to network to cloud 5
1.2 Generic IoT device and service architectures 6
1.3 Beecham research's IoT security threat map 8
1.4 Evolution of the hypervisor landscape 15
1.5 Levels of security concerns for designer and countermeasures for them 24
1.6 Cybersecurity co-processor for runtime trojan and side-channel detection 25

2.1 ARM TrustZone 53
2.2 Zynq-7000 SoC overview 57
2.3 Zynq-based platforms 61

3.1 LTZVisor: general architecture 77
3.2 LTZVisor: memory configuration 81
3.3 LTZVisor: interrupt management 83
3.4 LTZVisor: execution flow 86
3.5 LTZVisor: Thread-Metric benchmarks 90
3.6 LTZVisor: arithmetic operations latency benchmark 92
3.7 LTZVisor: memory bandwidth benchmark 93
3.8 LTZVisor: system calls latency benchmark 94

4.1 TZVisor: general architecture 108
4.2 TZVisor: guest management 110
4.3 TZVisor: system memory map 112
4.4 TZVisor: inter-VM communication 122
4.5 TZVisor for Aerospace: system architecture 124
4.6 TZVisor for Aerospace: execution flow 127
4.7 TZVisor for Aerospace: Thread-Metric benchmarks 131
4.8 TZVisor for Aerospace: guest-switching rate vs performance 132

4.9 TZVisor for ICS: system architecture 135
4.10 TZVisor for ICS: execution flow 137
4.11 TZVisor for ICS: arithmetic operations latency benchmark 142
4.12 TZVisor for ICS: memory bandwidth benchmark 143
4.13 TZVisor for ICS: guest-switching rate vs performance 144

5.1 T-TZVisor: general architecture 156
5.2 T-TZVisor: secure boot image format 161
5.3 T-TZVisor: secure boot process 162
5.4 T-TZVisor: Thread-Metric benchmark 170
5.5 T-TZVisor: establishing a secure session 174
5.6 T-TZVisor: performing a secure operation 175

6.1 Gartner' emerging technology hype cycle (2015) 187
6.2 Domain-specific development 191
6.3 Multiple DSL integration 193
6.4 EL 2-stage design workflow 196
6.5 TZVisor framework as an SCA composite 201
6.6 Ontology-driven integration DSL 205

List of Tables

1.1 Gap analysis among existing TCBs 27

2.1 ARMv7-A processor modes 49
2.2 ARMv7-A core registers 51
2.3 CP15 register summary 53
2.4 Zynq-7000 TrustZone security summary 60
2.5 Zynq-7000 family members 60

3.1 LTZVisor: memory footprint (bytes) 87
3.2 LTZVisor: performance statistics 89

4.1 TZVisor for Aerospace: health monitoring events and actions 126
4.2 TZVisor for Aerospace: memory footprint (bytes) 128
4.3 TZVisor for Aerospace: performance statistics 130
4.4 TZVisor for ICS: memory footprint (bytes) 138
4.5 TZVisor for ICS: performance statistics 141

5.1 BootROM header summary 160
5.2 TrustZone API: main data structures 166
5.3 TrustZone API: control functions 167
5.4 TrustZone API: encoder and decoder functions 168

6.1 Available EL's keywords 199
6.2 Implemented EL files 202

Acronyms

AES Advanced Encryption Standard

AML Automation Markup Language

AMP asymmetric multiprocessing

APB Advanced Peripheral Bus

API application programming interface

APU application processing unit

AXI Advanced eXtensible Interface

BBRAM battery backup random access memory

CAIC control, availability, integrity, and confidentiality

CIA confidentiality, integrity, and availability

CoT chain of trust

COTS commercial off-the-shelf

CPU central processing unit

DBT dynamic binary translation

DMA direct memory access

DoS denial-of-service

DSD domain-specific development

DSL domain-specific language

DSP digital signal processor

FIQ Fast Interrupt Request

FPGA field-programmable gate array

FPU floating-point unit

FSBL First Stage Boot Loader

GIC Generic Interrupt Controller

GPIO general-purpose input/output

GPOS general-purpose operating system

GPU graphics processing unit

HSM hardware security module

HMAC hash-based message authentication code

IC integrated circuit

ICS industrial control systems

iIoT industrial Internet of Things

IMA integrated modular avionics

IoT Internet of Things

IP intellectual property

IT information technology

IRQ Interrupt Request

JTAG Joint Test Action Group

MCU microcontroller

MMU memory management unit

MPU memory protection unit

NSCApps non-secure client applications

NVM non-volatile memory

OCM on-chip memory

OS operating system

OT operational technology

PL programmable logic

PMU Performance Monitor Unit

PPK primary public key

PS processing system

PSP Platform Security Processor

PUF physical unclonable function

QSPI Quad Serial Peripheral Interface

RA Reference Architecture

RAM random-access memory

REE rich execution environment

RISC reduced instruction set computer

ROM read-only memory

RoT root of trust

RTL register-transfer level

RTOS real-time operating system

SCA Service Component Architecture

SCU Snoop Control Unit

SD Secure Digital

SDIO Secure Digital Input Output

SDR software-defined radio

SGI software generated interrupt

SGX Software Guard Extensions

SMP symmetric multiprocessing

SOA service-oriented architecture

SoC system-on-chip

STP space and time partition

SWaP-C size, weight, power and cost

TCB trusted computing base

TEE trusted execution environment

TLB translation lookaside buffer

TPM Trusted Platform Module

TTC Triple Timer Counter

TZAPI TrustZone API

TZASC TrustZone Address Space Controller

TZMA TrustZone Memory Adapter

TZPC TrustZone Protection Controller

UART universal asynchronous receiver/transmitter

USB Universal Serial Bus

VE Virtualization Extensions

VM virtual machine

VMCB virtual machine control block

VMM virtual machine monitor

VT Virtualization Technology

XML Extensible Markup Language

"The Internet of Things has the potential to change the world,
just as the Internet did. Maybe even more so."
- Kevin Ashton

1 Introduction

The world is undergoing an unprecedented technological transformation, evolving from isolated systems to ubiquitous Internet-enabled 'things' capable of generating and exchanging vast amounts of valuable data. This novel paradigm, commonly referred as the Internet of Things (IoT), is a new reality that is enriching our everyday life, increasing business productivity, and improving government efficiency. In the IoT era, daily usage objects are becoming smarter, and start to play a key role in surrounding infrastructures. From a simple smart street lamp to a complex smart city, or from a simple industrial controller to a complex smart factory, this flourish of interconnected devices promise to drive a plethora of applications with technological, economic, and social prospects.

In this Chapter, I present an introductory vision about this thesis. I begin by explaining the motivational reasons for this work, while formalizing the problem statement and identifying the scope of this thesis. I formalize the research questions and explain the proposed methodology. I present a complete state-of-the-art identifying and describing related work that individually or partially addresses safety, security and real-time. Lastly, I present a gap analysis among the most important recent research, while comparing with my envisioned solution.

This Chapter is organized as follows: Section 1.1 motivates this thesis, while Section 1.2 formalizes the problem statement. Section 1.3 describes the scope of this work, and Section 1.4 presents the research questions and the proposed methodology to answer those questions. Section 1.5 shows the state-of-the-art in embedded virtualization and endpoint security. Section 1.6 describes the solution envisioned by me, and, finally, Sections 1.7 and 1.8 present the structure of this document, as well as a list of publications that directly or indirectly contributed for this thesis, respectively.

1.1 Motivation

The world is embracing an unprecedented technological trend for connecting the unconnected. The way key technologies such as embedded systems, cloud computing, system-on-chip (SoC), wired and wireless networking, and sensor and actuator have been evolved for the last decade is pushing industry and academia to shift from isolated systems to ubiquitous Internet-enabled 'things' [1.1, 1.2]. This novel paradigm, commonly referred as the Internet of Things, can be described as a global network infrastructure which enables people, process, data, and things to be connected at anytime, anyplace, with anything and anyone, ideally using any service. The IoT Era is a new reality that is enriching our everyday life, increasing business productivity, and improving government efficiency [1.2, 1.3]. Daily usage objects are becoming smarter, and start to play a key role in surrounding infrastructures.

From the architectural perspective, IoT has been emerged as endpoint and service ecosystems in which several and diverse endpoint devices are connected to the internet [1.4, 1.5, 1.6] via network infrastructure (Figure 1.1). Such modular or tiered architecture is proposed mainly to accommodate the wide range of system scope, topologies, and geography. The endpoint ecosystem is populated by several devices differentiated by their functionalities and complexities used to sense and actuate the physical world around them. It pushes the collected data to the service ecosystem and receive instruction back in response. The service ecosystem is typically based on a layered Reference Architecture (RA) and it consists of set of services, platforms, protocol and several other technologies required to gather data from endpoints and

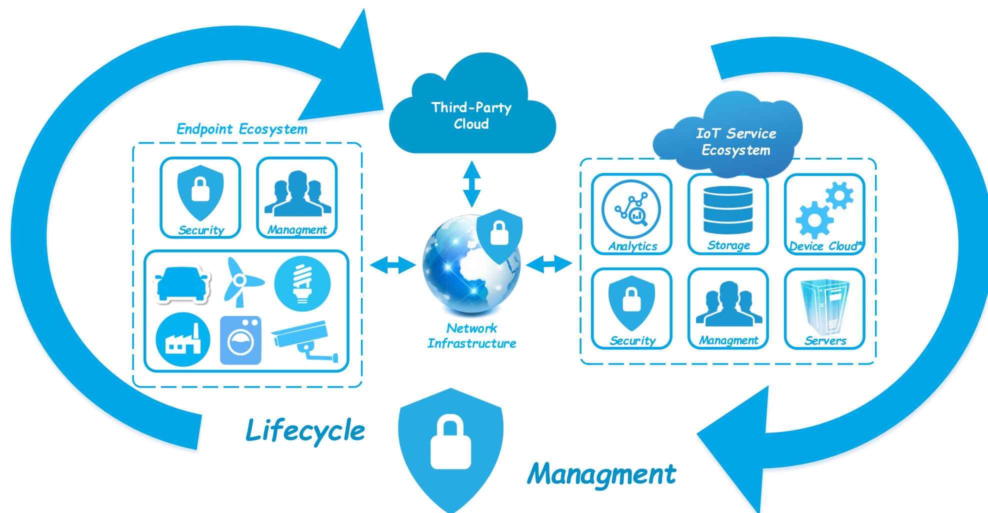


Figure 1.1: Generic end-to-end IoT model from things to network to cloud

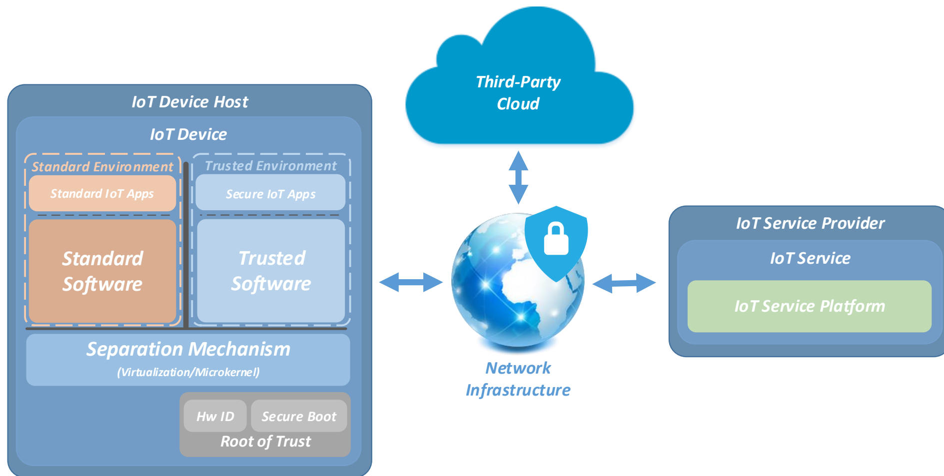


Figure 1.2: Generic IoT device and service architectures

store them for later processing within its server environment [1.5]. IoT computing will be embedded into almost all IoT device host (Figure 1.2) such as connected car, connected home, connected factory [1.7] or even in a complex connected city [1.8].

Despite the above generic IoT model illustrated by Figure 1.1, there are several definitions of IoT such as: (1) the next generation of the Internet, (2) an evolution of the networking industry enabling everything to become interconnected, with IoT focused on machine communications, (3) the bridging of operational technology (OT) and information technology (IT) and (4) pervasive control of highly distributed actuators [1.9], just to name a few.

To foster the realization of IoT paradigm, several standards/alliances, secure RAs [1.4, 1.5, 1.6, 1.9, 1.10, 1.11, 1.12, 1.13], and design guidelines [1.14, 1.15, 1.16, 1.17, 1.18, 1.19] have been proposed and discussed. For instance:

1. Intel IoT Solutions Alliance, Industrial Internet Alliance, OneM2M, AllSeen Alliance, Internet of Things Consortium, IPSO Alliance, Alliance for Internet of Things Innovation, and LORATM have been formed to reshape the fragmented IoT industry;

2. Intel IoT Platform [1.6], IoT-ARM [1.10], oneM2M framework [1.11], Cisco IoT RA [1.12], Oracle IoT RA [1.13] or WSO2 [1.20] are among some proposed reference architectures;

3. Guidelines [1.14, 1.15, 1.16, 1.17, 1.18, 1.19] to leverage a secure design of IoT endpoint devices.

In so doing, several features and design principles of IoT have been identified in

[1.21, 1.22] and among them, the following are enumerated:

1. **Big Analog Data**: "Really Real-Time" as IoT data are acquired in a real-time nature that is unfamiliar to traditional IT;

2. **Proximity**: Things can be categorized into market segments based on distance from the human heart;

3. **Shift-Left Computing**: Sensor- and control-enabled analytics will be migrate left along architectural tiers, offering a "Spectrum of Value", related to the place in the data flow;

4. **'WiTricity'**: IoT business models will need to be completely wireless, or better yet permanently wireless;

5. **Platforming**: Much like date, time, and temperature a new generation of "basic" features will need to be ubiquitous;

6. **Spreading of Hardware-Assisted Acceleration**: Hardware and software should be engineered, tested, and supported together to better achieve end-to-end security, reduce security impact on performance, real-time requirement, low latency and high throughput communication channels and power-efficiency.

Furthermore, to address the IoT vulnerabilities, the **Spreading of IoT Security** emerged as another principle of IoT system realization [1.15, 1.18, 1.23, 1.24, 1.25, 1.26]. To better address and increase security intelligence, the security should spans all IoT technology stack layers, dictating a holistic design approach including each IoT-tier from endpoint ecosystem to the service ecosystem for the lifetime of an IoT project [1.24, 1.25, 1.27].

1.2 Problem Statement

The beginning of the IoT explosive cycle driven by its forecasted enormous endpoint ecosystem footprint, will create a huge spectrum of business opportunities but also enormous security challenges, as the attack surface will be radically increased [1.28, 1.24, 1.25]. Figure 1.3 presents the Beecham's IoT security threat map[1] with the expanded threat surface and essential defenses. It basically suggests three possible vulnerability analyses:

[1] http://www.beechamresearch.com/files/Product-image-43.jpg

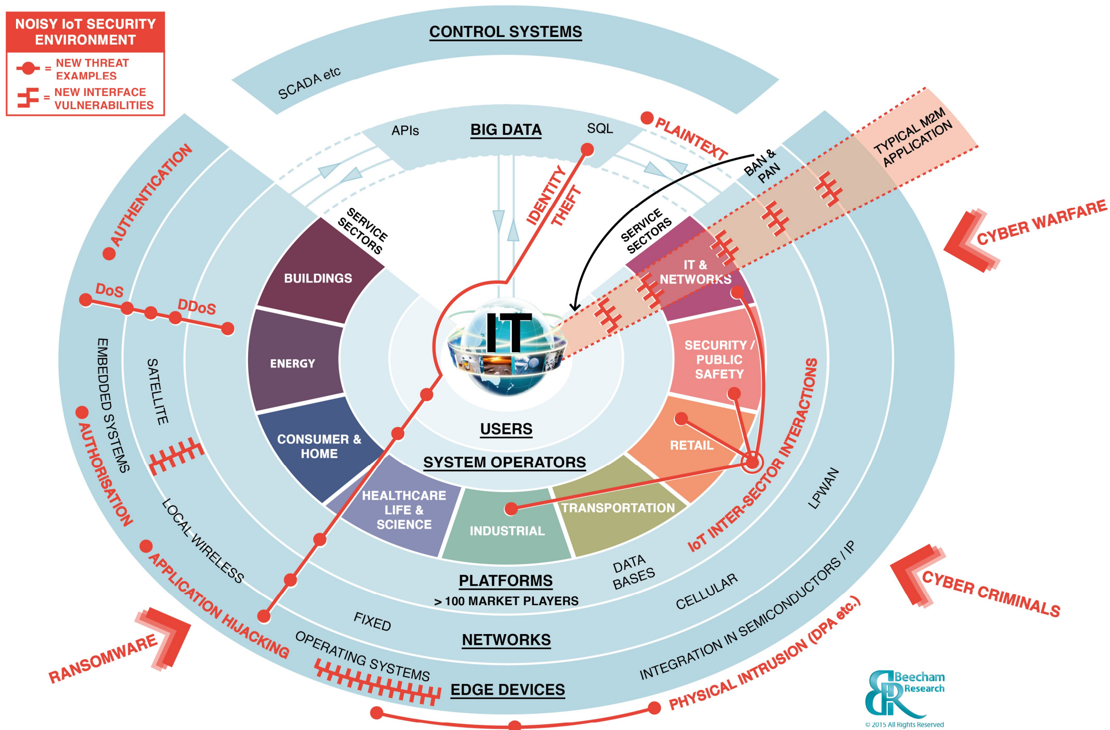


Figure 1.3: Beecham research's IoT security threat map

1. **IoT internal threat** including: the hijacking of IoT device application (Figure 1.2); increasing accessibility through communication enabling denial-of-service (DoS) attack; the current need for searchable databases in the new Big data arena to be stored in unprotected plaintext; complexities of IoT systems targeting multiple sector verticals; and the proliferation of internal interfaces and their introduction of weakness in advanced IoT solutions;

2. **IoT external threat** including: cyber warfare, cyber criminality, the use of ransomware, identify theft and physical intrusion attacks on endpoint devices;

3. **Other areas of IoT threat** including: needs for robust authentication, authorization and confidentiality; features and interactions between multiple networks used together in IoT; complexities of combining service sector optimized capabilities of different service enablement platforms; and the implementation and defenses of the endpoint device operating systems (OSes), chip integration and the associated root of trust (RoT).

Hence, the point is how the above threat map can be used to identify the right approach to protect the 'IoT system to be designed and then deployed' with the right-sized and affordable security, knowing such a large threat surface. From the previous three vulnerability analyses [1.27], the associated IoT security challenges

will translate into:

1. **Complexity** due to differing value, threats and budgets in multiple connected verticals, networks and endpoint device technologies that need to be secured. Contrary to business perspective based on using one type of device, one type of connectivity, one protocol and one set of data, IoT perspective is based on using several different endpoint devices, different protocols, different types of connectivity and different sources of data. To address this challenge, extremely flexible permissions and visibility capability will be required;

2. **Risk Mitigation** concerning over cyber-attacks finding devices on the Internet and it will dictate device-initiated communication through single method of IoT system connection;

3. **Secure Update** concerning with the inability to easily provide security updates to endpoint devices and it will require infrastructure to distribute security patches on an as-needed basis.

Regarding integrity and privacy, it is been recognized plenty of similarities in security requirements among the three verticals presented in the IoT model (Figure 1.1). However, verticals such as endpoint ecosystem, service ecosystem and networking, come with their own unique vulnerabilities and threats, and so, requiring their own specific security approaches. Hence, to leverage uniformed right-sized and affordable security across the three verticals, an IoT security-centric framework should be designed following principles such as: (1) security through common guidelines and context-awareness and (2) security through best practice-based design. In [1.27] is suggested the following three-step approach to protect the whole IoT value chain, from endpoint ecosystem to the service ecosystem, for the lifetime of an IoT project: (1) conducting an end-to-end risk assessment of the IoT infrastructure to be deployed as there is no one-size-fits-all recipe for security, (2) applying security and privacy by design to protect what matters, where it matters and when it matters most, and (3) facilitating long-term lifecycle management to ensure that the security protection can evolve at the same pace as new threats.

The multiplicity of endpoint devices in the IoT explosive cycle demands for a secure endpoint ecosystem to complement in-transit and in-cloud data with in-device integrity and privacy. Contrary to service ecosystem and communication infrastructure verticals where security problems and solutions have been consolidated for years, the security solution at the endpoint ecosystem is still very immature, mainly due to the novelty of endpoint devices and the related pressure for low cost, which

most of the time prevents the inclusion of security mechanisms [1.27]. Therefore, endpoint ecosystem has been seen as a security loophole in the whole IoT ecosystem and some works have been suggesting guidelines, technologies, and, architectural requirements and features [1.5, 1.14, 1.19, 1.29, 1.30, 1.31] for designing secure IoT endpoint devices. Firstly, a change in designer' mindset is recommended to view endpoint devices as connected devices instead of stand-alone while promoting easy-to-use security configuration. Secondly, a holistic security-by-design approach from the lowest levels of system-on-chip through to the IoT device application should be adopted to close the highly fragmented attack zones across the variety of application components, operating systems, virtualization components and hardware that compose today' endpoint devices.

However, to accommodate different classes of IoT endpoint devices (e.g., lightweight endpoint, complex endpoint and gateway [1.5]) while scaling up security without risking safety and real-time properties, it is my belief that virtualization technology should also scale to support strong partitioning, mainly when it is deployed on industrial IoT (iIoT) endpoint devices. Furthermore, since both IoT and iIoT endpoint devices have been addressed, it is worthwhile to emphasize the differences in security and safety priorities in both IT and OT environments. Such difference is mainly due to what is being secured in both world: IT has main focus on digital information protection, while OT focus on people and physical asset protection [1.32]. OT and IT worlds are benchmarked by CAIC and CIA respectively. CIA is ranked in order of priority and it stands for confidentiality, integrity, and availability. CAIC stands for control, availability, integrity, and confidentiality that is also ranked in order of priority. The added property addresses how to control processes and change of states in a safe and secure way.

1.3 Scope

Security at endpoint devices has becoming much more crucial with the distribution of data and security intelligence along architectural tiers to offer a better 'spectrum of value' related to place in the data flow, as proposed by fog computing or edge computing approaches. Intrinsically, shifting-left data and security analytics from cloud to endpoint devices not only leverages a better real-time for the 'sensing-processing-actuate' cycle, but also a stronger sense of security by reducing the attack surface of in-transit data and control. However, shifting-left intelligence is per se

not enough if key enabler elements of security such as authentication, authorization, availability, confidentiality, identification, integrity are not met. Regarding iIoT, the above security key enabler elements must be extended in order to achieve, real-time, safety, reliability, and resilience [1.24]. A brief review of the research literature points to the following main security challenges and desirable architectural requirements and features to take into account while designing safe and secure IoT endpoint devices [1.5, 1.14, 1.19, 1.29, 1.30, 1.33, 1.23, 1.24, 1.31]:

1. **Trust Anchor Technology** incorporating a RoT as the basis for a secure boot process where the hardware presents a trustworthy platform to the IoT device application. Examples of implemented mechanisms that provide a hardware-based root of trust are Intel Boot Guard [1.34], tRoot [1.35], ARM TrustZone [1.36], AMD Secure Processor[2], and Imaginations' OmniShield [1.37]. However, ARM TrustZone only partially implements RoT as it does not provide any kind of device identity support;

2. **Chain of Trust (CoT)** established under the hardware foundation of the RoT to validate at boot time all levels of software running on the IoT Endpoint device;

3. **Endpoint Identity** through a cryptographically unique identity number that the device must be able to prove that it truly represents that number;

4. **Secure Update** ensuring only correctly signed firmware updates can be applied for the long life cycle of an IoT endpoint device;

5. **Tamper-protection** (i.e., Physical Security Protection) to physically protect all hardware components of endpoint devices and their related interfaces from attack of criminals;

6. **Security Monitoring** to detect compromised endpoint devices by continuously checking some device' dynamics such as execution deadlines, control and data flows;

7. **Trusted Computing Base (TCB)** which assembles and leverages all the above architectural features among other algorithms and policies and allow the endpoint device to measure its own trustworthiness and verify the integrity of exchanged messages, internally or with service ecosystem;

8. **Low Power Consumption** may be required to achieve long battery life, and

[2] http://www.amd.com/en-us/innovations/software-technologies/security

can usually only undertake computationally simple cryptographic operations;

9. **Low cost** as the business case for many IoT Services demands that the cost of the IoT endpoint device be kept low, resulting in a resource-constrained device in term of processing capability, amounts of memory and system software (e.g., small-sized RTOS).

Several TCBs assisted by a CoT and internal CPU (e.g., ARM TrustZone) trust anchor or security coprocessor [1.38, 1.39, 1.40, 1.41, 1.42, 1.43, 1.44, 1.45, 1.46, 1.47, 1.48, 1.49, 1.50, 1.51, 1.52, 1.53, 1.54] have been implemented and deployed on IoT endpoints. Mainly for lightweight endpoints they are assisted by specialized microkernel-based RTOSes because of resource constraint. Although some implementations explore TrustZone dual-guest virtualization capability, they are implementation specific, failing to fully leverage TrustZone as a scalable, safe and secure technology across the three classes of IoT endpoint devices.

This work is about secure and safe embedded system design with the main focus on iIoT endpoint devices. The proposal goes towards engineering a scalable virtualization- and TrustZone-assisted TCB which scales up security without risking safety and real-time properties of IoT endpoint devices.

1.4 Research Questions and Methodology

By extending ARM TrustZone technology previously deployed on Cortex-A processors to Cortex-M microcontrollers (MCUs), ARM processors' ecosystem will leverage more efficient and easier to deploy security, as well as enhanced scalability of horsepower and code portability. These will be crucial as IoT endpoint TCBs are expected to be deployed across devices with different capabilities and resources (i.e., from lightweight to complex and gateway endpoints). For this reason, this thesis tries to answer a main question:

How to engineer a scalable virtualization- and TrustZone-assisted TCB which scales up security from low- to high-end processors without risking safety and real-time properties required by different IoT endpoint devices?

To a better understanding, the above question is further split into the following ones:

1. *How to implement TrustZone-assisted virtualization?*

2. *How to overcome the inherent ARM TrustZone dual-guest virtualization capability?*
3. *How to conceive a secure and real-time TrustZone-assisted TCB?*
4. *How to promote secure-by-design virtualized and TrustZone-assisted TCBs?*
5. *How to simplify system configurability and integration through a design ecosystem?*

To answer this questions, the following methodology is used:

1. Enumerating TrustZone' main features and their advantages when compared to existing security technologies based on internal CPU and coprocessor;
2. Conceiving and deploying several virtualized TCBs following an incremental approach to prove the scalability of the proposed solution upon ARM TrustZone scalability. In so doing, security and safety will scale up to include other features, evolving the proposed TCB technology to adapt to meet IoT endpoint device' requirements;
3. Analyzing approaches to extend ARM TrustZone to a fully RoT, mainly by studying existing solutions to support IoT device identity;
4. Conceiving and designing a domain-specific language (DSL) to promote easy design, integration and customization of a secure virtualized TCB assisted by TrustZone and other security building blocks.

1.5 State-of-the-Art

Safety and security are two main system-level requirements that drive the current development of IoT devices. There are several broad classes of approaches that have been applied to address them: software-based approaches for isolation such as microkernels, sandboxes and virtualization; hardware-assisted solutions for isolation such as hardware virtualization extensions; and hardware-based security oriented technologies such as secure processor architectures, hardware security modules, and secure silicon against insider attacks. Due to the extensive list of works on this spectrum, I will focus the description on most important research regarding TCB' implementations that individually or partially tackle the aforementioned requirements.

1.5.1 Software-based Isolation and Virtualization

Software-based isolation is a well-established strategy to provide safety and at some extent security. Among the existent approaches, there are two classes of systems that are widely used: microkernels and virtualization.

Microkernels are minimal operating system kernels where the basic idea is to reduce the kernel code to fundamental mechanisms, and implement actual system services (and policies) in user-level servers. Examples of existent microkernels deployed at large scale include QNX [1.55], Green Hills Integrity [3] and OKL4 microkernel [1.56]. The formal verification of the seL4 microkernel [1.57] shows that microkernels can be small enough, making a formal proof of functional correctness feasible.

Virtualization supports multiple OSes to run on top of the same hardware platform, by putting each OS into a separate virtual machine. The virtual machine (VM) is implemented by a hypervisor (or virtual-machine monitor) which provides virtual resources. While originally virtualization was understood to be pure or full (virtual resources are essentially indistinguishable from real ones), hardware limitations and performance issues led to widespread use of para-virtualization. Xen on ARM [1.58] is an hypervisor based on XEN source code and devoted to ARM cores which do not support any virtualization mechanism. The solution follows a para-virtualization approach which keep performance overhead minimal at the cost of some adaptations of guest OSes.

In [1.59], Lacoste discusses the three major disruptions shaping up the future of virtualization-assisted security (1) extension to embedded systems, (2) migration of security towards the hardware and (3) evolution towards multi-clouds, as well as how new hypervisor architectures should be defined to address upcoming threats. Figure 1.4 [1.59] depicts, according to Marc Lacoste's view, the evolution of the hypervisor landscape.

While some existent software-based virtualization solutions are completely focused in providing separation of concerns between functionally independent safety-critical software components, such as Xtratum [1.60] and Rodosvisor [1.61] in the aerospace industry, others solutions are completely devoted in implementing security-critical functionalities, such as Terra [1.62] and HIMA [1.63]. The problem is the TCB of a typical hypervisor has to be big enough to handle resource allocation and hardware virtualization. Therefore, commodity hypervisors are already struggling with their

[3] http://www.ghs.com/products/rtos/integrity.html

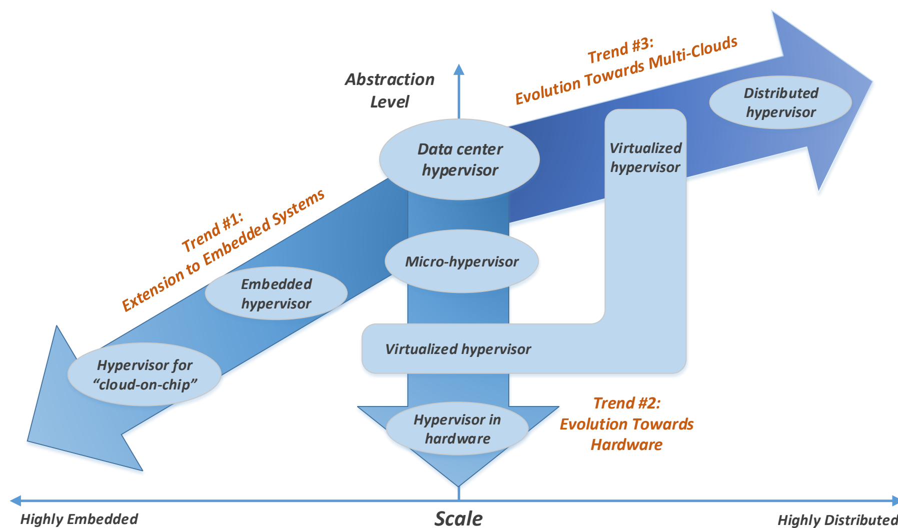


Figure 1.4: Evolution of the hypervisor landscape

own security problems [1.64].

Microkernels aim to provide a minimal layer of privileged software, while hypervisors aim to replicate and multiplex hardware resources. Both have an inherent need to abstract the hardware, although with different emphasis. For a hypervisor it is fundamental that virtual resources look as much as possible as the real ones, while implementation size is not a primary concern. For a microkernel, similarity of real and virtual is not a main driver, but is focused in the minimality requirement. To combine the best of both worlds Heiser proposed the concept of microvisor [1.65] through the development of OKL4 MicroVisor, a type of kernel that satisfies the combined objectives of microkernels and hypervisors: meets the hypervisor objective of minimal overhead for virtualization as well as the microkernel objective of minimal size.

1.5.2 Hardware-based Virtualization

Taking in mind the penalties incurred by traditional virtualization, industry and academia focused their attention in providing hardware support to assist virtualization. While the Big Players of processors industry introduced their own commercial off-the-shelf (COTS) technologies, some researchers developed their own customized hardware [1.66]. Both technologies rely on a new privileged processor mode - the hypervisor mode - altogether with MMU support for 2-level address translations. This features allow for direct execution of guests without the need for de-privileging, enabling certain instruction to directly affect virtual registers instead of trapping to

the hypervisor.

Intel VT, first released in 2005, has been a key factor in the growing adoption of full virtualization throughout the enterprise-computing world. Virtualization Technology for x86 (VT-x) provides a number of hypervisor assistance capabilities, including a true hardware hypervisor mode that enables unmodified guest operating systems to execute with reduced privilege. NOVA [1.40] implements a microhypervisor-based secure architecture with the support of Intel-VT. To minimize the attack surface, NOVA takes an extreme microkernel-like approach to virtualization by moving most functionality to user level. Lares [1.67] exploits Intel VT to implement an architecture to perform secure, active monitoring in a virtualized environment. IBM Turtles project [1.68] implements nested virtualization for Intel's virtualization technology based on the KVM hypervisor. It can host multiple guest hypervisors simultaneously, each with its own multiple nested guest operating systems.

In 2010, ARM announced the addition of hardware virtualization extensions (VE) to the ARM architecture as well as the first ARM core, the Cortex A15, that implements them. KVM/ARM [1.46] and CASL-Hypervisor [1.69] are examples of hypervisor solutions that make use of ARM hardware virtualization extensions to implement a full system virtualization that can run multiple unmodified guest operating systems on ARM platforms, while keeping the virtualization overhead low. There are other solutions taking advantage of ARM VE for virtualization, such as OSP [1.53], PrivateZone [1.54] and T-KVM [1.51], but their focus devoted to guarantee a higher level of security drive them to be complemented with other security oriented technologies. Hence, they will be presented and discussed in Section 1.5.3.

The Hellfire Hypervisor [1.70, 1.71] was the first virtualization solution supporting full-virtualization on MIPS processors. In [1.70], Zampiva et al. described the hypervisor implementation approach with real-time support to the MIPS M5150 processor, while in [1.71] Moratelli et al. demonstrated, through an exhaustive set of experiments, how MIPS virtualization can effectively be used to improve responsiveness while keeping the small footprint required by IoT applications. Hypervisors for MIPS architectures include also Kernel-based Virtual Machine (KVM) and Pike OS from Sysgo AG (now part of Thales Group).

There are still also some closed-source products available on market that exploit hardware-assisted virtualization technologies to implement commercial hypervisors.

INTEGRITY Multivisor[4] from Green Hills, SierraVisor[5] from Sierraware, Mentor Embedded Hypervisor[6] from Mentor Graphics are examples of existent hypervisors, deployed at large scale, that support high performance full-virtualization where no changes to the guest operating system are needed.

1.5.3 Secure Processor Architectures

Despite the introduction of hardware virtualization support in recent processors, the major players in the silicon industry introduced also some secure processor architectures. This architectures provide an ideal and proven foundation for hardware enablement and extensions needed for guaranteeing the required level of security of today's and next-generation IoT devices.

AMD Secure Processor, previously called Platform Security Processor (PSP), is a dedicated processor embedded inside of the main AMD CPU. It uses ARM TrustZone technology and a software-based trusted execution environment (TEE) to enable running third-party trusted applications. AMD Secure Processor is a hardware-based technology which enables secure boot up from BIOS level into the TEE. Intel Software Guard Extensions (SGX) [1.72], considered from some researchers as new generation of Intel TXT, is a set of instructions and mechanisms for memory accesses added to Intel architecture processors. These extensions allow an application to instantiate a protected container, designated as an enclave. An enclave could be used as a TEE, which provides confidentiality and integrity even without trusting several layers of the software system stack (BIOS, firmware, hypervisors, and OSes). OmniShield is the Imagination's security technology which ensures that applications that need to be secure are effectively and reliably isolated from each other, as well as protected from non-secure applications. This technology is so new that, to the best of my knowledge, there is no available closed- or open-source solution.

Among the existent secure processor architectures the most prevalent is ARM TrustZone. Introduced in 2004 [1.73] with the ARM1176, this hardware security extensions virtualizes a physical core as two virtual cores, providing two completely separated execution domains: the secure world and the non-secure world. A lot of research has been done around this technology, ranging from efficient virtualization solutions [1.38, 1.39, 1.41, 1.42, 1.74, 1.43], to trusted execution environments

[4] https://www.ghs.com/products/rtos/integrity_virtualization.html
[5] https://www.sierraware.com/arm_hypervisor.html
[6] https://www.mentor.com/embedded-software/hypervisor/

[1.75, 1.76, 1.45, 1.50, 1.77], or even a mix of both [1.53, 1.54].

The idea of using TrustZone technology to assist virtualization is not new. The work presented by Johannes Winter [1.38] in 2008 was the first scientific public attempt to exploit the TrustZone technology to assist virtualization. The paper introduces a virtualization framework for handling non-secure world guests, and presented a prototype based on a secure version of the Linux-kernel that was able to boot only an adapted Linux kernel as non-secure world guest. Later, Cereia et al. [1.39] describe an asymmetric virtualization layer on top of the TrustZone technology in order to support the concurrent execution of both a RTOS and a GPOS on the same processor. The evaluation process was conducted only on an emulator, and presented limited results about the virtual machine monitor (VMM) overhead and no concrete results about the VMM interference on the real-time characteristics. In [1.41] Frenzel et al. presented a minimal adapted version of Linux-kernel (as normal world OS) on top of a hypervisor running on the secure world side. SafeG [1.42], from TOPPERS Project, is a dual-OS open-source solution that takes advantage of ARM TrustZone extensions to concurrently execute an RTOS and a GPOS on the same hardware platform. ViMoExpress [1.74] is a lightweight virtualization solution, proposed by Oh et al., which exploits the TrustZone technology to accelerate the execution of two guest OSes. The prototype was tested in two different configurations: Android + μC/OS-II and Android + WinCE. Both works do not conducted any evaluation neither reported any experiments. Schwarz et al. [1.43] proposed an alternative system virtualization approach based on TrustZone which allows the switch between a virtualized and non-virtualized execution mode through soft reboots. There are still also some closed-source products available on market that exploit the hardware extensions of TrustZone technology for virtualization. Green Hills exploits TrustZone to provide facilities in INTEGRITY Multivisor to execute one or more guest operating systems on TrustZone-enabled ARM cores. The SierraVisor Hypervisor, from Sierraware, also leverages hardware security extensions included in ARM TrustZone-enabled devices to run multiple, high-level operating systems concurrently. Mentor Graphics and Cogs Systems also recently released their own solutions, the Mentor Embedded Hypervisor and the OKL4 Microvisor Lite[7], respectively.

The TEE's ability to offer isolated safe execution of authorized security software, known as 'trusted applications', enables it to provide end-to-end security by enforcing protected execution of authenticated code, confidentiality, authenticity, privacy, system integrity and data access rights [1.78]. ARM TrustZone is one among multi-

[7] https://cog.systems/okl4-microvisor/

ple technologies which provide the hardware foundation for a TEE implementation. Established companies have invested a lot of effort and resources to define their own TEE and integrate them in their own devices. Some companies have published their architectures, while some have preferred to preserve them into the obscurity. Companies which open their TEE include Nokia, Samsung and Nvidia. Nokia, integrate their TEE, called on-board credentials (ObC) [1.75, 1.76] into Nokia Symbian devices, and more recently into Nokia Lumia devices. Samsung proposed TZ-RKP [1.45] to provide real-time protection against attacks that aim at modifying an OS kernel running on the non-secure world side, and deployed on the latest Samsung Galaxy series. Nvidia proposes an open-source implementation of TEE called TLK [1.79]. Closed-TEE architectures include t-base from Trustonic[8], SecuriTEE[9] from Solacia, QSEE[10] from Qualcomm, and SierraTEE[11] from Sierraware. Trusted Foundation, developed by Trusted Logic, and Mobicore, developed by G&D, are disappearing from the market because the two companies joined their efforts and formed Trustonic. In the academic world, there are also some prototypes of TEE: Andix OS [1.50] developed by Graz University of Technology, ARMithril [1.80] implemented by North Carolina State University, and Trust-E [1.77] designed by University of Electronic Science and Technology of China.

Leveraging TrustZone technology, a number of monitoring, trusted storage, communication, attestation, introspection and integrity checking schemes have been proposed [1.81, 1.82, 1.83, 1.84, 1.45, 1.85, 1.44, 1.86, 1.87, 1.88, 1.89, 1.90, 1.91]. Santos et al. [1.81] proposes the use of TrustZone technology to implement a Trusted Language Runtime (TLR), a .NET framework which enables security critical codes programmed with the .NET bytecode to execute inside a trusted environment. ARMlock [1.82], is a hardware-based fault isolation system for ARM platforms. It uniquely leverages the memory domain feature in the commodity ARM processors to create multiple sandboxes for untrusted modules. Javier et al. propose Trusted Cell [1.83, 1.84] as a distributed framework that leverages the capabilities of a TrustZone-assisted TEE to provide Trusted Services such as a trusted storage. TrustDump [1.85] is a TrustZone-based memory acquisition mechanism capable of performing forensic analysis and facilitate malware analysis. Sprobes [1.44] implements an introspection mechanism for operating systems running on ARM TrustZone hardware. VeriUI [1.86] runs a Linux in TrustZone secure world to provide

[8]https://www.trustonic.com/solutions/trusted-secured-platform
[9]https://www.sola-cia.com/en/securiTee/product.asp
[10]https://www.qualcomm.com/products/snapdragon/security
[11]https://www.sierraware.com/open-source-ARM-TrustZone.html

an attested login for users. TrustUI [1.87] implements a system aiming at providing trusted path for mobile devices, which enables secure interaction between end users and services using TrustZone. DAA-TZ [1.88] is an efficient direct anonymous attestation scheme using TrustZone, to deal with the security and privacy issues specially for mobile users. Sun et al. presented TrustICE [1.89], a TrustZone-based isolation framework to provide isolated computing environments (ICEs) on mobile devices. The main idea of TrustICE is to create ICEs in the normal world domain rather than in the secure world side. Instead of using a hypervisor, TrustICE relies on TrustZone extensions to ensure the secure code in ICEs is securely isolated from an untrusted OS in the normal world. SeCReT [1.90] is a framework that builds a secure channel between the REE and TEE by enabling REE processes to use session keys in the REE that is regarded as unsafe region. SKEE [1.91], which stands for Secure Kernel-level Execution Environment, is a system that enables ARM platforms to support an isolated execution environment, designed to provide security monitoring and protection of the OS kernel.

OSP [1.53], PrivateZone [1.54] and T-KVM [1.51] are examples of solutions that simultaneously exploit ARM TrustZone and ARM VE to implement a combination of virtualization with TEE support. OSP [1.53] is an hybrid approach that utilizes both ARM TrustZone and VE, by implementing a TEE using a hypervisor, and activating the hypervisor only when the TEE is demanded by security critical codes. PrivateZone framework [1.54] explores both security and virtualization extensions of the ARM architecture to implement a private execution environment (PrEE) which allows developers to run security critical logic that is isolated from both the rich execution environment (REE) and TEE. T-KVM [1.51] proposes a security architecture for the KVM-on-ARM hypervisor following a combination of four isolation layers: ARM virtualization and security extensions (i.e., ARM VE and TrustZone), GlobalPlatform TEE specification and SELinux mandatory access control security policy.

There are still some works that make use of processor architectural features to implement security on resource constrained devices. TrustLite [1.92] and TyTAN [1.47] are both security architectures providing trusted computing functionality on tiny embedded systems. TrustLite [1.92] requires all software components to be loaded and their isolation to be configured at boot time. In contrast, TyTAN [1.47] provides higher flexibility by providing dynamic loading and unloading of multiple tasks at runtime, secure IPC with sender and receiver authentication, and real-time scheduling. Both technologies rely on Intel technology and exploit the Execution-Aware

Memory Protection Unit to provide isolation. Finally, from a different perspective, Aichouch et al. [1.93] introduce the general idea of a blind hypervisor, a hardware/-software co-designed approach to prevent attackers from accessing private elements of other virtual machines. Blind hypervision limits the rights of the hypervisor regarding memory access, so that a malicious agent executing with hypervisor rights cannot access the data of the VMs. The authors described a set of hardware extensions for many-core architectures to support such design.

1.5.4 Hardware Security Modules

Hardware security modules (HSMs) are specific hardware components that encapsulate security functions and provide the necessary trust primitives. HSMs are integrated chips specifically conceived and designed with security use-cases in mind. Typically, implementations range from smart cards [1.94] used for identification and authentication purposes, such as banking cards and identity documents, to Trusted Platform Modules (TPMs) [1.95], which are HSMs commonly used in personal computers. HSMs typically consist of a CPU core, data storage, a memory protection unit, sensors, cryptographic accelerators, and further peripheral components. Most HSMs employ sophisticated countermeasures against physical attacks, such as active sensors to detect fault and glitching attacks, and also employ cryptographic implementations which are hardened against side channel attacks [1.31]. Anderson et al. give an overview of cryptographic processors and their use in [1.96].

TPM [1.95] is an international standard for a secure cryptoprocessor, which is a dedicated MCU designed to secure hardware by integrating cryptographic keys into devices. TPM's technical specification was written by a computer industry consortium called Trusted Computing Group (TCG). Many manufacturers make TPMs. The Trusted Computing Group has certified TPMs manufactured by Infineon Technologies, Nuvoton, and STMicroelectronics.

There are five different types of TPM 2.0 [1.97] implementations:

1. ***Discrete TPM*** - Discrete TPMs are chips that implement only the TPM functionality in their own package. Functions are implement in hardware to resist software bugs and they also support tamper resistance. Discrete TPM provides the highest level of security;

2. ***Integrated TPM*** - Integrated TPM is the next level down in terms of secu-

rity. This level still has a hardware TPM but it is integrated into a chip that provides functions other than security. While they use hardware that resists software bugs, they are not required to implement tamper resistance. Intel has integrated TPMs in some of its chipsets;

3. ***Firmware TPM*** - Firmware TPMs are software-only solutions that run in a CPU's protected software. The code runs on the main CPU, so a separate chip is not required. While running like any other program, the code is protected in a TEE that is separated from the rest of the programs that are running on the CPU. Since these TPMs are entirely software solutions, they are vulnerable to software bugs within themselves. ARM and AMD have implemented firmware TPMs with TrustZone Technology;

4. ***Software TPM*** - Software TPMs are software emulators of TPMs that run with no more protection than a regular program that is part of an operating system. They depend entirely on the environment that they run in, so they provide no more security than what can be provided by the normal execution environment, and they are vulnerable to their own software bugs. They are useful for development purposes.

5. ***Virtual TPM*** - Virtual TPMs are provided by a hypervisor. These are therefore reliant on the hypervisor for security beyond the execution environment provided to the software running inside the virtual machine and therefore they provide a security level similar to a firmware TPM.

Barrett et al. discusses several frameworks built on the Trusted Platform Module in [1.98]. TinyTPM [1.99] is a lightweight cryptographic module for IP protection and for building trustworthy embedded systems. TinyTPM makes use of partial reconfiguration to perform hardware updates. The TinyTPM consumes only a few resources and is therefore well-suited to design secure, efficient, and low cost FPGA-based embedded systems. vTPM [1.100] implements the full TPM specification in software, and is integrated into a hypervisor environment to make TPM functions available to virtual machines. The virtual TPM facility supports four designs for certificate chains to link the virtual TPM to a hardware TPM, with security vs. efficiency trade-offs based on threat models.

Bosch developed its own HSM that satisfies automotive requirements [1.101]. The HSM was especially designed for protecting e-safety applications such as emergency break based on communications between vehicles or emergency call based on communications between vehicles and (traffic) infrastructures. The core of the HSM is

a secure CPU where security critical tasks are executed. The HSM also possesses its own random-access memory (RAM), boot read-only memory (ROM), Advanced Encryption Standard (AES) engine as well as a true random number generator as cryptographic peripheral. Parts of the HSM are also debug interfaces and an on-chip interconnect interface which is used for communication with the host core and to access the flash. The host core is a typical automotive qualified application processor providing an execution environment for safety-critical tasks. The flash is shared between host core and HSM, and the firmware both of the HSM and the host core is stored into the shared flash. A memory protection unit ensures that only the HSM is allowed to access its own HSM allocated data in the flash. When the HSM is powered up, the local boot code is loaded from the boot ROM and the HSM is initialized with the code stored in the shared flash.

Concluding, Hardware Security Modules are a necessary building block to harden embedded systems against attacks. To provide the necessary trust primitives and resistance to physical attacks, the security mechanisms must be rooted in hardware. HSMs are dedicated hardware security components that encapsulate security functions and provide the necessary trust primitives.

1.5.5 Secure Silicon Against Insider Attacks

So far, many deployed secure systems resistant to software threats have been assisted by hardware security technologies such as HSMs and secure processors. Notwithstanding such protection against software attacks, both approaches are not flawless, leading to the growing importance of the silicon security. Several reasons have been pointed [1.102, 1.103, 1.104, 1.105], mainly because chips are prone to insider attacks during their design and fabrication, and among them the following five are of the utmost relevance: (1) defenses against software-based attacks becomes mature and strong, (2) creative, new side-channel attacks is continually emerging, (3) compromised chip with embedded hardware Trojan, (4) threats embedded in hardware and firmware are currently undetectable by traditional security tools and (5) counterfeiting in supply chain is becoming bigger and mainstream in semiconductor industry. Due to the global economy, third-party components manufactured around the world have been integrated into new IoT endpoint devices, becoming difficult to ensure they have not been compromised or to trace them back to their sources.

In [1.102] several issues and challenges regarding silicon security are discussed such as (1) vulnerability and migration of security mechanism from software to hardware, (2)

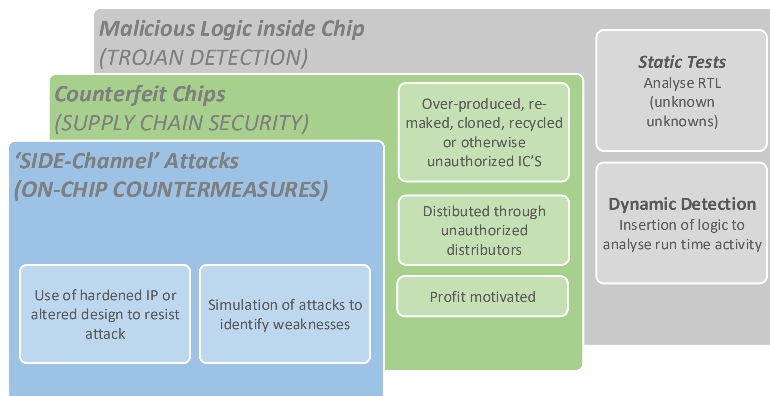


Figure 1.5: Levels of security concerns for designer and countermeasures for them

type of attacks covering all execution stack and their relative impact in each layer, (3) levels of security concerns for chip designer such as malicious logic embedded on chip, counterfeit chips and side-channel attacks (Figure 1.5 [1.102]), and (4) countermeasures for the above design concerns based on Trojan detection, supply chain security and on-chip monitoring for side-channel attacks.

Attacks at different stages of integrated circuit (IC) design flow were recognized and categorized as (1) third-party intellectual property (IP) and code reuse during requirements, design specification and register-transfer level (RTL) coding stages, (2) complicated third-party scripts during functional verification and logic synthesis stages and (3) physical IP during gate-level synthesis, place and route, and layout verification stages.

Additionally, modification to traditional system-on-chip design flow and methodology was suggested to enable robust hardware design methodology, traceability, and proof of health as demanded by several standards, in several domains like medical, automotive, avionics, railway and military. For proofing of health, an identity microscopy dielet or chiplet complemented with a cybersecurity co-processor were proposed as part of the design. The identity dielet should provide a unique identification to make the design genuine, enabling the chip to work only after the use of an activation key. The cybersecurity co-processor (Figure 1.6 [1.102]) is an IP block targeting issues such as hidden functionality, prevention of undeclared communications and chip usage (e.g., based on some physical events like memory access and power cycle to check if it is a second hand chip).

In [1.103] is claimed that main reasons for vulnerability and migration of security mechanism towards hardware are: (1) maturity of secure software development and its pro-active approach to security, (2) undefined hardware development with no

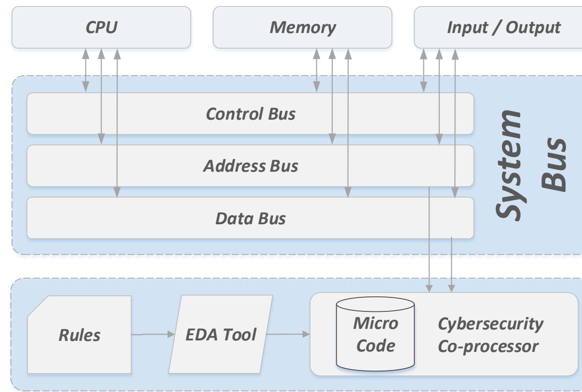


Figure 1.6: Cybersecurity co-processor for runtime trojan and side-channel detection

similar scrutiny in terms of security as software design methodology and (3) the inherently slower, more expensive and difficult hardware hacking which make us to blindly trust in hardware. Three RTL designs were reviewed for common security vulnerabilities and then the process to discover, exploit and fix them were discussed.

Rajendran et al. [1.104] also proposed modification to traditional SoC design flow with additional stages to test and search for *UNKNOWN UNKNOWNS*, i.e., hardware trojan as maliciously inserted rogue functionality during design and fabrication. They recognize that detection techniques targeting Trojans inserted in a foundry are limited by their detection sensitivity. Therefore, carefully designed Trojans whose sizes are much less than this detection sensitivity may go undetected. Their proposed secure hardware design flow will firstly leverage processor encryption by using trusted security validation team, trusted integration team and logic encryption techniques (e.g., by adding extra XOR/XNOR gates, logic states into the state machine, or by inserting memory elements). They define logic encryption as hiding the hardware's functionality instead of encrypting the design file by a cryptographic algorithm. Secondly, the trusted security team performs logic encryption on the components obtained from the design teams and finally, security modules are designed with provisions to store keys. Processor encryption will be crucial as it ensures that inserted Trojans larger and smaller than the detection sensitivity will be detected and will not function, respectively. They also proposed several security modules for secure execution of a program and approaches to bypass them by rogue insiders in the design house and the foundry, if processor encryption is not leveraged.

Power fingerprinting (PFP) [1.105] is an IP as a trojan carrier capable of detecting tampering at all levels of the execution stack, from hardware to firmware to software. It utilizes side channels to assess the integrity of an electronic device through monitoring, analysis and identification of otherwise undetectable threats in hardware and

firmware. It looks for anomalies that could be indicators of malicious behavior which are manifested in alternating current, direct current and electromagnetic interference (EMI) power signals. Because PFP can be embedded in the chip, it operates within the resource constraints of IoT while leveraging the following characteristics: it is able to detect dormant as well as active attacks, it does not require threat intelligence, it requires no additional software, and it cannot be detected or evaded by attackers. PFP observes a chip operation to look for some signature based on power consumption, timing/deadline, thermal, or electromagnetic emissions and detect deviation from expected operation. For example, since power consumption and electromagnetic emissions only depend on the circuit layout, semiconductor technology, and manufacturing process, physical sensors are used to capture fine-grained side-channel signals, which contain unique signature that emerge during operation of a given hardware/firmware combination. To assign unique signature to an IoT endpoint device, the execution code can be personalized for desired functionalities and the signature extract and load as microcode each time the device is updated.

1.6 Conclusions

Table 1.1 presents a gap analysis among the most important recent research regarding TCB' implementations for IoT endpoint devices compared to my envisioned solution. Several existing safe and secure TCB' solutions have been compared based on key parameters such as (1) support for device identity, (2) ARM TrustZone assistance, (3) virtualization assistance, (4) multiple guest support, (5) trusted execution environment support, (6) real-time guaranty, (7) scalability and (8) cost-effectiveness.

Firstly I conclude that hardware security depends on physical security, supply chain security, software security and firmware security. Secondly, to leverage demanded features of IoT endpoint devices as envisioned by the fog computing trend, the scalability of security and data intelligence at the IoT endpoint ecosystem will be a must. To achieve such scalability while dealing with other design metrics (e.g., real-time, safety, low power consumption and low cost), a scalable TCB framework should be promoted based on a mixed of hardware and software building blocks such as RoT, secure boot, device identity, secure update, tamper-protection and virtualization-assisted technology. The safety architecture of IoT endpoints will be guaranteed through virtualization technology alongside deployment of redundant

Table 1.1: Gap analysis among existing TCBs

	Device ID	*V-Assisted*	*TZ-Assisted*	*Guest OSes*	*TEE*	*Real-Time*	*Scalability*	*Cost-Effective*
Winter et al. [1.38]	Yes	No	Yes	1	Yes	No	No	Medium
Cereia et al. [1.39]	No	ARM TZ	Yes	2	No	Yes	Partially	High
NOVA [1.40]	No	Intel VT	No	Multi	No	No	No	Low
Frenzel et al. [1.41]	No	ARM TZ	Yes	1	Yes	No	No	Medium
SafeG [1.42]	No	ARM TZ	Yes	2	No	Yes	Partially	High
Schwarz et al. [1.43]	No	ARM TZ	Yes	2	No	No	No	Medium
SPROBES [1.44]	No	No	Yes	1	Yes	No	No	Medium
TZ-RKP [1.45]	Yes	No	Yes	1	No	No	No	Medium
KVM/ARM [1.46]	No	ARM VE	No	Multi	No	No	No	Low
TyTAN [1.47]	Yes	No	No	1	No	Yes	No	High
XVisor [1.48]	No	ParaV	No	Multi	No	No	Partially	High
H-SVM [1.49]	No	AMD-V	No	Multi	No	No	No	Low
ANDIX [1.50]	No	No	Yes	2	Yes	No	Partially	High
T-KVM [1.51]	Yes	ARM VE	Yes	Multi	Yes	Yes	Partially	Low
Hellfire[1.52]	No	MIPS-V	No	Multi	No	Yes	No	Medium
OSP [1.53]	Yes	ARM VE	Yes	Multi	Yes	No	Partially	Low
PrivateZone [1.54]	No	ARM VE	Yes	Multi	Yes	No	Partially	Low

(i.e., shadow/hot standby) functional VMs, as well as VMs for trust monitoring. Due to ARM TrustZone popularity and scalability, it will be used as a TEE anchor and partially as a TPM. Finally, for easy design and deployment of secure and safe TCB, DSL technology was conceived and designed for the purpose.

1.7 Thesis Structure

This thesis is structured as follows:

- **Chapter 2** describes the research platform and tools used during the development of this thesis. I start by identifying the platform requirements and discussing its choice based on an outlined criteria. I describe several architectural aspects of the ARMv7-A architecture and TrustZone security extensions. I review the general architecture of the Zynq device, and, finally, I also present an overview and motivation for the used tools and benchmark suites;

- **Chapter 3** proposes the development of a lightweight TrustZone-assisted hypervisor (LTZVisor) as a study to clearly understand and evaluate the use of TrustZone hardware technology to assist virtualization. I describe all the details behind its implementation, highlighting the benefits and stating the identified limitations, based on an extensive set of performed experiments;

- **Chapter 4** describes TZVisor as a fully-featured virtualization-assisted TCB providing complete hardware isolation as well as multiple guest OS support.

I explain how the identified limitations in the development of LTZVisor are completely overcame in TZVisor, and how the solution can scale from the powerful applications processors to the smallest of microcontrollers. Two use cases in different application domains (aerospace and industrial) are described, discussed, and evaluated;

- **Chapter 5** presents T-TZVisor as a TCB implementation that fully and simultaneously addresses safety, security and real-time, guaranteed through the use of virtualization alongside an enhanced trusted execution environment, and other hardware trusted anchors. I describe how T-TZVisor integrates software and hardware components to secure guest OSes and enable trusted computing in ARM platforms, without requiring additional hardware virtualization extensions.
- **Chapter 6** presents a domain-specific language which mainly decouples the building blocks of virtualization-assisted TCB, leveraging easy customization towards target platforms and applications. I describe how a service-oriented programming model can help automating the generation of a customizable TCB system, ensuring correctness by design while powering components development based on service compositions, and boosting the development time due to the high abstraction level of the process.
- **Chapter 7** concludes this thesis. I present the conclusions obtained from this research, highlighting the contributions, identifying the limitations, and suggesting future work towards addressing pointed limitations.

1.8 List of Publications

The work developed in this thesis has directly and indirectly contributed to the following publications:

Journal Publications

- **S. Pinto**, T. Gomes, J. Pereira, J. Cabral and A. Tavares, "*IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices*", in IEEE Internet Computing, vol. 21, no. 1, pp. 40-47, Jan.-Feb, 2017.

- **S. Pinto**; J. Pereira; T. Gomes; M. Ekpanyapong; A. Tavares, "*Towards a TrustZone-assisted Hypervisor for Real Time Embedded Systems*", *in IEEE Computer Architecture Letters*, vol.PP, no.99, pp.1-1

- T. Gomes, P. Garcia, **S. Pinto**, J. Monteiro and A. Tavares, "*Bringing Hardware Multithreading to the Real-Time Domain*", in IEEE Embedded Systems Letters, vol. 8, no. 1, pp. 2-5, March 2016.

- T. Gomes, J. Pereira, P. Garcia, F. Salgado, V. Silva, **S. Pinto**, M. Ekpanyapong and A. Tavares, "Hybrid real-time operating systems: deployment of critical FreeRTOS features on FPGA", in International Journal of Embedded Systems, vol. 8, no. 5-6, pp.483-492, 2016.

Book Chapters

- J. Pereira, D. Oliveira, P. Matos, R. Machado, **S. Pinto**, T. Gomes, V. Silva, E. Qaralleh, N. Cardoso, and P. Cardoso, "*Hardware-assisted Real-Time Operating System Deployed on FPGA*", in "Informatik/Kommunikationstechnik" subseries of the "Fortschritt-Berichte VDI" series edited by VDI Verlag, 2014.

Conference Proceedings

- **S. Pinto**, J. Cabral, and T. Gomes, "*We-Care: An IoT-based Health Care System for Elderly People*", *in Proceedings of International Conference on Industrial Technology (ICIT)*, Toronto, 2017.

- **S. Pinto**, A. Tavares, and S. Montenegro, "*Space and Time Partitioning with hardware support for Space Applications*", *in Proceedings of the Data Systems In Aerospace (DASIA)*, Tallinn, 2016.

- **S. Pinto**, A. Tavares, and S. Montenegro, "*Hypervisor for Real-Time Space Applications*", *in Proceedings of 4S Symposium*, Malta, 2016.

- R. Machado, **S. Pinto**, J. Cabral, and A. Tavares, "*FPGA vendor-agnostic IP-XACT- and XSLT-based RTL design generator*", 2016 18th Mediterranean Electrotechnical Conference (MELECON), Lemesos, 2016, pp. 1-6.

- T. Gomes, F. Salgado, **S. Pinto**, J. Cabral, and A. Tavares, "*Towards an FPGA-based network layer filter for the Internet of Things edge devices*", 2016

IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), Berlin, 2016, pp. 1-4.

- **S. Pinto**, D. Oliveira, J. Pereira, J. Cabral, and A. Tavares, "*FreeTEE: When real-time and security meet*", 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), Luxembourg, 2015, pp. 1-4.

- E. Qaralleh, D. Lima, T. Gomes, A. Tavares and **S. Pinto**, "*HcM-FreeRTOS: Hardware-centric FreeRTOS for ARM Multicore*", 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), Luxembourg, 2015, pp. 1-4.

- T. Gomes, **S. Pinto**, P. Garcia, and A. Tavares, "*RT-SHADOWS: Real-time system hardware for agnostic and deterministic OSes within softcore*", 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), Luxembourg, 2015, pp. 1-4.

- T. Gomes, **S. Pinto**, T. Gomes, A. Tavares, and J. Cabral, "*Towards an FPGA-based edge device for the Internet of Things*", 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), Luxembourg, 2015, pp. 1-4.

- **S. Pinto**, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares, "*Towards a Lightweight Embedded Virtualization Architecture Exploiting ARM TrustZone*", Proceedings of the 2014 IEEE Conference on Emerging Technology & Factory Automation (ETFA), Barcelona, 2014, pp. 1-4.

- **S. Pinto**, J. Pereira, D. Oliveira, F. Alves, E. Qaralleh, M. Ekpanyapong, J. Cabral, and A. Tavares, "*Porting SLOTH system to FreeRTOS running on ARM Cortex-M3*", 2014 IEEE 23rd International Symposium on Industrial Electronics (ISIE), Istanbul, 2014, pp. 1888-1893.

- J. Pereira, D. Oliveira, **S. Pinto**, N. Cardoso, V. Silva, T. Gomes, J. Mendes, P. Cardoso, "*Co-Designed FreeRTOS Deployed on FPGA*", 2014 Brazilian Symposium on Computing Systems Engineering, Manaus, 2014, pp. 121-125.

1.9 Summary

The goal of this Chapter was to provide an introductory vision about this thesis. I started by presenting the motivational reasons for the proposed work, and formalizing the problem statement. Then I explained and limited the focus of this thesis, and identified the research questions and the methodology proposed to answer them. I presented a survey about the most important state-of-the-art solutions that individually or partially addresses safety, security and real-time requirements. In doing so, I then described my envisioned solution, ending with an overview of the structure of this document as well as a complete list of publications that directly and indirectly contributed to this thesis.

References

[1.1] G. Kortuem, F. Kawsar, V. Sundramoorthy, and D. Fitton, "Smart objects as building blocks for the Internet of things," *IEEE Internet Computing*, vol. 14, pp. 44–51, Jan 2010.

[1.2] L. Tan and N. Wang, "Future Internet: The Internet of Things," in *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, vol. 5, pp. V5–376–V5–380, Aug 2010.

[1.3] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010.

[1.4] Symantec, "An Internet of Things Reference Architecture." White Paper, 2016.

[1.5] GSMA, "IoT Security Guidelines Overview Document." White Paper, Version 1.1, November 2016.

[1.6] Intel, "The Intel IoT Platform: Architecture Specification." White Paper, 2015.

[1.7] L. D. Xu, W. He, and S. Li, "Internet of Things in Industries: A Survey," *IEEE Transactions on Industrial Informatics*, vol. 10, pp. 2233–2243, Nov 2014.

[1.8] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of Things for Smart Cities," *IEEE Internet of Things Journal*, vol. 1, pp. 22–32, Feb 2014.

[1.9] Cisco, "The Cisco Edge Analytics Fabric System." White Paper, 2016.

[1.10] A. Bassi, M. Bauer, M. Fiedler, T. Kramp, R. Van Kranenburg, S. Lange, and S. Meissner, "Enabling Things to Talk," *Designing IoT Solutions With the IoT Architectural Reference Model*, pp. 163–211, 2013.

[1.11] oneM2M, "The Interoperability Enabler for the Entire M2M and IoT Ecosystem." White Paper, January 2015.

[1.12] Cisco, "The Internet of Things Reference Model." White Paper, 2016.

[1.13] Oracle, "The Oracle Enterprise Architecture Framework." White Paper, October 2009.

[1.14] GSMA, "IoT Security Guidelines Endpoint Ecosystem." White Paper, Version 1.1, November 2016.

[1.15] D. Bodeau and R. Graubart, "Cyber resiliency engineering framework," *MTR110237, MITRE Corporation*, 2011.

[1.16] PRQA, "Developing Secure Embedded Software." White Paper, 2016.

[1.17] PRQA, "Addressing Security Vulnerabilities in Embedded Applitications using Best Practice Software Development Processes and Standards." White Paper, 2016.

[1.18] PRQA, "How IoT is Making Security Imperative for All Embedded Software." White Paper, 2016.

[1.19] M. Vai, D. Whelihan, B. Nahill, D. Utin, S. O'Melia, and R. Khazan, "Secure Embedded Systems," *Lincoln Laboratory Journal*, vol. 22, no. 1, pp. 110–122, 2016.

[1.20] P. Fremantle, "A Reference Architecture for the Internet of Things." White Paper, Version 0.9.0, October 2015.

[1.21] T. Bradicich, "7 principles of IoT - A Personal Perspective," in *Industrial Internet Consortium Summit, Houston, Texas*, 2015.

[1.22] P. Mannion, "Embedded computing in the age of IoT - Part 3: Design for Data Harvesting." Intel IoT Solutions Alliance, 2016.

[1.23] Q. Jing, A. V. Vasilakos, J. Wan, J. Lu, and D. Qiu, "Security of the Internet of Things: perspectives and challenges," *Wireless Networks*, vol. 20, no. 8,

pp. 2481–2501, 2014.

[1.24] A.-R. Sadeghi, C. Wachsmann, and M. Waidner, "Security and Privacy Challenges in Industrial Internet of Things," in *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pp. 54:1–54:6, ACM, 2015.

[1.25] IIC, "Industrial Internet of Things - Volume G4: Security Framework." Industrial Internet Consortium, Version 1.0, Sept 2016.

[1.26] R. T. Tiburski, L. A. Amaral, E. de Matos, D. F. G. de Azevedo, and F. Hessel, "The Role of Lightweight Approaches Towards the Standardization of a Security Architecture for IoT Middleware Systems," *IEEE Communications Magazine*, vol. 54, pp. 56–62, December 2016.

[1.27] M. MacKenzie and A. Haegele, "IoT security challenge (Talking Heads)," *IoT Now*, vol. 6, pp. 12–15, December 2016/ January 2017.

[1.28] M. E. Porter and J. E. Heppelmann, "How smart, connected products are transforming companies," *Harvard Business Review*, vol. 93, no. 10, pp. 53–71, 2015.

[1.29] Y. Loisel and S. di Vito, "Secure the IoT: Part 1, Public Key Cryptography Secures Connected Devices." Maxim Application Note 6004, Jul 2016.

[1.30] Y. Loisel and S. di Vito, "Secure the IoT: Part 2, A Secure Boot, the Root of Trust for Embedded Devices." Maxim Application Note 6005, Jul 2016.

[1.31] M. Wolf and A. Weimerskirch, "Hardware Security Modules for Protecting Embedded Systems." White Paper, escrypt.

[1.32] W. S. Technologies, "An Executive Guide to Cyber Security for Operational Technology." Wurldtech Executive Guide, 2016.

[1.33] S. L. Keoh, S. S. Kumar, and H. Tschofenig, "Securing the Internet of Things: A Standardization Perspective," *IEEE Internet of Things Journal*, vol. 1, pp. 265–275, June 2014.

[1.34] Intel, "New Microarchitectures for 4th Gen Intel Core Processor Platforms." Product Brief, 2013.

[1.35] Synopsys, "DesignWare tRoot Secure Hardware Root of Trusty." Datasheet, 2015.

[1.36] ARM, "ARM Security Technology - Building a Secure System using TrustZone Technology." PRD29-GENC-009492C, April 2009.

[1.37] Imagination, "Omnishield - An Overview & Requirements." White Paper, MD01185, Version 1.1, November 2016.

[1.38] J. Winter, "Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms," in *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, STC '08, pp. 21–30, ACM, 2008.

[1.39] M. Cereia and I. C. Bertolotti, "Virtual Machines for Distributed Real-time Systems," *Comput. Stand. Interfaces*, vol. 31, pp. 30–39, Jan. 2009.

[1.40] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-based Secure Virtualization Architecture," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pp. 209–222, ACM, 2010.

[1.41] T. Frenzel, A. Lackorzynski, A. Warg, and H. Härtig, "ARM Trustzone as a Virtualization Technique in Embedded Systems," in *Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya*, 2010.

[1.42] D. Sangorrin, S. Honda, and H. Takada, "Dual Operating System Architecture for Real-Time Embedded Systems," in *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Brussels, Belgium*, pp. 6–15, 2010.

[1.43] O. Schwarz, C. Gehrmann, and V. Do, "Affordable Separation on Embedded Platforms," in *Proceedings of the 7th International Conference on Trust and Trustworthy Computing - Volume 8564*, pp. 37–54, Springer-Verlag New York, Inc., 2014.

[1.44] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing Kernel Code Integrity on the Trustzone Architecture," *arXiv preprint arXiv:1410.7747*, 2014.

[1.45] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pp. 90–102, ACM, 2014.

[1.46] C. Dall and J. Nieh, "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor," *SIGPLAN Not.*, vol. 49, pp. 333–348, Feb. 2014.

[1.47] F. Brasser, B. E. Mahjoub, A. R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2015.

[1.48] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi, "Embedded Hypervisor Xvisor: A Comparative Analysis," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 682–691, March 2015.

[1.49] S. Jin, J. Ahn, J. Seol, S. Cha, J. Huh, and S. Maeng, "H-SVM: Hardware-Assisted Secure Virtual Machines under a Vulnerable Hypervisor," *IEEE Transactions on Computers*, vol. 64, pp. 2833–2846, Oct 2015.

[1.50] A. Fitzek, F. Achleitner, J. Winter, and D. Hein, "The ANDIX research OS - ARM TrustZone meets industrial control systems security," in *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pp. 88–93, July 2015.

[1.51] M. Paolino, A. Rigo, A. Spyridakis, J. Fanguede, P. Lalov, and D. Raho, "T-KVM: A Trusted Architecture for KVM ARM v7 and v8 Virtual Machines," in *The Sixth International Conference on Cloud Computing, GRIDs, and Virtualization*, pp. 39–45, March 2015.

[1.52] C. Moratelli, S. Johann, and F. Hessel, "Exploring Embedded Systems Virtualization Using MIPS Virtualization Module," in *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pp. 214–221, ACM, 2016.

[1.53] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[1.54] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. Kang, "PrivateZone: Providing a Private Execution Environment using ARM TrustZone," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, pp. 1–1, 2016.

[1.55] P. Laroux and B. Graham, "Secure by Design: Using a Microkernel RTOS to Build Secure, Fault-Tolerant Systems," in *QNX Sotfware Systems, White Paper*, April 2009.

[1.56] K. Elphinstone and G. Heiser, "From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 133–150, ACM, 2013.

[1.57] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive Formal Verification of an OS Microkernel," *ACM Trans. Comput. Syst.*, vol. 32, pp. 2:1–2:70, Feb. 2014.

[1.58] J. Y. Hwang, S. B. Suh, S. K. Heo, C. J. Park, J. M. Ryu, S. Y. Park, and C. R. Kim, "Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones," in *2008 5th IEEE Consumer Communications and Networking Conference*, pp. 257–261, Jan 2008.

[1.59] M. Lacoste, "What Does the Future Hold for Hypervisor Security?," in *Workshop on Trustworthy Clouds (ESORICS)*, 2013.

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

[1.61] A. Tavares, A. Carvalho, P. Rodrigues, P. Garcia, T. Gomes, J. Cabral, P. Cardoso, S. Montenegro, and M. Ekpanyapong, "A customizable and ARINC 653 quasi-compliant hypervisor," in *2012 IEEE International Conference on Industrial Technology*, pp. 140–147, March 2012.

[1.62] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-based Platform for Trusted Computing," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 193–206, Oct. 2003.

[1.63] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang, "HIMA: A Hypervisor-Based Integrity Measurement Agent," in *2009 Annual Computer Security Applications Conference*, pp. 461–470, Dec 2009.

[1.64] M. Pearce, S. Zeadally, and R. Hunt, "Virtualization: Issues, Security Threats, and Solutions," *ACM Comput. Surv.*, vol. 45, pp. 17:1–17:39, Mar. 2013.

[1.65] G. Heiser and B. Leslie, "The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors," in *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*, APSys '10, (New York, NY, USA), pp. 19–24,

ACM, 2010.

[1.66] P. Garcia, T. Gomes, F. Salgado, J. Monteiro, and A. Tavares, “Towards Hardware Embedded Virtualization Technology: Architectural Enhancements to an ARM SoC,” *SIGBED Rev.*, vol. 11, pp. 45–47, Sept. 2014.

[1.67] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, “Lares: An Architecture for Secure Active Monitoring Using Virtualization,” in *2008 IEEE Symposium on Security and Privacy*, pp. 233–247, May 2008.

[1.68] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har’El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, “The Turtles Project: Design and Implementation of Nested Virtualization,” in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI’10, pp. 423–436, USENIX Association, 2010.

[1.69] C. T. Liu, K. C. Chen, and C. H. Chen, “CASL hypervisor and its virtualization platform,” in *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*, pp. 1224–1227, May 2013.

[1.70] S. Zampiva, C. Moratelli, and F. Hessel, “A hypervisor approach with realtime support to the MIPS M5150 processor,” in *Sixteenth International Symposium on Quality Electronic Design*, pp. 495–501, March 2015.

[1.71] C. Moratelli, S. Filho, M. Neves, and F. Hessel, “Embedded Virtualization for the Design of Secure IoT Applications,” in *International Symposium on Rapid System Prototyping*, October 2016.

[1.72] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative Instructions and Software Model for Isolated Execution,” in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP ’13, pp. 10:1–10:1, ACM, 2013.

[1.73] T. Alves and D. Felton, “TrustZone: Integrated Hardware and Software Security,” *Technology In-Depth*, vol. 3, no. 4, pp. 18–24, 2004.

[1.74] S. Oh, K. Koh, C. Kim, K. Kim, and S. Kim, “Acceleration of dual OS virtualization in embedded systems,” in *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*, pp. 1098–1101, Dec 2012.

[1.75] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala, “On-board Creden-

tials with Open Provisioning," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pp. 104–115, ACM, 2009.

[1.76] K. Kostiainen, *On-board Credentials: An Open Credential Platform for Mobile Devices*. Doctoral Dissertation, Aalto University, 2012.

[1.77] X. Yang, P. Shi, B. Tian, B. Zeng, and W. Xiao, "Trust-E: A Trusted Embedded Operating System Based on the ARM Trustzone," in *2014 IEEE 11th Intl Conf on Ubiquitous Intelligence and Computing*, pp. 495–501, Dec 2014.

[1.78] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, pp. 57–64, Aug 2015.

[1.79] Nvidia, "TLK: A FOSS Stack for Secure Hardware Tokens," 2016.

[1.80] J. H. Shah *et al.*, "ARMithril: A Secure OS Leveraging ARM's TrustZone Technology," 2012.

[1.81] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications," *SIGARCH Comput. Archit. News*, vol. 42, pp. 67–80, Feb. 2014.

[1.82] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "ARMlock: Hardware-based Fault Isolation for ARM," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pp. 558–569, ACM, 2014.

[1.83] J. González and P. Bonnet, *Towards an Open Framework Leveraging a Trusted Execution Environment*, pp. 458–467. Cham: Springer International Publishing, 2013.

[1.84] J. González, *Operating System Support for Run-time Security with a Trusted Execution Environment*. Doctoral Dissertation, IT University of Copenhagen, 2015.

[1.85] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, *TrustDump: Reliable Memory Acquisition on Smartphones*, pp. 202–218. Cham: Springer International Publishing, 2014.

[1.86] D. Liu and L. P. Cox, "VeriUI: Attested Login for Mobile Devices," in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*,

HotMobile '14, pp. 7:1–7:6, ACM, 2014.

[1.87] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building Trusted Path on Untrusted Device Drivers for Mobile Devices," in *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pp. 8:1–8:7, 2014.

[1.88] B. Yang, K. Yang, Y. Qin, Z. Zhang, and D. Feng, *DAA-TZ: An Efficient DAA Scheme for Mobile Devices Using ARM TrustZone*, pp. 209–227. Cham: Springer International Publishing, 2015.

[1.89] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices," in *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pp. 367–378, IEEE Computer Society, 2015.

[1.90] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment," in *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[1.91] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[1.92] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A Security Architecture for Tiny Embedded Devices," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pp. 10:1–10:14, 2014.

[1.93] P. Dubrulle, R. Sirdey, P. Dore, M. Aichouch, and E. Ohayon, "Blind hypervision to protect virtual machine privacy against hypervisor escape vulnerabilities," in *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pp. 1394–1399, July 2015.

[1.94] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Examining smart-card security under the threat of power analysis attacks," *IEEE Transactions on Computers*, vol. 51, pp. 541–552, May 2002.

[1.95] T. Morris, "Trusted platform module," in *Encyclopedia of Cryptography and Security*, pp. 1332–1335, Springer, 2011.

[1.96] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov, "Cryptographic Processors-A Survey," *Proceedings of the IEEE*, vol. 94, pp. 357–369, Feb 2006.

[1.97] W. Arthur and D. Challener, *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security.* Apress, 2015.

[1.98] M. Barrett and C. Thomborson, "Frameworks Built on the Trusted Platform Module," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 2, pp. 59–62, Sept 2006.

[1.99] T. Feller, S. Malipatlolla, D. Meister, and S. A. Huss, "TinyTPM: A lightweight module aimed to IP protection and trusted embedded platforms," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 6–11, June 2011.

[1.100] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: Virtualizing the Trusted Platform Module," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, (Berkeley, CA, USA), USENIX Association, 2006.

[1.101] M. Wolf and T. Gendrullis, "Design, Implementation, and Evaluation of a Vehicular Hardware Security Module," in *Proceedings of the 14th International Conference on Information Security and Cryptology*, ICISC'11, (Berlin, Heidelberg), pp. 302–318, Springer-Verlag, 2012.

[1.102] W. Rhines, "Secure silicon: Enabler for the Internet of Things," in *Design, Automation and Test in Europe (DATE), Keynote*, 2016.

[1.103] J. FitzPatrick, "SecSi Product Development: Techniques for ensuring Secure Silicon applied to open-source Verilog projects," in *Black Hat*, 2014.

[1.104] J. Rajendran, A. K. Kanuparthi, M. Zahran, S. K. Addepalli, G. Ormazabal, and R. Karri, "Securing Processors Against Insider Attacks: A Circuit-Microarchitecture Co-Design Approach," *IEEE Design Test*, vol. 30, pp. 35–44, April 2013.

[1.105] P. Cybersecurity, "Embedding Security in the Internet of Things," in *A White Paper from PFP Cybersecurity*, 2016.

"ARM processors continue to power the vast majority of mobile devices in high-volume consumer markets, where the need to work at the ES level to optimize designs for performance, low power, and cost is most acute."

- A.K. Kalekos

2

Research Platform and Tools

The choice of the platform and tools that system designers make for developing their solution plays a crucial role in the eventual success of their product. Usually, system designers have multiple platform options, and, obviously, the process of selection is not trivial, being typically hindered by several aspects and restricted by several requirements.

In this Chapter, I describe the research platform and tools used during the development of this thesis. I start by pointing out the platform requirements and discussing the platform selection based on an outlined criteria. Then, I describe several architectural aspects of the ARMv7-A architecture, and TrustZone technology. I cover several aspects of the Zynq device, describing its general architecture and highlighting the provided security facilities. Finally, I describe the choices for operating system stacks, as well as benchmark suites.

This Chapter is organized as follows: Section 2.1 presents the platform requirements and justifies the selection of the Zynq device targeting an ARM Cortex-A9 with TrustZone support. Section 2.2 describes the ARMv7-A architecture, mainly focusing on the processor modes, as well as core and system registers. Then, Section 2.3 explains the hardware and software architecture of ARM TrustZone technology.

Section 2.4 reviews the general architecture of the Zynq device, describing several security facilities. A comparison is made between the devices of the Zynq-7000 family, and three important Zynq-based platforms are described. Finally, the Chapter closes with the discussion and selection of the Operating System stacks, as well as the benchmark suites (Sections 2.5 and 2.6), followed then by a short summary (Section 2.7).

2.1 Platform Requirements

The previous Chapter demonstrated ARM processors are becoming widespread in the embedded space. The ongoing trend for using ARM-based platforms is undeniable, and numbers definitely support this trend: more than 86 billion ARM-based chips were shipped, according to the ARM media stats[1] in September 2016. The combination of performance, wide offering and low cost make it such they simply cannot be ignored. The use of ARM TrustZone for securing and virtualizing embedded products is also gaining momentum. While ARM continues to spread TrustZone technology from the applications processors to the smallest of microcontrollers, it is undeniable that this technology is gaining an increasing relevance.

The ultimate goal of this thesis goes towards the implementation of a scalable virtualization- and TrustZone-assisted TCB, which scales up security without risking safety and real-time properties. Taking this into consideration, the following requirements were identified and established:

- ***Requirement 1:*** The selected platform must use an ARM processor;
- ***Requirement 2:*** The selected ARM processor must be able to implement a cost-effective solution;
- ***Requirement 3:*** The selected ARM processor must be able to run general-purpose operating systems (GPOSes), which means the processor must be endowed with a memory management unit (MMU);
- ***Requirement 4:*** The selected platform must be endowed with an ARM processor enhanced with TrustZone security extensions;
- ***Requirement 5:*** The selected platform must be endowed with an ARM processor that provides facilities for hardware-assisted virtualization;
- ***Requirement 6:*** The selected platform must provide some hardware security facilities that go beyond TrustZone support;

A quick look over the presented requirements can easily drive the first restriction: the use of an ARM processor. ARM processors have evolved considerably since the first release in 1985, presenting, by the time of starting of this thesis, seven architectures: from ARMv1 to ARMv7. ARMv8 was just introduced recently, and

[1] https://www.arm.com/-/media/arm-com/news/ARM-media-fact-sheet-2016.pdf?la=en

obviously was not taken into consideration during the platform selection process. Among the existent ARM architectures, three of them (ARMv1 to ARMv3) are already obsolete. Among the remaining architectures, ARMv7 was the state-of-the-art, and at that time, the main architecture used among the market leading ARM processors.

For the ARMv7 architecture, ARM adopted the brand name Cortex for its processors, with a supplementary letter indicating which of the three profiles the processor supports: the application processors (Cortex-A), the real-time processors (Cortex-R), and the microcontrollers (Cortex-M). Taking into consideration the second and the third requirements, my choices are, naturally, limited to the Application processors (ARMv7-A), since it is the unique family of processors that includes an MMU. The other two families of processors typically provide just a memory protection unit (MPU), which prevents the execution of GPOSes.

There are several processors that implement the ARMv7-A architecture: Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A12, Cortex-A15 and Cortex-A17. All of these Cortex-A based processors fulfill the first five requirements. Regarding the support of hardware-assisted virtualization, Cortex-A12, Cortex-A15 and Cortex-A17 distinguish from the others, by implementing the ARM VE. This technology is the specific answer from ARM for enabling hardware-assisted virtualization. Nevertheless, ARM TrustZone is also seen as a hardware-based alternative for system virtualization. Therefore, despite of the fact that Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9 do not implement ARM VE, they still provide facilities for hardware-assisted virtualization, becoming a more cost-effective alternative. Among all mentioned Cortex-A processors, the ARM Cortex-A9 is a well-established mid-range processor used in several markets, and one of ARM's most widely deployed and mature application processor.

The most difficult part during this process was to find a platform with detailed documentation about the TrustZone implementation. This security technology was under a lot of obscurity during many years. Manufacturers of TrustZone-enabled SoC appeared to be somewhat reluctant on disclosing technical details about the security extensions. What I experienced was technical support inquiries regarding the TrustZone implementation on a particular SoC, mostly end up as being completely ignored, or, in the best case, in a reply stating that information would be provided after signing a non-disclosure agreement (NDA). At that time, Xilinx, which was just launching the Zynq-7000 SoC, was the manufacturer that provided the best support. So, after signing an NDA, they sent to me the confidential release of the

document, which some years later was publicly released [2.1].

This strong obscurity around TrustZone technology clearly limited the choice for the Zynq-7000 device, featuring two (in the first devices) ARM Cortex-A9 processors. Just recently, other manufacturers (NXP, Nvidia, and Samsung) started opening the details of their implementation, but even nowadays is sometimes difficult to find public information about the TrustZone implementation for some platforms. The Zynq device also provides a wide range of security features, such as hardware support for multiple encryption standards, and secure system boot facilities. These features will be extremely helpful regarding the main goal of this thesis: the implementation of a safe and secure virtualization solution. The fact of Zynq be endowed with programmable logic is also a plus, due to the possibility of offloading some software components to hardware for achieving better determinism and performance, or either to implement some hardware mechanisms for security. Nevertheless, since security is a mutable feature, there is no panacea for all security risks. Having liquid silicon available in a platform makes possible to easily update these secure hardware components.

In sum, the Zynq-based platforms arise as the optimal choice for the development of this thesis. The processing system of the SoC is equipped with, at least, one ARM Cortex-A9 enhanced with an MMU component and TrustZone security extensions. Detailed documentation about the TrustZone implementation was provided by Xilinx, and, nowadays, it is publicly available. The fact that this processor does not support ARM VE does not necessarily mean hardware-assisted virtualization cannot be implemented. It is my belief that, when adequately exploited, TrustZone allows the implementation of a more cost-effective hardware-assisted virtualization solution.

2.2 ARM Architecture Overview

ARM processors can be found everywhere. More than 10 billion ARM processor based devices had been manufactured by the end of 2008, and at the end of 2013, over 52 billion ARM processors had been shipped [2.2]. Although ARM processors are one of the best sellers worldwide, ARM does not actually manufacture silicon devices. In fact, ARM licenses its intellectual property to semiconductor companies and original equipment manufacturers (OEMs), which then integrate them into SoC devices. This is one of the strong points for ARM: a multitude of ARM-powered

processors exist, and they vary greatly in their use and operation.

The ARM architecture is a reduced instruction set computer (RISC) architecture, as it incorporates these RISC architecture features [2.2, 2.3]:

- a large uniform register file;
- a load/store architecture, where data-related operations only operate on register contents, not directly on memory contents;
- simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only.

In addition, the ARM architecture introduces some enhancements to a basic RISC architecture to achieve a good balance of high performance, small program size, low power consumption, and small silicon area:

- instructions that combine a shift with an arithmetic or logical operation;
- auto-increment and auto-decrement addressing modes to optimize program loops;
- Load and Store Multiple instructions to maximize data throughput;
- conditional execution of many instructions to maximize execution throughput.

The ARM architecture has evolved significantly, and eight major versions of the architecture have been defined to date. The most popular central processing units (CPUs) in the market now use either the ARMv7 (32-bit, i.e. Cortex-A8, Cortex-A9, Cortex-A15, Cortex-M3) or ARMv8 (64-bit, i.e. Cortex-A53, Cortex-A57) architectures. The ARMv8 architecture was just introduced recently, and as explained in Section 2.1, at the time of starting of this thesis (Spring 2013), ARMv7-based processors were the state-of-the-art platforms. Furthermore, ARMv8 architecture is completely devoted for high-end devices, while ARMv7 covers both middle- and high-end devices, presenting a better solution to achieve scalability.

2.2.1 ARMv7-A Architecture

The ARMv7 architecture [2.3, 2.4] specifies three profiles:

- **A** - The application profile defines an architecture aimed at high performance processors, supporting a virtual memory system using a memory management

unit, and so capable of running fully featured operating systems. Support for both ARM and Thumb instruction sets is provided;

- **R** - The real-time profile defines an architecture aimed at systems that require deterministic timing and low interrupt latency. There is no support for a virtual memory system, but memory regions can be protected using a simple memory protection unit;
- **M** - The microcontroller profile defines an architecture aimed at low-cost systems, where low-latency interrupt processing is vital. It uses a different exception handling model from the other profiles, and supports only a variant of the Thumb instruction set.

The choice for the ARM application processors family is justified by the need for an MMU. The remainder of this Section describes architectural details of the ARMv7-A architecture, namely processor modes and states, core registers, and system registers.

Processor Modes

The ARMv7-A architecture has up to 9 different processor modes (depending on if optional extensions have been implemented), as summarized in Table 2.1. The current processor mode is determined by the Mode field (M) of the Current Program Status Register (CPSR). Processor mode changes can be triggered by exceptions, or by writing directly to the CPSR register in a privileged mode.

Most applications run in User mode. In this mode (privilege level 0), the memory is protected (if the CPU has an MMU or an MPU). The only way a program, running in User mode, has to change modes is to execute an SVC instruction, or even by triggering external events (such as interrupts). The other modes, i.e. System, Supervisor, FIQ, IRQ, Abort, and Undefined mode, are known as privileged modes.

Table 2.1: ARMv7-A processor modes

Processor Mode	***Privilege Level***	***Security State***	***Function***
User (*usr*)	PL0	Both	Unprivileged mode in which most applications run
System (*sys*)	PL1	Both	Privileged mode, sharing the register view with User mode
Supervisor (*svc*)	PL1	Both	Entered on reset or when a SVC is executed
FIQ (*fiq*)	PL1	Both	Entered on an FIQ interrupt exception
IRQ (*irq*)	PL1	Both	Entered on an IRQ interrupt exception
Abort (*abt*)	PL1	Both	Entered on a memory access exception
Undef (*und*)	PL1	Both	Entered when an undefined instruction is executed
Monitor (*mon*)	PL1	Secure	Implemented with Security Extensions
Hyp (*hyp*)	PL2	Non-Secure	Implemented with Virtualization Extensions

System mode is a mode that can only be entered via an instruction that specifically writes to the mode bits of the CPSR. System mode uses the User mode registers, and it is used to run tasks that require privileged access to memory and coprocessors, without limitation on which exceptions can occur during the task. It is often used for handling nested exceptions, and also by operating systems to avoid problems with nested SVC calls. Supervisor mode is a privileged mode that is entered whenever the CPU is reset or when a SVC instruction is executed. Kernels will start in Supervisor mode, configuring devices that require a privileged state, before running applications that do not require privileges. FIQ and IRQ modes are privileged modes entered whenever the processor handles an FIQ or IRQ interrupt, respectively. Abort mode is a privileged mode that is entered whenever a Prefetch Abort or Data Abort exception occurs. This means that the processor could not access some memory region for whatever reason. Undefined mode is a privileged mode that is entered whenever an Undefined Instruction exception occurs. This normally happens when the ARM core is fetching instructions in the wrong place (corrupted PC), or if the memory itself is corrupted.

The optional security extensions, referred to as TrustZone, introduce a new processor mode called Monitor. The Monitor mode is a privileged mode different from the other privileged modes, because it is only available in the secure state. The ARM VE are also optional extensions to the ARMv7-A architecture profile. These extensions introduce a higher privilege mode of execution (privilege level 2) called Hypervisor mode. However, since this mode exists only for the non-secure state, it is, in practice, less privileged than the secure Monitor mode. Nevertheless, Monitor mode is classified with privilege level 1. This classification of the privilege level of the processor modes was then reviewed by ARM with ARMv8 architecture, where Hypervisor mode has an exception level of 2 (EL2) and the Monitor mode an exception level of 3 (EL3) [2.5].

Processor States

The introduction of the TrustZone security extensions (for more details please refer to Section 2.3) created two security states for the processor, that are independent of the privilege and processor mode. The new Monitor mode acts as a gateway between the secure and non-secure states, where modes exist independently for each security state. This distinction between the two states is completely orthogonal to the privilege level of the execution mode. As explained above, the Monitor mode

is only available in the secure state and the Hypervisor mode only exists for the non-secure state. Despite the Hypervisor mode having a classification of PL2, the Monitor mode, classified as PL1, is, in fact, more privileged than the Hypervisor mode. The *Non-Secure* (*NS*) bit, accessible through the *Secure Configuration Register* (*SCR*), indicates in which world the processor is currently executing. In the Monitor mode, the processor is always secure, independently of the state of the NS bit.

Core Registers

The ARMv7-A architecture provides sixteen 32-bit general purpose registers (R0-R15). In fact there are several more, because some registers are mode-specific, which means they are banked. Registers R0 to R7 are the same across all CPU modes, and they are never banked. Registers R8 to R12 are the same across all CPU modes, except for FIQ mode. This means, for example, when a processor is executing in the FIQ mode, R0 refers to R0_usr, but R12 refers to R12_fiq instead of R12_usr. R13 and R14 are unique to each mode and do not need to be saved (except for System and User modes). Table 2.2 depicts the system level view of the ARMv7 core registers.

According to the ARM coding conventions (the AAPCS, Procedure Call Standard for the ARM Architecture), the first four registers, R0 to R3, are used to pass

Table 2.2: ARMv7-A core registers

System Level View *Core Registers*								
usr	*sys*	*svc*	*fiq*	*irq*	*abt*	*und*	*mon*	*hyp*
R0	-	-	-	-	-	-	-	-
R1	-	-	-	-	-	-	-	-
R2	-	-	-	-	-	-	-	-
R3	-	-	-	-	-	-	-	-
R4	-	-	-	-	-	-	-	-
R5	-	-	-	-	-	-	-	-
R6	-	-	-	-	-	-	-	-
R7	-	-	-	-	-	-	-	-
R8	-	-	R8_fiq	-	-	-	-	-
R9	-	-	R9_fiq	-	-	-	-	-
R10	-	-	R10_fiq	-	-	-	-	-
R11	-	-	R11_fiq	-	-	-	-	-
R12	-	-	R12_fiq	-	-	-	-	-
R13 (sp)	-	SP_svc	SP_fiq	SP_irq	SP_abt	SP_und	SP_mon	SP_hyp
R14 (lr)	-	LR_svc	LR_fiq	LR_irq	LR_abt	LR_und	LR_mon	LR_hyp
R15 (pc)	-	-	-	-	-	-	-	-
(A/C)PSR	-	-	-	-	-	-	-	-
N/A	N/A	SPSR_svc	SPSR_fiq	SPSR_irq	SPSR_abt	SPSR_und	SPSR_mon	SPSR_hyp
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	ELR_hyp

arguments to a function, as well as to return values (i.e, caller-saved registers). R4 to R12 are general purpose registers and can be used for any calculation. R13 has a special function: it is the stack pointer. Just like the other registers, it is possible to read and write to this register, but most dedicated instructions will change the stack pointer as required. R14 holds the value of the Link Register, the memory address of an instruction to be run when a subroutine has been completed. R15 holds the value of the Program Counter, the memory address of the next instruction to be fetched from memory.

The Current Program Status Register (CPSR) is a register somewhat different from R0 to R15. The CPSR is a critical register that holds condition code flags (e.g., zero, carry, overflow) as well as the current processor mode and other critical configurations of the processor. It is updated continuously, specially when compare instructions are executed. If the CPSR is the Current PSR, the SPSR is the Saved PSR. When an ARM processor responds to an event that generates an exception, the CPSR is saved into the SPSR of the corresponding mode. Each mode can have its own CPSR, and when the exception has been handled, the SPSR is restored into the CPSR, and program execution can continue.

Coprocessors and System Registers

The ARM architecture supports a way of extending the instruction set by using Coprocessors. The ARM architecture supports sixteen coprocessors, namely CP0 - CP15. On the Cortex-A series processors, only internal coprocessors are supported, including the CP15 for the control and configuration of the processor system, the CP14 for debug, and CP10 and CP11 for NEON and VFP operations, respectively. Hardware manufacturers can also define other coprocessors for their own purposes.

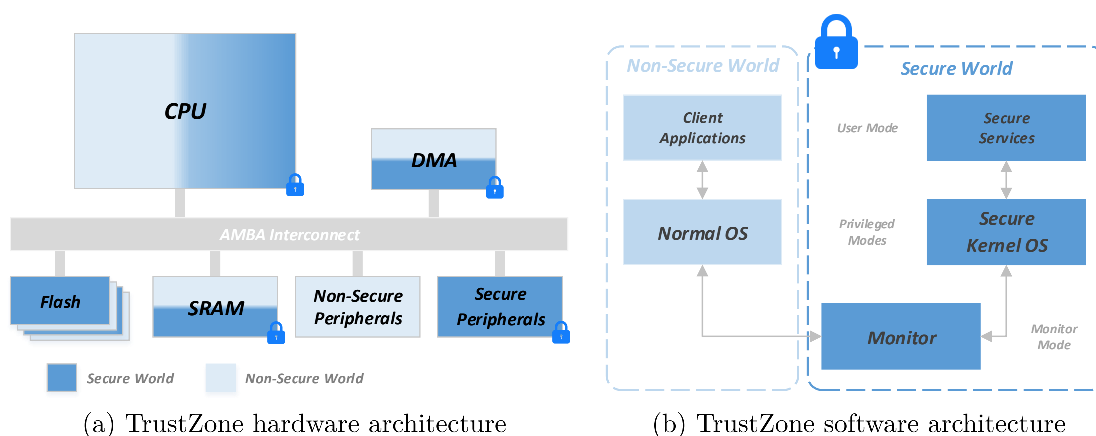
The System Control coprocessor, provides control of many features of the core, including (1) system control and access registers (SCTLR, ACTLR, SCR), (2) memory protection and control registers (TTBR0, TTBR1, TTBCR), (3) memory system fault registers (DFSR, IFSR), (4) cache and MMU maintenance operations, (5) security extensions registers, (6) process, context and thread ID registers, and others. Given the special purpose of CP15 system registers, many of them are banked between secure and non-secure states. However, the registers that configure the global system status, such as SCTLR and SCR, are not banked. Table 2.3 lists some of the most important CP15 system registers. A complete enumeration and a detailed description of this coprocessor registers can be found in [2.3] and [2.6].

Table 2.3: CP15 register summary

Name	*Register*	*Security*	*Description*
CP15 c1 System Control registers			
System Control Register	SCTLR	Banked	The main processor control register
Auxiliary Control Register	ACTLR	Secure	Implementation specific configuration options
Coprocessor Access Control Register	CPACR	Secure	Controls access to coprocessors
Secure Configuration Register	SCR	Secure	Configuration of the current security state
Secure Debug Enable Register	SDER	Secure	Controls processor debug
Non-secure Access Control Register	NSACR	RO NS	Sets the NS access permission for coprocessors
CP15 c2 and c3, memory protection and control registers			
Translation Table Base Register 0	TTBR0	Banked	Base address of level 1 translation table
Translation Table Base Register 1	TTBR1	Banked	Base address of level 1 translation table
Translation Table Base Control Register	TTBCR	Banked	Controls the use of TTBR0 and TTBR1
Domain Access Control Register	DACR	Banked	Control memory access permissions
CP15 c12, Security Extensions registers			
Vector Base Address Register	VBAR	Banked	Base address of secure vector table
Monitor Vector Base Address Register	MVBAR	Secure	Base address of monitor vector table

2.3 TrustZone: The ARM Security Extensions

TrustZone technology [2.7] refers to the security extensions introduced with ARMv6K in all ARM Cortex-A processors. The TrustZone hardware architecture can be seen as a dual-virtual system, partitioning all system's physical resources into two isolated execution environments (Figure 2.1a). Recently, ARM also decided to extend TrustZone for the Cortex-M processor family [2.8]. TrustZone for ARMv8-M has the same high-level features as TrustZone for applications processors, with the benefit that context-switching between both worlds is done in hardware for faster transitions. In the remainder of this Section, when describing TrustZone, I am focused on the specificities of this technology for Cortex-A processors. The distinctive aspects of TrustZone for ARMv8-M are out of the scope of this thesis.



(a) TrustZone hardware architecture (b) TrustZone software architecture

Figure 2.1: ARM TrustZone

2.3.1 TrustZone: Hardware Component

At the processor level, the most significant architectural change is its partition into two separate worlds: the secure and the non-secure worlds. A new 33rd processor bit, the *NS* bit, accessible through the *Secure Configuration Register*, indicates in which world the processor is currently executing, and is propagated over the memory and peripherals buses. To preserve the processor state during the world switch, TrustZone adds an extra processor mode: the monitor mode. The monitor mode is completely different from other supported modes, because when the processor runs in this mode the state is always considered secure, independently of the NS bit state. Software stacks in the two worlds can be bridged via a new privileged instruction - *Secure Monitor Call* (*SMC*). The monitor mode can also be entered by configuring it to handle IRQ, FIQ and Aborts exceptions in the secure world. To guarantee a strong isolation between secure and non-secure states, some special registers are banked, such as several *System Control Coprocessor* (*CP15*) registers. Some secure critical processor core bits and *CP15* registers are either totally unavailable to non-secure world or access permissions are closely under supervision of the secure world. To provide the exception behavior described above, TrustZone specifies three sets of exception vector tables - one for the non-secure world, one for the secure world, and another for the monitor mode.

The TrustZone Address Space Controller (TZASC) extends TrustZone security to the memory infrastructure. The TZASC can partition the off-chip RAM (DRAM) into different memory regions: this hardware controller has a programming interface, accessible only from the secure side, that can be used to configure a specific memory region as secure or non-secure. By default, secure world applications can access non-secure world memory but the reverse is not possible. The number of memory regions, and the bus widths of the TZASC interfaces, are configurable when the design is synthesized by each vendor. The TrustZone Memory Adapter (TZMA) provides similar functionality but for on-chip memory. This means it enables a design to secure a region within an on-chip static memory such as a ROM or an SRAM. The TZMA allows a single static memory of up to 2MB to be partitioned into two regions, where the lower part is secure, and the upper part non-secure. The location of the partition between the secure and non-secure regions is always a multiple of 4KB. The TZMA cannot be used for partitioning dynamic memories, or memories that require more than one secure region. In this specific case, the TZASC must be used. The TrustZone-aware MMU provides two distinct MMU interfaces, enabling each

world to have a local set of virtual-to-physical memory address translation tables. The isolation is still available at the cache-level, because processor's caches have been extended with an additional tag bit which signals in which state the processor accesses the memory.

System devices can be dynamically configured as secure or non-secure through the TrustZone Protection Controller (TZPC). The TZPC is a configurable signal control block which can be placed on the Advanced Peripheral Bus (APB) to supply control signals to other components on the SoC. The direct memory access (DMA) controller is a dedicated engine used for moving data around the physical memory system, instead of using the processor to perform this task. The DMA can support concurrent secure and non-secure channels, each with independent interrupt events and controlled by a dedicated APB interface. A non-secure transaction trying to program a DMA transfer to or from secure memory will result in the DMA transfer failing.

To support the robust management of secure and non-secure interrupts, the Generic Interrupt Controller (GIC) provides both secure and non-secure prioritized interrupt sources. An interrupt can be configured as a secure interrupt through the Interrupt Security Register. In addition, the interrupt controller supports interrupt prioritization, allowing the configuration of secure interrupts with a higher priority than the non-secure interrupts. Such configurability prevents non-secure software to perform a denial-of-service attack against the secure side. The GIC also supports several interrupt models, allowing for the configuration of IRQs and FIQs as secure or non-secure interrupt sources. The suggested model by ARM proposes the use of IRQs as non-secure world interrupt sources, and FIQs as secure interrupt sources.

2.3.2 TrustZone: Software Component

There are many possible software architectures which a software stack on a TrustZone-enabled processor core could implement. Figure 2.1b depicts the generic software architecture, where software components are distributed between both worlds. Adopting a bottom-up description, the software running in the secure world is composed by the Monitor layer, the secure kernel and its corresponding service tasks. The Monitor component, running in Monitor mode, provides a robust gatekeeper which manages the switches between the Secure and Non-secure processor states. The secure kernel, running in privileged mode, provides facilities for the concurrent execution of multiple independent secure services (running in a non-privileged mode). The soft-

ware running in the non-secure world side, in turn, consists of a general-purpose operating system with the corresponding TrustZone API-dependent software and the client applications.

The TrustZone API (TZAPI) [2.9] is an application programming interface which specifies how normal applications running on the rich OS interact with the isolated execution environment. Basically, following a client-server model, the API defines a set of abstract software interfaces through which non-secure client applications (NSCApps) can interact with the secure services. The API allows clients to send commands and requests to a secure service, and exchanges data between both worlds. Secondary features of the API allow, for example, to query the properties of installed services as well as download new security services at run-time. The (publicly available) TrustZone API does not include any specification about how to develop applications running inside the isolated execution environment. Hence, while it could be useful for application developers, by itself it does not fully specify the APIs needed for developing secure services.

2.4 The Zynq Device

The Zynq-7000 family is based on the Xilinx All Programmable SoC (AP SoC) architecture, which integrates a feature-rich single or dual-core ARM Cortex-A9 based processing system (PS) and Xilinx programmable logic (PL) in a single device [2.10]. A block diagram depicting the Zynq-7000 AP SoC architecture is presented in Figure 2.2.

All Zynq devices have the same basic architecture, and all of them contain, as the basis of the processing system, at least one ARM Cortex-A9 processor. This is a "hard" processor, which means it exists as a silicon element on the device. The ARM Cortex-A9 CPU(s) is/are the heart of the PS, but the Zynq processing system encompasses also a set of associated computational units forming an application processing unit (APU), as well as further peripheral interfaces, cache memory, and memory interfaces. The APU is endowed with one or two ARM processing cores, each with associated computational units such as a NEON engine and floating-point unit (FPU), an MMU, and a Level 1 data and instruction cache (both of which are 32KB). The APU also contains a Level 2 cache of 512KB for instructions and data, and there is a further 256KB of on-chip memory within the APU.

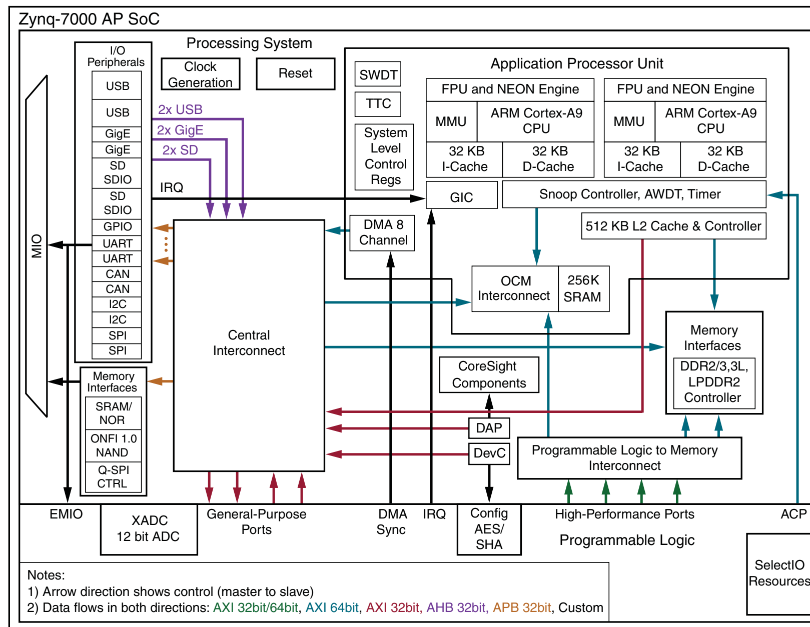


Figure 2.2: Zynq-7000 SoC overview

The second principal part of the Zynq architecture is the programmable logic. It is based on the Artix-7 and Kintex-7 field-programmable gate array (FPGA) fabric, depending on the specific device of Zynq family (please refer Section 2.4.2 for more details). The PL is predominantly composed of general purpose FPGA logic fabric, which is composed of slices (flip-flops, LUTs, and other logic) and configurable logic blocks, input/output blocks for interfacing, and other special resources such as block RAMs.

Interactions between the PS and the PL are supported through a set of nine AXI interfaces, each of which is composed of multiple channels. The current version is AXI4, which is part of the ARM AMBA 3.0 open standard. There are different roles between the several types of PS-PL AXI interfaces:

- ***General Purpose AXI*** - A 32-bit data bus, which is suitable for low and medium rate communications between the PL and PS. The interface is direct and does not include buffering. There are four general purpose interfaces in total: the PS is the master of two, and the PL is the master of the other two;
- ***High Performance Ports*** - The four high performance AXI interfaces include FIFO buffers to accommodate burst transactions, and support high rate communications between the PL and the PS. The data bus can support 32 or 64 bits, and the PL is the master of all four interfaces;
- ***Accelerator Coherency Port*** - There is a single asynchronous connection

between the PL and the Snoop Control Unit (SCU) within the APU, with a bus width of 64 bits. This port is used to achieve coherency between the APU caches and elements within the PL. The PL is the master.

2.4.1 Security

Zynq-7000 devices provide a wide range of security features which offer protection of the internal functionality of the system, ranging from dedicated hardware support for multiple encryption standards, secure system boot facilities, and software execution protection. The remainder of this Section briefly introduces the security features provided by Zynq devices.

Hardware Support

Zynq-7000 devices have a number of embedded blocks which can support the creation of secure systems. The functionality of these security IPs includes anti-tamper, trust and information assurance, to protect the system from power-on and through runtime. These blocks include authentication, decryption engines, key storage and unique device identification possibilities. Some of the features of Zynq devices which relate to security are listed as follows [2.11, 2.12]:

- ARM TrustZone support (PS and PL);
- Secure configuration and boot (PS and PL);
- AES-256 encryption (BBRAM key and eFUSE key);
- HMAC bitstream authentication;
- First stage boot loader (FSBL) RSA-2048 authentication;
- JTAG disable/monitor.

Secure Boot

Booting a device securely starts with the BootROM code loading the FSBL, and continues serially with the FSBL loading the bitstream and software. With a secure boot foundation established by the BootROM code, the chain of trust is created by the successive authentication of all software loaded into the device. This prevents

an adversary from tampering with software or the bitstream file.

Several security-related features have been incorporated into Zynq-7000 devices, which facilitate the secure booting process [2.11]. One of these features is the BootROM, which has been designed to handle various forms of security. Both asymmetric and symmetric authentication of the FSBL, U-Boot, PL bitstream and software are supported. In the case of asymmetric authentication, RSA-2048 primary and secondary public keys are used, whereas HMAC (SHA-256) is used for symmetric authentication. Further, encryption of the boot files mentioned above is supported with 256-bit AES/CBC key which can be either volatile (battery backed RAM) or non-volatile (eFUSES). Another feature which facilitates the secure boot is the on-chip memory (OCM), which has been provided to be large enough (256KB) to run the FSBL from an internal location which is immune to any external probing attack. The OCM is also large enough to securely store TrustZone software routines.

Runtime Security

The need for preventing unwanted access to the internal device data or memory does not end after the boot process has completed, and, obviously, there is a need to provide runtime security.

One feature of Zynq devices which can prevent such vulnerabilities is the Zynq-specific implementation of ARM TrustZone technology [2.13, 2.1]. As previously explained, the Zynq-7000 SoC is divided into two domains: a processing system and a programmable logic domain. The Zynq-7000 AP SoC supports ARM TrustZone technology in both the PS and PL domains of the device. The PS provides a set of configuration registers related to TrustZone support for all hard custom blocks. These configuration registers can be dynamically programmed during software execution. Table 2.4 summarizes the TrustZone security for the hardware components in the Zynq-7000 PS. In the PL, a security-checking feature is provided for each master interface slot in the AXI interconnect IP. A static secure or non-secure status can be assigned to an AXI interconnect master interface slot. All slave IP cores instantiated in the logic can also be individually assigned a secure or non-secure designation. For Xilinx slave IP cores, secure/non-secure configuration can be designated also at the AXI interconnect level.

Table 2.4: Zynq-7000 TrustZone security summary

PS Entity	***TrustZone Security***	***Notes***
ARM A9 Core	Both	-
L1 Cache Controller	Secure	-
L1 Cache	Both	-
MMU	Both	-
L2 Cache Controller	Secure	-
L2 Cache	Both	-
Triple Timer-Counter0	Secure	-
Triple Timer-Counter1	Configurable	-
Watch Dog	Secure	-
SoC CoreSight Debug	Secure	-
OCM	Secure and Non-secure	256KB RAM can be divided into 4KB segments
DDR memory	Secure and Non-secure	Divided into 64MB segments
IOU Devices	Configurable	I2C, GPIO, SPI, Ethernet, SDIO, CAN, USB and UART, Quad-SPI, NOR

2.4.2 Zynq-7000 Family

At the time of writing this thesis, the Zynq product range comprises ten different general purpose Zynq-7000 devices, all with slightly different features and sizes. The features of these devices are summarized in Table 2.5, but an extended description can be found in [2.14].

As depicted in the Table 2.5, the main differences between the specific devices within the Zynq family is the parallel processing capability of the processing system, as well as the type and quantity of the programmable logic. Among the Zynq family members, the smaller devices (i.e., cost-optimized devices) are based on the Xilinx Artix-7 FPGA logic fabric while the larger devices (i.e., mid-range devices) on the Kintex-7 logic fabric. Each of the ten family members provides a different amount of general purpose logic, Block RAMs, digital signal processor (DSP) slices, and naturally the overall processing capability of the PL section increases in proportion

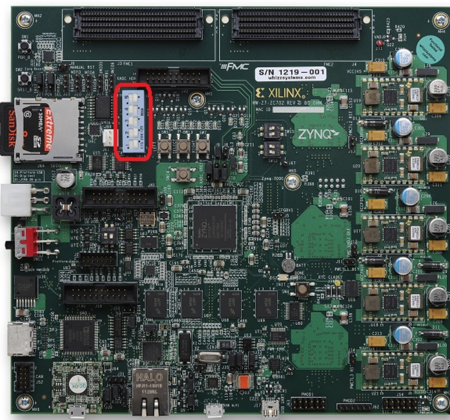
Table 2.5: Zynq-7000 family members

	Cost-Optimized Devices						***Mid-Range Devices***			
Device	***Z-7007S***	***Z-7012S***	***Z-7014S***	***Z-7010***	***Z-7015***	***Z-7020***	***Z-7030***	***Z-7035***	***Z-7045***	***Z-7100***
Part	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020	XC7Z030	XC7Z035	XC7Z045	XC7Z100
Core	***Single-Core ARM Cortex-A9 SCore*** Up to 766MHz			***Dual-Core ARM Cortex-A9 MPCore*** Up to 866MHz			***Dual-Core ARM Cortex-A9 MPCore*** Up to 1GHz			
L1 Cache	32KB Instruction, 32KB Data per processor									
L2 Cache	512KB									
Memory	256KB - on-chip memory DDR3, DDR3L, DDR2, LPDDR2 - external memory support 2x Quad-SPI, NAND, NOR - external static memory support									
PL	***Artix-7***			***Artix-7***			***Kintex-7***			
Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
LUTs	14400	34400	40600	17600	46200	53200	78600	171900	218600	277400
Flip-Flops	28800	68800	81200	35200	92400	106400	157200	343800	437200	554800
Block RAM	1.8Mb	2.5Mb	3.8Mb	2.1Mb	3.3Mb	4.9Mb	9.3Mb	17.6Mb	19.1Mb	26.5Mb
(36Kb Blocks)	(50)	(72)	(107)	(60)	(95)	(140)	(265)	(500)	(545)	(755)
DSP Slices	66	120	170	80	160	220	400	900	900	2020

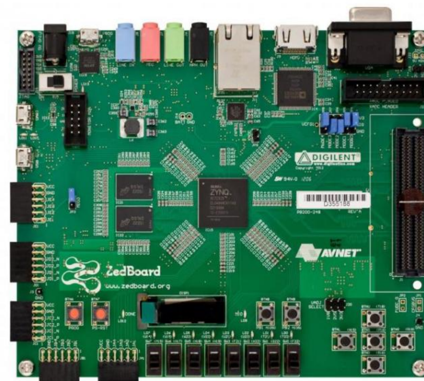
to its resources. The PS varies in the multiprocessing support: small devices are endowed with a single-core ARM Cortex-A9 while the others with a dual-core ARM Cortex-A9. The maximum frequency of the ARM core is also different: the PS on the Artix-7 based devices can be clocked at up to 766MHz or 866MHz, and the Kintex-based devices up to 1GHz.

2.4.3 Development Boards

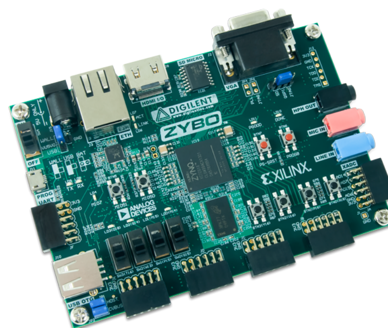
There are several development boards available for Zynq. Some of them are evaluation kits while others are community-based and commercial boards. This Section is dedicated to provide an overview of those three Zynq-based boards that were, in fact, used during the time of realization of this thesis. Figure 2.3 presents those platforms.



(a) ZC702



(b) ZedBoard



(c) ZYBO

Figure 2.3: Zynq-based platforms

ZC702

The ZC702 evaluation board for the XC7Z020 AP SoC provides a hardware environment for developing and evaluating designs targeting the Zynq XC7Z020-1CLG484C device. The ZC702 board interfaces a 16MB flash memory and 1GB DDR3 memory. There are a number of peripheral interfaces on the ZC702: general purpose inputs/outputs (GPIOs), HDMI video, Ethernet, USB-OTG (peripherals), USB-JTAG (programming), and USB-UART (communication), SD card slot, FPGA Mezzanine Card (FMC) interface, and Xilinx JTAG header. Figure 2.3a depicts the upper layer of ZC702 platform.

ZedBoard

The ZedBoard is a low-cost, community-based board which features a XC7Z020 Zynq device. It is a joint venture between Xilinx, Avnet (the distributor), and Digilent (the board manufacturer). The ZedBoard features a ZC7Z020 Zynq device. The Zynq device interfaces a 256Mbit flash memory and 512MB DDR3 memory. There are diverse peripheral interfaces on the ZedBoard: general purpose I/O, HDMI and VGA video, Ethernet, USB-OTG (peripherals), USB-JTAG (programming), and USB-UART (communication), SD card slot, FMC interface, and Xilinx JTAG header. Figure 2.3b depicts the upper layer of ZedBoard platform.

ZYBO

The ZYBO (diminutive of Zynq Board) is an ultra-low cost alternative to the ZedBoard featuring the smallest Zynq device, the Z-7010, which is based on the Artix-7 PL fabric. It is aimed at designers looking to get started developing for Zynq but who do not have a requirement for the high density I/O or the FMC connector present in mid-level and above boards. Figure 2.3c demonstrates how the ZYBO manages to include memory, video and audio I/O, Ethernet, and several GPIO and more on a compact board.

2.5 Operating System Stacks

With the increasing complexity of today's systems, applications are demanding a broader consolidation of different functionalities. Therefore, it is hard to find an embedded system without an OS included in its software stack [2.15]. Operating systems tend to alleviate the complexity of embedded systems development by providing several different mechanisms, such as multithreading, semaphores, timers, and interrupt handling, in order to abstract, simplify and coordinate the overall system behavior. While the number of general-purpose operating systems for PCs and server-like computers has undergone a strong consolidation over the last two decades (eventually resulting in Windows, Linux, and MacOS), embedded application developers can select from a plethora of available operating systems, most of which are real-time operating systems. This thesis targets the development of a secure and safe virtualization solution. This solution implicitly requires OSes to run on top of the virtualization stack. This Section provides a quick look over the OS stacks used during the development of this thesis, providing concrete arguments and justifications behind their choice.

2.5.1 Real-Time Operating Systems

A real-time system is a computer system that requires not only that the computing results are correct, but also that the results are produced within a specific deadline. Results produced after the deadline may have no real value, and can even result in catastrophic consequences. Real-time systems are classified according to two different types: hard and soft. A hard real-time system has the most stringent timing requirements, guaranteeing that critical real-time tasks are completed within their deadlines. Safety-critical systems are typically hard real-time systems. A soft real-time system is less restrictive, simply ensuring that a critical real-time task will receive priority over other tasks and that it will retain that priority until it completes.

A real-time operating system (RTOS) is an operating system intended to serve real-time applications, guaranteeing, therefore, a certain behavior within a specified time constraint. The main difference between an RTOS to general-purpose OSes is the response time to external events. An RTOS has to guarantee a real time response, providing a fast, highly deterministic reaction to external events. When switching

between tasks, the RTOS has to choose the most appropriate task to load next. There are several scheduling algorithms available, including round-robin and co-operative scheduling. However, to provide a responsive system most RTOSes use a preemptive scheduling algorithm.

A wide variety of RTOSes are available to suit most embedded applications. The process of selecting an RTOS is very complex and hindered by several factors. According to Express Logic[2], the most valued features are (ordered by relevance): real-time responsiveness, royalty-free pricing, source code availability, tools integration (IDE) and microprocessor coverage.

FreeRTOS

FreeRTOS[3] is an RTOS designed to be deployed on embedded systems with scarce resources. It is characterized by a very simple and small kernel core, written mostly in C, presenting a software architecture divided into two main layers: the "hardware independent" and the "portable" layer. The former is responsible for performing processor independent functions and is maintained intact for all architectures, while the second implements some architecture-specific routines (e.g. context-switching).

The FreeRTOS source structure is very small: the core of the RTOS kernel is contained in only three C files. The tasks.c file provides a set of task management functionalities, including the scheduler component. FreeRTOS implements a preemptive priority-based scheduler policy, which privileges the execution of the highest priority tasks. For tasks with the same priority, the scheduler follows a round-robin model. In addition, the list.c file implements a list data structure for maintaining task queues (ready, waiting and running). These two files, altogether with the port-specific code, implement the minimum core kernel high-level functionalities. The optional file queue.c implements a list of queues used for inter-task communication and synchronization. The timers.c file offers a set of functions to implement software timers used by application tasks. The operating system features also a special type of tasks, called "co-routines", that present high memory efficiency. Those kinds of tasks are implemented within croutine.c file. The port.c file contains not only the hardware-specific code, but also the standard API of the OS. At last, the heap.c file provides the memory allocation and deallocation functionality, specific to the target architecture.

[2]http://rtos.com/PDFs/MeasuringRTOSPerformance.pdf
[3]http://www.freertos.org/

Among the extensive list of existing RTOSes, FreeRTOS is a very interesting choice for several reasons. First, FreeRTOS is open-source, which allows an internal re-design when needed. The kernel core is simple and small, allowing to perform the necessary changes without a huge engineering effort. Finally, it is widely used and a market leading RTOS, due to the large number of supported architectures: FreeRTOS is considered the de-facto standard solution for microcontrollers and small microprocessors.

RODOS

Real-time onboard dependable operating system (RODOS) [2.16] is an RTOS for embedded systems and was designed for application domains demanding high dependability. RODOS was developed at the German Aerospace Center, and further enhanced and extended at the department for aerospace information technology at the University of Würzburg. It is used for the current micro satellite program of the German Aerospace Center. The system runs on the operational satellite TET-1 and will be used for the satellite BiROS.

RODOS was designed for application domains demanding high dependability (e.g., space) and targets the irreducible complexity in all implemented functions. An important aspect in the selection of RODOS is its integrated real-time middleware. Developing the control and payload software on the top of a middleware provides a high level of modularity. Applications/modules can be developed independently and it is very simple to interchange modules without worrying about side effects, because all modules are encapsulated as building blocks (BB) and they can access other resources only by well-defined interfaces.

RODOS was implemented as a software framework following an object-oriented approach (C++). It is organized in layers: the lowest layer (1) is responsible for managing the embedded system hardware (hardware abstraction layer); the next layer (2), kernel, administrates the local resources, threads and time. On top of the kernel is located the middleware (layer 3) which enables communication between BBs using a publisher-subscriber multicast protocol. Finally on the top of the middleware the users may implement their applications (layer 4) as a distributed software network of simple BBs.

RODOS was developed at the German Aerospace Center by Dr. Sergio Montenegro, and has been extended and maintained by the Aerospace Information Technology

research group, led by Dr. Montenegro, at the University of Würzburg. Part of this thesis was done in cooperation with them, when I was a visiting PhD student in Germany. RODOS emerged as an obvious and natural choice while developing my virtualization solution for aerospace, once it arises as an "in-house" RTOS solution that fulfills the safety-critical requirements of aerospace industry.

2.5.2 General-Purpose Operating Systems

A general-purpose operating system is a fully-featured operating system intended to provide a better user experience. These OSes are designed to optimize average performance of application programs at the expense of predictability. OSes typically provide a non-deterministic response, where there are no guarantees as to when each task will complete, but they will try to stay responsive to the user. Examples of GPOSes include the well-known Windows, Linux and MacOS.

Linux

From smartphones to cars, from desktop and server computers to home appliances, the Linux operating system is everywhere. Linux is a Unix flavor operating system assembled under the model of free and open-source software development and distribution. It is a general-purpose operating system originally developed for personal computers based on the Intel x86 architecture, but has been ported to a multitude of mainly MMU-enabled platforms, and has been used on ARM-based platforms for decades. Linux implements a monolithic kernel, which means it handles process management, networking, access to the peripherals, and file systems in kernel space. Device drivers are either integrated directly into the kernel, or added as modules that are loaded while the system is running.

Linux has a huge user base and support community, and the possibility of compiling the kernel is a major advantage. When adding new hardware, there are lots of resources necessary for adding drivers, and it is possible that in the open-source community someone has already developed such driver. Different Linux distributions have been ported to several Zynq-based platforms. For example, the Xilinx Linux distribution, the Linaro Linux distribution, and Digilent Linux distribution. All of them have support for ZC702, Zedboard and ZYBO platforms.

2.6 Benchmarks

Benchmarking and performance analysis is not a new endeavor. It is a well-established method of comparing the performance of various subsystems across different processors and system architectures. Several benchmark suites exist targeting different metrics, systems and domains. This Section provides a quick look over the Benchmark suites used during the development of this thesis, explaining the main reasons behind their choice.

2.6.1 Thread-Metric

The Thread-Metric benchmark suite[4], from Express Logic, is a freely-available set of benchmarks that measures many aspects of RTOS performance. Criteria such as interrupt response, context-switching, message passing, thread scheduling, memory allocation, and synchronization are particularly important when evaluating an RTOS. To be applicable to multiple RTOSes, for comparison, a set of common services has been selected; it encompasses seven benchmarks: cooperative scheduling, preemptive scheduling, interrupt processing, interrupt preemption processing, synchronization processing, message processing, and memory allocation. Each benchmark outputs a counter value, representing the RTOS impact on the running application: the higher the value, the smaller the impact.

The number of benchmarks available for evaluating the RTOS overhead/performance is scarce. Thread-Metric has been widely used across academia and industry. It has the advantage of being freely available and made open-source by Express Logic. It is also easily adapted to other RTOSes, just by mapping the generic APIs into the RTOS-specific APIs. No special hardware is required, and the code was tested with various compilers.

2.6.2 LMBench

LMBench [2.17] is a widely used suite of micro-benchmarks that measure a variety of important aspects of system performance, such as latency and bandwidth. The timing harness is the heart of the system, because it manages the benchmarking process: starting the benchmarked activity, repeating the benchmarked activity as

[4]http://rtos.com/PDFs/MeasuringRTOSPerformance.pdf

long as necessary to ensure accurate results, and finally managing statistics to report representative results. The suite is written in portable ANSI-C using POSIX interfaces and targeting UNIX systems.

The LMBench 3.0 suite includes more than forty micro-benchmarks within three different categories:

- *bandwidth* - file read, memory read/write/copy, memory map, and others;
- *latency* - memory latency, inter-process communication using Transmission Control Protocol (TCP), User Datagram Protocol (UDP), pipe and unix sockets, file creation and deletion, arithmetic operations, and others;
- *other* - CPU clock speed, translation lookaside buffer (TLB) size, cache line size, arithmetic operations parallelism, memory parallelism, and others.

The number of available benchmarks for GPOSes is generous and diverse, namely targeting different architectural components. LMBench provides a plethora of micro-benchmarks, in the same suite, ranging from computing intensive (e.g., arithmetic operations) to memory, communication and I/O intensive tests. Its availability as an open-source tool, as well as its widespread in Unix platforms, make it an attractive option compared to other benchmark suites. The benchmarks are all in C, and so, fairly portable. The source is small and easy to extend.

2.7 Summary

The goal of this Chapter was to describe the research platform and tools used during the development of this thesis. I started by presenting the identified requirements and justifying the selection of a Zynq-based platform, endowed with, at least, one ARM Cortex-A9 with TrustZone support. In doing so, I then described several architectural aspects of the ARMv7-A architecture, as well as the hardware and software architecture of the ARM TrustZone technology. Several aspects of Zynq devices were then reviewed, describing the general architecture, highlighting the provided security facilities, comparing the several devices of the Zynq-7000 family, and finally describing the three used Zynq-based platforms: ZC702, Zedboard, and ZYBO. FreeRTOS, RODOS and Linux were chosen as operating systems on the system stacks, and Thread-Metric and LMbench as benchmarks suites. I highlighted the several reasons behind the selection of such tools.

References

[2.1] Xilinx, "Programming ARM TrustZone Architecture on the Xilinx Zynq-7000 All Programmable SoC." UG1019 (v1.0), May 2014.

[2.2] A. Sloss, D. Symes, and C. Wright, *ARM system developer's guide: designing and optimizing system software*. Morgan Kaufmann, 2004.

[2.3] ARM, "ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition." ARM DDI 0406C.b (ID072512), July 2012.

[2.4] ARM, "ARM Cortex-A Series: Programmer's Guide." ARM DEN0013D (ID012214), January 2014.

[2.5] ARM, "ARM Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile." ARM DDI 0487A.a (ID090413), September 2013.

[2.6] ARM, "Cortex-A9: Technical Reference Manual." ARM DDI 0388E (ID113009), November 2009.

[2.7] ARM, "ARM Security Technology: Building a Secure System using TrustZone Technology." PRD29-GENC-009492C, April 2009.

[2.8] J. Taylor, "Security for the next generation of safe real-time systems," in *Proceedings of Embedded World Conference, Nuremberg, Germany*, March 2016.

[2.9] ARM, "TrustZone API Specification." PRD29-USGC-000089 (3.1), February 2009.

[2.10] Xilinx, "Zynq-7000 All Programmable SoC: Technical Reference Manual." UG585 (v1.11), September 2017.

[2.11] L. Sanders, "Secure Boot of Zynq-7000 All Programmable SoC." XAPP1175 (v2.0), April 2015.

[2.12] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[2.13] Y. Gosain and P. Palanichamy, "TrustZone Technology Support in Zynq-7000 All Programmable SoCs." WP429 (v1.0), May 2014.

[2.14] Xilinx, "Zynq-7000 All Programmable SoC Overview." DS190 (v1.10),

September 2016.

[2.15] T. Gomes, "Multithreading RTOS processor design," *PhD Thesis, Universidade do Minho*, 2015.

[2.16] S. Montenegro and F. Dannemann, "RODOS-real time kernel design for dependability," in *ESA Special Publication*, vol. 669, p. 66, 2009.

[2.17] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pp. 23–23, USENIX Association, 1996.

"We believe that ARM TrustZone opens up a number of opportunities for securing ICSs and that now is the time to research the applicability of this technology."

- Johannes Winter

3

LTZVisor: TrustZone is the Key

Platform virtualization, which enables multiple operating systems to run on top of the same hardware platform, is gaining momentum in the embedded systems arena, driven by the growing interest in consolidating and isolating multiple and heterogeneous environments [3.1]. While in industrial control or automotive systems virtualization has been used to integrate real-time control functionality with high-level or infotainment environments [3.2, 3.3], in aeronautics and aerospace virtualization provides isolation for safety-critical components [3.4, 3.5]. Despite the differences among several embedded industries, they all share an upward trend for integration, towards the common interest in building systems with reduced size, weight, power and cost (SWaP-C) budget [3.1, 3.4].

The penalties incurred by standard software-based embedded virtualization [3.4, 3.5, 3.6], altogether with the top-level requirements (e.g., performance, memory, power, safety, security) that drive the development of current embedded devices, are pushing academia and industry to focus on the development of hardware-assisted solutions [3.7, 3.8, 3.9, 3.10]. Among the existing commercial off-the-shelf technologies for secure virtualization, ARM TrustZone is attracting particular attention [3.11, 3.12, 3.9, 3.13, 3.14]. The problem is for a long time this technology was undercover in a

lot of obscurity, and nowadays it is still seen with a lot of scepticism [3.15, 3.16].

To give answers to a plethora of doubts and questions I propose the development of the Lightweight TrustZone-assisted Hypervisor (LTZVisor) as a tool to clearly understand and evaluate how TrustZone hardware can be efficiently exploited to assist virtualization. I describe all the details behind the implementation, highlighting its benefits and discussing identified limitations and how they can be overcome. I conducted an extensive set of experiments which corroborate the viability of the proposed solution, encouraging future research on this direction.

This Chapter is organized as follows: Section 3.1 clearly states the objectives with the development of LTZvisor, and Section 3.2 overview the proposed architecture and outlines its design principles. The implementation of the hypervisor is described in Sections 3.3 and 3.4, and then evaluated in Section 3.5. The benefits and limitations of the proposed solution are discussed in Section 3.6, and, finally, Section 3.7 summarizes the Chapter.

Related Publications

The ideas and results presented in this Chapter have partly been published as:

- **S. Pinto**, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares, "*Towards a Lightweight Embedded Virtualization Architecture Exploiting ARM TrustZone*", *in Proceedings of IEEE Conference on Emerging Technology & Factory Automation (ETFA)*, Barcelona, 2014, pp. 1-4.
- **S. Pinto**, J. Pereira, D. Oliveira, F. Alves, E. Qaralleh, M. Ekpanyapong, J. Cabral, and A. Tavares, "*Porting SLOTH system to FreeRTOS running on ARM Cortex-M3*", *in Proceedings of IEEE International Symposium on Industrial Electronics (ISIE)*, Istanbul, 2014, pp. 1888-1893.

Related Awards

The ideas and results presented in this Chapter have also been awarded as:

- **Work in Progress Best Paper Award in Emerging Technologies** for the paper "*Towards a Lightweight Embedded Virtualization Architecture Exploiting ARM TrustZone*", *in Proceedings of the 2014 IEEE Conference on Emerging Technology & Factory Automation (ETFA)*, Barcelona, 2014.

3.1 LTZVisor: Objectives

The idea of using TrustZone as a virtualization technique is not completely new. The problem is for a long time this technology (developed for security purposes) was maintained under a lot of obscurity, and nowadays it is still seen with a lot of scepticism when regarding virtualization. The overall goal with the development of LTZVisor is to study, evaluate and understand the feasibility of exploiting TrustZone to assist virtualization, highlighting the benefits and stating the limitations. In other words, the main goal is to increase the awareness regarding the applicability of ARM TrustZone technology to assist embedded virtualization:

- ***Objective 1:*** Analyze how CPU virtualization can be achieved. Evaluate if it can be guaranteed by means of TrustZone hardware and quantify how much overhead it introduces;
- ***Objective 2:*** Analyze how memory isolation can be achieved. Evaluate which mechanisms are provided by means of TrustZone hardware and quantify the introduced overhead;
- ***Objective 3:*** Analyze how caches and MMU can be managed. Evaluate if it can be done by means of TrustZone hardware and quantify how much overhead it introduces;
- ***Objective 4:*** Analyze how device partition can be achieved. Evaluate which mechanisms are provided by means of TrustZone hardware and how much overhead it introduces;
- ***Objective 5:*** Analyze how interrupts for different guest OSes can be managed. Evaluate which mechanisms are provided by means of TrustZone hardware and quantify the introduced overhead;
- ***Objective 6:*** Analyze how the real-time behavior of a real time guest OS can be preserved. Evaluate which mechanisms of TrustZone hardware can be exploited and quantify how much overhead it introduces;
- ***Objective 7:*** Analyze and evaluate if and how Operating Systems need to be modified to run as guest OSes.

Each of the above objectives require some analysis and some evaluation work to be carried out. The evaluation work is actually constructive in the sense that the

question is operationalized by "trying to implement it exploiting TrustZone hardware technology". So, for each objective, an analytical part and a constructive part had to be embodied in the actual study design.

3.2 LTZVisor: Design

LTZvisor exploits the dual-virtual environment provided by TrustZone technology to implement an efficient dual-OS virtualization solution. This Section describes the adopted design principles as well as the proposed system architecture.

3.2.1 Design Principles

The main design idea behind LTZVisor is the use of TrustZone hardware to assist virtualization. The key towards TrustZone-assisted virtualization is to rely on hardware support as much as possible, while containing software implementation and components privileges, and promoting the secure environment with a higher privilege of execution. This leads to three fundamental principles:

- ***The principle of minimal implementation:*** Spaghetti code is the main source of vulnerabilities in software and provides an avenue of exploitation for hackers. Relying on the hardware support of TrustZone technology for virtualization as much as possible, as well as promoting the careful design and static configuration of each hypervisor component, will definitively help us minimize the trusted computing base of the system and, consequently, contain the attack surface.

- ***The principle of least privilege:*** Components must be given access only to those resources (e.g., I/O devices, system services, etc) that are absolutely required. TrustZone technology guarantees, by design, that the non-secure world is always less privileged than the secure one, despite the CPU execution mode. Furthermore, in the secure world, the monitor mode introduces a third level of privileges. Exploring these features to implement a well-layered virtualization approach will help promoting privileged execution and hardware-enforced isolation of the real-time environment from the non-real-time one.

- ***The principle of asymmetric scheduling:*** Virtualization of a real-time environment is very challenging, mainly due to strict timing requirements and

hierarchical scheduling problems that those systems introduce. The adoption of an asymmetric scheduling policy, where the secure environment has a higher privilege of execution than the non-secure one, will guarantee that timing requirements are met, even executing real-time tasks over the RTOS running on top of a virtual CPU.

3.2.2 General Architecture

LTZVisor provides a virtualization solution based on the two virtual execution environments provided by the TrustZone hardware. The secure world is responsible for hosting the privileged software, while the non-secure world is responsible for hosting the non-privileged software. Figure 3.1 depicts the proposed virtualization architecture. In this figure, three main software components can be identified: the hypervisor, the secure VM, and the non-secure VM.

LTZVisor runs in the highest privileged processor mode, i.e., in monitor mode. When running in this mode, the processor state is considered always secure. The hypervisor has full control of all hardware and software resources, and is responsible for configuring memory, interrupts and devices assigned to each VM, as well as managing the virtual machine control block (VMCB) of each VM during a partition switch. When a VM is about to be executed by the physical processor, the hypervisor transfers the VM state, saved on the respective VMCB, to the physical processor context. When the hypervisor assigns the physical processor to another virtual machine, the processor context of the active VM is saved back into the VMCB.

The secure VM runs in the supervisor mode of the secure world side. This VM needs to have a small footprint, because when the processor state is secure it has

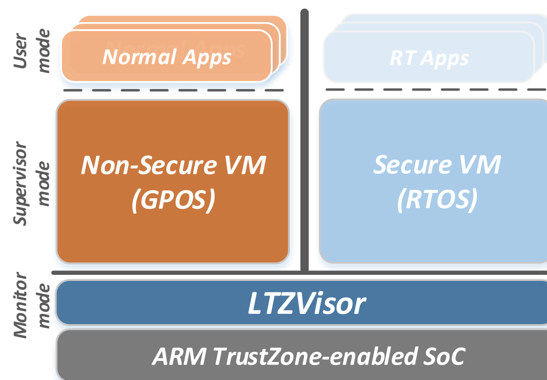


Figure 3.1: LTZVisor: general architecture

full view over the non-secure world side. As such, the privileged guest code can interfere with the other virtual machine, by accessing or modifying its state or the state of its resources (memory or memory mapped devices). For this reason, the OS hosted on the secure VM must be aware of the virtualization, and is considered part of the system's TCB. The secure VM is ideal to run an RTOS, because the higher privilege of execution help meeting the timing requirements of such environments. Furthermore, RTOSes typically have small memory footprint.

The non-secure VM runs in the supervisor mode of the non-secure world side. This VM is ideal to host a general purpose guest OS, useful for running human-machine interfaces as well as internet-based applications and services. The software running on the non-secure world side is completely isolated from the privileged software running on the secure world side. When the processor is operating in a privileged mode but not in the secure state, it cannot access nor modify any state information belonging to the secure world. Any attempt from the non-secure guest OS to access any resource of the secure world side immediately triggers an exception to the hypervisor. The only limitation posed on the operating system hosted on the non-secure side is that it can no longer use the TrustZone features by itself. The virtual architecture is not completely identical to the physical one, but it is identical to the bare architecture without TrustZone enhancements.

3.3 LTZVisor: Implementation

LTZVisor exploits ARM TrustZone to provide time and space isolation between both partitions. The asymmetric design principle allows to preserve the real-time characteristics of the secure virtual machine (RTOS) without any non-real-time interference. This Section provides all the details behind LTZVisor implementation, describing how CPU virtualization and memory isolation is ensured, presenting how MMU and caches are managed, describing how device partition is achieved, explaining how interrupts and time are managed for different guest OSes, and illustrating how inter-VM communication is implemented.

3.3.1 Virtual CPU

TrustZone technology virtualizes each physical CPU into two virtual CPUs: one for the secure world and another for the non-secure world. Between both worlds there

is a list of banked registers, i.e., an individual copy of those registers exists for each world. Since each guest OS is running in a different world, in this particular case, a huge part of the virtual CPU support is guaranteed by the hardware itself, minimizing the number of registers to be saved and restored in each partition-switching operation. The VMCB of the non-secure side is composed by 25 registers: 13 *General Purpose Registers* (*R0-R12*), the Stack Pointer (*SP*), the Linker Register (*LR*) and *Saved Program Status Register* (*SPSR*) for the Supervisor, System, Abort and Undef modes. The "high" *General Purpose Registers* (*R8-R12*), as well as the *SP*, *LR* and *SPSR* of the FIQ and IRQ modes are not included, as they are mutually exclusive for each world. Among the coprocessor registers, almost all of them are banked: only the *SCTLR* and the *ACTLR* need to be preserved. For optimization purposes, the VMCB of the secure side is composed of only 16 registers: 13 *General Purpose Registers* (*R0-R12*), the *SP*, the *LR* and *SPSR* for the System mode. The Monitor mode is, by design, uniquely dedicated to the secure world side. These optimizations reduce the interrupt latency from the secure guest OS (RTOS) perspective, speeding up the transition from the non-secure to the secure world side, when a secure interrupt arises while the non-secure OS is executing.

Among the aforementioned unbanked registers, there are those which are only modifiable from the secure side: they can be read when the processor is in the non-secure state, but an attempt to modify them will be ignored. This is stated on TrustZone specification to guarantee a high degree of security in the system, incurring a cost for the non-secure guest OS. For example, the *System Control Register* (*SCTLR*) and the *Auxiliary Control Register* (*ACTLR*) provide control and configuration over memory, cache, MMU, AXI accesses, etc. These registers are used to enable and disable MMU, and are only accessible in the secure state. During the non-secure guest OS boot process, an attempt to modify them will be ignored, leading the GPOS to get stuck. For that reason, the hypervisor must fill some registers of the non-secure VMCB with a specific initialization value. For example, the *SCTLR* register of the non-secure VMCB should be initialized appropriately (0x00c50078), so that MMU and Level1 cache of the non-secure world are enabled before the GPOS starts booting.

3.3.2 Scheduler

An identified issue in virtualizing a real-time environment is the well-known hierarchical scheduling problem. Typically, a hypervisor schedules virtual CPUs while

a guest RTOS running over the virtual CPU schedules its own tasks. Ensuring real-time execution of tasks over the RTOS executing on top of a virtual CPU involves a complex hierarchical scheduling analysis, requiring that both schedulers are accordingly modeled [3.10].

LTZVisor overcomes this problem by implementing an asymmetric or idle scheduler. This scheduling policy guarantees that the non-secure guest OS is only scheduled during the idle periods of the secure guest OS, and the secure guest OS can preempt the execution of the non-secure one. In fact, the secure virtual machine (RTOS) has a higher scheduling priority than the non-secure one, and LTZVisor is not the software component that directly schedules the virtual machines, but it is scheduled itself by the secure guest OS. Although this can seem contradictory, the concept of ring protection is never jeopardized, as the LTZvisor continues executing in a more privileged mode than the secure guest OS: the hypervisor is just configured to behave in a passive way.

3.3.3 Memory Partition

Traditional hardware-assisted memory virtualization relies on MMU support for 2-level address translation, mapping guest virtual to guest physical addresses and then guest physical to host physical addresses. This MMU feature is a key enabler to run unmodified partition OSes, and also to implement isolation between partitions.

TrustZone-enabled SoCs (which are not VE-enabled) only provide MMU support for single-level address translation. Therefore, the existence of a TZASC is a major requirement for the proposed solution, because this component allows partition of memory into different segments. This memory segmentation feature can be exploited to guarantee spatial isolation between the non-secure VM and the secure one, basically by adequately configuring the security state of the memory segments of respective partitions. The non-secure VM should have its own memory segment(s) configured as non-secure, and the remaining memory as secure. If the non-secure guest OS tries to access a secure memory region (either belonging to the secure partition or the hypervisor), an exception is automatically triggered and the execution control redirected to the hypervisor.

Memory segments can be configured with a specific granularity, which is implementation defined, depending on the vendor. In the hardware under which the system was deployed, Xilinx ZC702, memory regions can be configured with a granularity

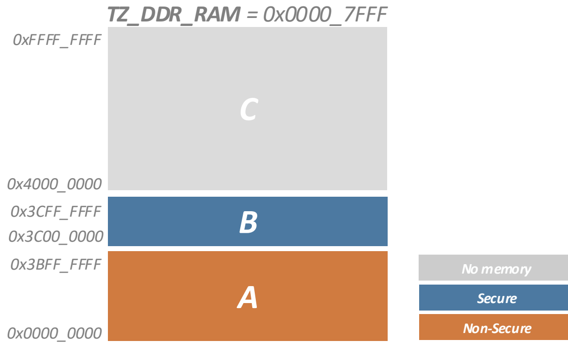


Figure 3.2: LTZVisor: memory configuration

of 64MB. This configuration is provided via a system level control register named `TZ_DDR_RAM`. A 0 or 1 on a particular bit indicates a secure or non-secure memory region for that particular memory segment, respectively. Figure 3.2 depicts the memory setup and respective secure/non-secure mappings, for a virtualized system consisting of the hypervisor altogether with the secure virtual machine (B), and the non-secure virtual machine (A). In this specific configuration, the non-secure VM (GPOS) uses the first fifteen memory segments (0x00000000 - 0x3BFFFFFF), corresponding to a total of 960MB of non-secure memory. The hypervisor and the secure VM, due to their low memory footprint, use only the last available memory segment (0x3C000000 - 0x3FFFFFFF), corresponding to a 64MB of secure memory. The remainder of the 32-bit memory address space is not accessible (C), because Xilinx ZC702 only comes with a 1GB DDR3 memory.

3.3.4 MMU and Cache Management

The TrustZone-aware MMU provides two distinct MMU interfaces, enabling each world to have a local set of virtual-to-physical memory address translation tables. This means each world has its own copy of the TTBR register set, as well as an independent MMU configuration. This reduces the list of activities to perform on each guest-switching operation, because translation lookaside buffer entries do not need to be invalidated.

The same kind of isolation is still available at cache-level. The processor caches have been extended with an additional tag bit (NS bit) which records the security state of the transaction that accesses the memory. This NS bit is set by hardware and it is not directly accessible by system software. Therefore, in terms of the cache coherence design, when the system switches between the two worlds, none of the cache lines need to be flushed. This means that this design feature at cache-level

significantly improves the performance of LTZVisor, because no cache management operation needs to be performed on each guest-switching operation: cache isolation is enforced and guaranteed by the hardware itself. On Xilinx ZC702, there are a few notes regarding the TrustZone support in L2 cache (PL310). The L2 Control register (`reg1_control`) can only be written with an access tagged as secure, which means that an attempt to enable or disable the L2 cache from the non-secure world side will be ignored. Similarly to the support that the hypervisor needs to perform in the L1 cache initialization (aforementioned in Section 3.3.1), LTZVisor also needs to enable the L2 cache on the secure world side before the non-secure guest OS starts booting. Once the L2 cache is enabled, maintenance operations on the non-secure entries can be performed directly from the non-secure world side.

3.3.5 Device Partition

TrustZone technology allows devices to be (statically or dynamically) configured as secure or non-secure. This hardware feature allows the partition of devices between both worlds while enforcing isolation at the device level.

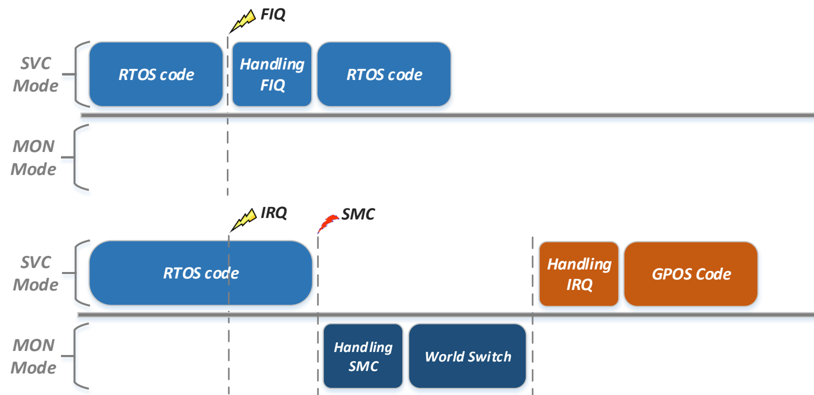
LTZVisor implements device virtualization adopting a pass-through policy, which means devices are managed directly by guest partitions. To ensure strong isolation between them, devices are not shared between guests and are assigned to the respective partitions at design time, and then configured during boot time. The devices assigned to the RTOS are configured as secure devices, while devices assigned to the GPOS are configured as non-secure devices. This guarantees the GPOS cannot compromise the state of any device belonging to the RTOS, and if the non-secure guest partition tries to access a secure device then an exception will be automatically triggered and handled by hypervisor. On Xilinx ZC702, the security state of devices can be configured through a set of secure registers accessible from the secure side. This set includes, for example, the Secure Digital Input Output (SDIO) slave security registers (`security2_sdio0` and `security3_sdio1`) and the APB slave security register (`security6_apb_slaves`).

3.3.6 Interrupt Management

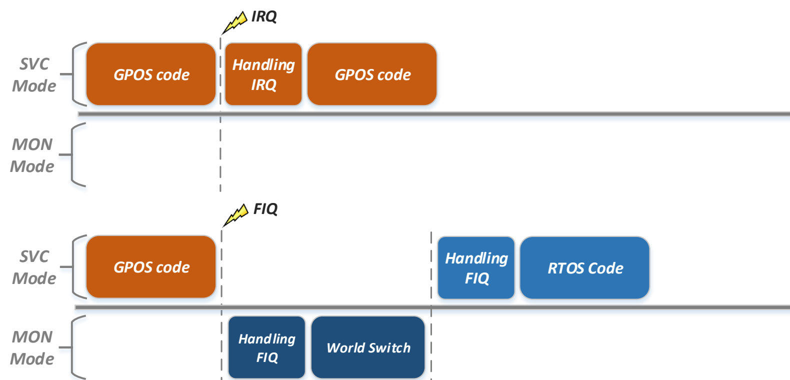
In TrustZone-enabled SoCs, the GIC supports the coexistence of secure and non-secure interrupt sources. It also allows the configuration of secure interrupts with a

higher priority than the non-secure ones, and has several configuration models that enable the assignment of IRQs and FIQs to secure or non-secure interrupt sources.

LTZVisor configures interrupts of secure devices (i.e., secure interrupts) as FIQs, and interrupts of non-secure devices (i.e., non-secure interrupts) as IRQs. A TrustZone-enabled GIC permits all implemented interrupts to be individually defined as secure or non-secure, through the *Interrupt Security Registers* set (*ICDISRn*). To program secure interrupts to use the FIQ interrupt mechanism of the processor, the *FIQen* bit in the *CPU Interface Control Register* (*ICPICR*) should be set. When the secure guest OS (i.e., RTOS) is under execution, secure interrupts (i.e., FIQs) are redirected to the RTOS without hypervisor interference, guaranteeing that no overhead is added to the interrupt latency of the secure guest OS. This can be done by disabling the *FIQ* bit into the *Secure Configuration Register* (*SCR*). If an IRQ (i.e., an interrupt for the GPOS partition) arises while the RTOS is executing, it doesn't affect the expected RTOS behavior. As soon as the non-secure guest becomes active, the interrupt will be then processed.



(a) Secure guest OS (RTOS) perspective



(b) Non-secure guest OS (GPOS) perspective

Figure 3.3: LTZVisor: interrupt management

The prioritization of secure interrupts prevents a denial-of-service attack against the secure side (from the GPOS partition). From a different perspective, when the non-secure guest OS (i.e., GPOS) is executing and an FIQ (i.e., an interrupt for the RTOS partition) arises, the execution flow is immediately redirected to the hypervisor, which will be responsible for handling the interrupt directly in monitor mode. This design decision minimizes the interrupt latency from the RTOS perspective, ensuring the interrupt is attended as soon as possible. On the other hand, if an IRQ arises, it will be directly managed by the non-secure guest. Non-secure interrupts are always signaled (by design) using the IRQ mechanism of the processor. Fig. 3.3 summarizes the interrupt management process from every guest OS perspective.

3.3.7 Time Management

Temporal isolation in virtualized systems is typically achieved using two levels of timing: at hypervisor level and at partition level. For the partition level, hypervisors typically provide timing services which allow guests to have notion of virtual or real time. In the first case, each time a partition is inactive, the time is paused, and once the guest is rescheduled, the timekeeping is resumed. In the second case, when the partition is paused, the hypervisor is responsible for keeping track of the wall-clock time, and, once resumed, update the partition timing structures.

LTZVisor provides a distinctive time management implementation. Due to its dual-OS configuration, as well as the intrinsic design principle of asymmetric scheduling, the hypervisor dedicates one independent timing unit for each guest OS. The secure VM uses the Triple Timer Counter (TTC) 0, while the non-secure VM uses the TTC1. It is fundamental that the hypervisor configures TTC1 as a non-secure device, otherwise an exception will be triggered on the first attempt to access it. This specific time management implementation ensures that each VM has its timing structures updated at all times. The RTOS does not miss any system-tick interrupt, and the GPOS, as a tickless OS, is completely aware of the real passage of time.

3.4 LTZVisor: Execution Flow

The system starts on the secure side with the system boot process. This procedure is responsible for a set of operations which includes specific processor and coprocessor registers initialization, as well as stacks, memory, peripherals and interrupt controller

configuration; e.g., an amount of memory is configured as secure and another as non-secure. The GIC is also configured to route FIQs to the secure world, and IRQs to the non-secure world. On the *SCR* register the *FIQ* and *IRQ* bits are disabled to guarantee that FIQ/IRQ exceptions do not cause a switch to monitor mode, and consequently the secure to non-secure world switch is only performed through the SMC instruction.

After the system boot process, the RTOS is booted and starts scheduling its own tasks. When all the real-time tasks are blocked and/or suspended, the idle task performs a system call that is responsible for explicitly invoking the hypervisor, through a *SMC* instruction. Immediately, the processor changes to the monitor mode and starts executing the VMM, jumping to the specific handler of the monitor vector table. Hence, the processor execution is routed to the SMC handler which prepares the transition to the non-secure world.

The next step performs the context-switch operation. Concretely, the processor state of the secure side (FreeRTOS) is saved in its own VMCB, and the VMCB of the non-secure side (Linux) is restored. An exception occurs at the first execution, when due to optimization purpose, only the processor state of the secure side is saved, the supervisor mode is set, and the linker register is updated with the start address of the non-secure OS kernel. At the end, LTZVisor enables the FIQ and NS bits of SCR register and jumps to the initialized/restored non-secure address.

As it can be noticed, until this moment no operation on cache was performed. As previously explained, TrustZone permits that cache entries of the secure and the non-secure world co-exist together. This support removes the need for a cache flush when switching between worlds, and contributes to a reduced context-switch time.

Once on the non-secure side, the GPOS will run until the moment that a FIQ is triggered. Since the FIQ bit had been previously enabled, the arrival of a FIQ request brings the processor into monitor mode, jumping to the FIQ handler of the monitor vector table. At this moment, the VMM begins executing and prepares the context-switch operation. It starts by disabling the FIQ and NS bits of SCR register, saves the full processor state view of the non-secure side into its VMCB, acknowledges the FIQ request and restores the secure side context from the VMCB.

At this point, the processor returns to the RTOS kernel, which will start dispatching real-time tasks. The processor will remain in the secure world side until the moment that the idle task is re-scheduled. When it happens, the processor performs all previously described steps again. Fig.3.4 summarizes the execution flow process.

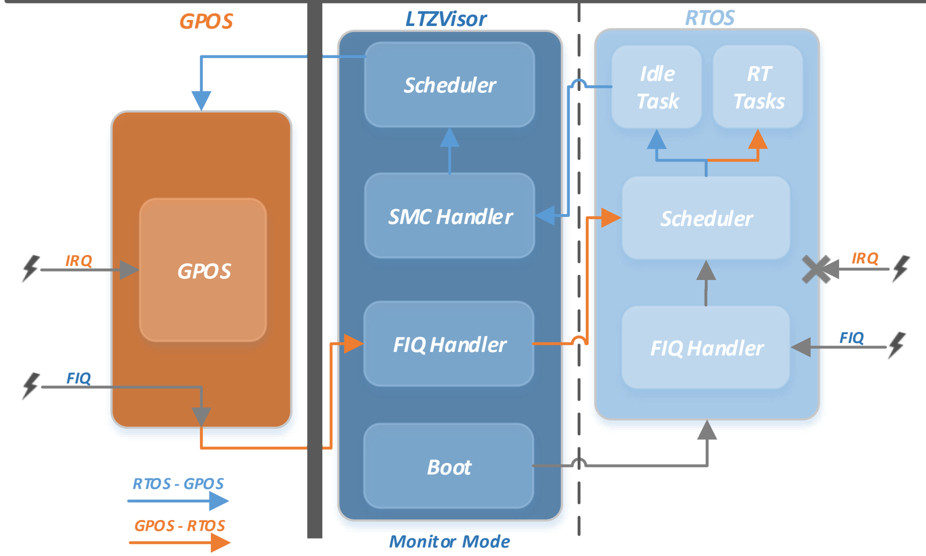


Figure 3.4: LTZVisor: execution flow

3.5 Evaluation

LTZVisor was evaluated on a Xilinx ZC702 evaluation board targeting a dual ARM Cortex-A9 running at 667MHz. In spite of using a multicore hardware architecture, the evaluated implementation only supports a single-core configuration. The evaluation focused on three metrics: memory footprint, performance overhead, and interrupt latency. LTZVisor and both OS partitions were compiled using the ARM GNU toolchain, with compilation optimizations disabled (-O0). Linaro Linux (v3.3.0) and FreeRTOS (v7.0.2) were used as non-secure and secure partitions, respectively. MMU, data and instruction cache and branch predictor were disabled on the secure world side.

3.5.1 Memory Footprint

In order to assess the memory footprint of each software component of the implemented architecture I used the size tool of the ARM GNU toolchain. I evaluated LTZVisor, as well as the native, modified and virtualized version of FreeRTOS. Table 3.1 presents the collected measurements, where boot code, libraries and drivers were not taken into consideration. As it can be seen, the memory overhead introduced by the hypervisor is really small, i.e., 2880 bytes. The main reasons behind such a low memory footprint are related to the principle of minimal implementation followed during LTZVisor design which relies on (1) the hardware support of TrustZone tech-

Table 3.1: LTZVisor: memory footprint (bytes)

Software	***Memory Footprint***			
	.text	*.data*	.bss	**Total**
LTZVisor	2368	0	512	**2880**
FreeRTOS IRQ (v7.0.2)	17942	20	920	**18882**
FreeRTOS FIQ (v7.0.2)	17954	20	924	**18898**
vFreeRTOS FIQ (v7.0.2)	17974	20	924	**18918**

nology for virtualization and (2) the careful design and static configuration of each hypervisor component. The native version of FreeRTOS, supporting IRQ, requires 18882 bytes, the modified version, supporting FIQ, requires 18898, and the virtualized version requires 18918 bytes. From the native version to the modified one there is a slight increase of 0.08% in the memory footprint, while from the native version to the virtualized one there is an increase of 0.19%. This slight increase is completely acceptable and encompasses small modifications and adaptations for FIQ and context-switch handling (from native to modified), and in the FreeRTOS scheduler (from modified to virtualized).

3.5.2 Performance

The performance evaluation process was split into three different test case scenarios. Firstly, LTZVisor was evaluated for specific micro-operations of the guest-switching operation. Then, I evaluated the virtualization overhead (using the Thread-Metric), as well as the interrupt latency over the secure VM (RTOS). Finally, I assessed the virtualization overhead over the non-secure VM (GPOS) using the LMBench3 Suite.

Partition context switching

To evaluate the guest context-switch time I used the Performance Monitor Unit (PMU) component. To measure the time consumed by each internal activity of a round-trip world switch, a PMU-specific instruction was added at the beginning and end of each code portion to be measured. Results were gathered in clock cycles and converted to microseconds accordingly to the processor's frequency (667MHz). Each value represents the average and the standard deviation of ten collected samples.

The list of internal activities to perform a full switch between secure to non-secure and non-secure to secure worlds are:

1. ***SMC handling*** - The secure guest OS schedules the idle task. The idle task

performs a secure call that is responsible for invoking the hypervisor (*SMC*). It is the time since the processor enters into the monitor's vector table until LTZVisor completes the SMC handling;

2. ***Save secure guest OS context*** - LTZvisor handles the SMC request and saves the context of the secure guest OS. It is the time to save the current state of the secure guest OS to its respective VMCB;

3. ***Restore non-secure guest OS context*** - LTZvisor saves the context of the secure guest OS and then restores the context of the non-secure guest OS. It is the time to restore the state of the non-secure guest OS from its respective VMCB;

4. ***FIQ acknowledge*** - The non-secure guest OS is running while a secure interrupt is triggered (e.g., RTOS timer tick). It is the time since the processor enters in the monitor's vector table until LTZVisor acknowledges the FIQ;

5. ***Save non-secure guest OS context*** - LTZvisor acknowledges the FIQ request and then saves the context of the non-secure guest OS. It is the time to save the current state of the non-secure guest OS to its respective VMCB;

6. ***FIQ handling*** - LTZvisor saves the context of the non-secure guest OS and then immediately handles the FIQ request. It is the time since the hypervisor save the current state of the non-secure guest OS until LTZVisor completes the FIQ handling;

7. ***Restore secure guest OS context*** - LTZvisor handles the FIQ and then restores the context of the secure guest OS. It is the time to restore the state of the secure guest OS from its respective VMCB;

8. ***Scheduler*** - LTZvisor restores the execution of the RTOS. The RTOS continues executing the idle task loop and verifies if there are real-time tasks to run. If not, the idle task performs a system call (*SMC*) that is responsible for invoking the hypervisor. It is the time since the processor restores the idle task execution until it enters in the monitor's vector table.

Table 3.2 presents the collected results. As it can be seen, the complete partition-switch operation takes around 19.21 microseconds. This value assumes there are no real-time tasks ready to run once the RTOS is rescheduled. The process of checking, by the RTOS, for a real-time task to run and, accordingly, trigger the switch to the non-secure world takes around 11.32 microseconds. The process of switching from

Table 3.2: LTZVisor: performance statistics

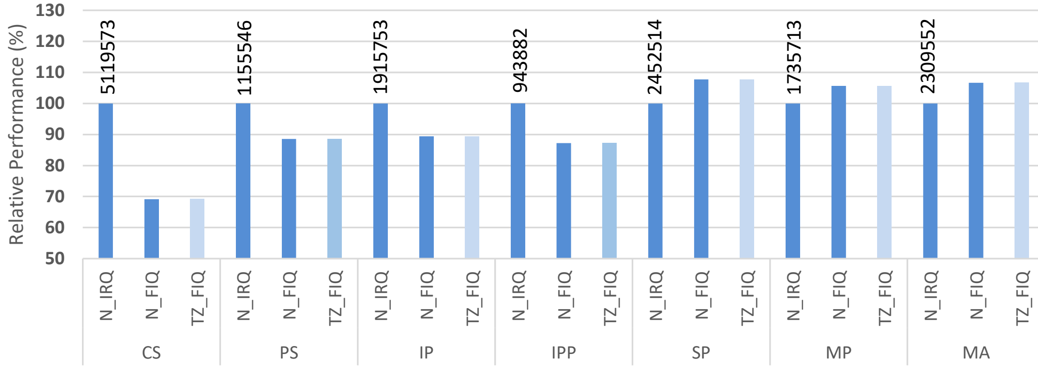
World Switch	***Operation***	***Latency***		***Time***
		$\overline{x}$	s	@667MHz
	(1) SMC handling	570	0.843	855ns
Switch to NS world	*(2) Save S guest OS context*	421	1.174	631ns
	(3) Restore NS guest OS context	950	1.989	1424ns
	(4) FIQ acknowledge	466	0.516	699ns
Switch to S world	*(5) Save NS guest OS context*	983	1.567	1474ns
	(6) FIQ Handling	1633	48.252	2448ns
	(7) Restore S guest OS context	243	0.483	364ns
Scheduler	*(8) Assymetric Policy*	7548	10.266	11316ns
	Total	**12814**		**19211**ns

the RTOS to the GPOS takes just 2.91 microseconds, and is the most deterministic activity of the partition-switching operation. Experiments demonstrated just a few clock cycles of deviation from the average value. Once the GPOS is executing and a FIQ is triggered, the hypervisor ensures a 2.17 microseconds of interrupt latency, and then in a further 2.81 microseconds the RTOS is restored. The FIQ handling operation is the major source of non-determinism of the partition-switching operation. The reason is related to the nonlinearities in accessing the peripheral bus, when handling the interrupt request (in this specific case, the system tick timer).

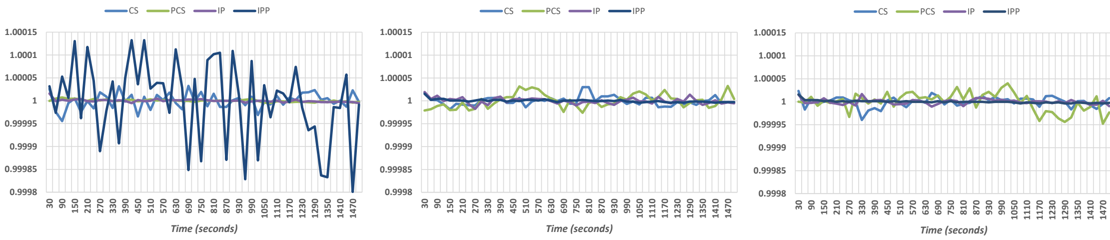
Secure VM (RTOS)

As said before, the Thread-Metric Benchmark Suite consists of a set of benchmarks properly conceived to evaluate RTOSes performance. The suite comprises 7 benchmarks, evaluating the most common RTOS services and interrupt processing: cooperative scheduling (CS); preemptive scheduling (PS); interrupt processing (IP); interrupt preemption processing (IPP); synchronization processing (SP); message processing (MP); and memory allocation (MA). Each benchmark outputs a counter value, representing the RTOS impact on the running application: the higher the value, the smaller the impact.

Benchmarks were executed in the native version of FreeRTOS (N_IRQ), where interrupts are handled as IRQs, in a modified version of FreeRTOS, where interrupts are handled as FIQs (N_FIQ), and then compared against the virtualized version (TZ_FIQ). Figure 3.5 presents the achieved results, corresponding to the average relative performance (as well as the average absolute performance) of 50 collected samples for each benchmark. Each sample reflects the benchmark score for a 30 seconds execution time, encompassing a total execution time of 25 minutes for each



(a) Relative performance



(b) Variation: Native (IRQ) (c) Variation: Native (FIQ) (d) Variation:Virtualized (FIQ)

Figure 3.5: LTZVisor: Thread-Metric benchmarks

benchmark.

In accordance with Figure 3.5a the execution of the modified version of FreeRTOS (N_FIQ) is very dependent from the benchmark. In some cases, the performance decreases, while in others the performance increases. The increase of performance on the modified version of FreeRTOS is completely understandable since FIQ interrupts present low hardware latency than IRQs, but the decrease is apparently strange. The reason behind this phenomenon is related to an adaption made to the yield macro of FreeRTOS. The native version of FreeRTOS implements the yield through the use of the SVC exception. When an SVC is triggered a context-switch happens and the IRQ bit of the CPSR is set, so that there is no preemption during the execution of the critical routine (atomic execution). Thus, the modification of FreeRTOS for handling interrupts as FIQs should include the modification of the context-switch function to set the FIQ bit, instead of the IRQ bit. The problem is that according to the ARMv7-A specification, this bit is implementation defined. In the case of Xilinx Zynq, for security reasons, this bit is read-only, and only changes when triggered by hardware (e.g. when a FIQ happens). For this reason, I was forced to change the yield function to use a software generated interrupt (SGI) as FIQ, instead of the SVC exception. The SGI has a higher latency than the SVC, which, on yield-intensive tests (i.e., the case of CS, PS, IP and IPP), this translates in a decrease

of performance. It should be noted this is platform- and workload-specific problem that does not necessarily mean it can occur in other platforms and be noticeable in real application scenarios. In fact, the overhead introduced by LTZVisor is null, as demonstrated by the comparison of the N_FIQ and TZ_FIQ versions. This is perfectly understandable because, once FreeRTOS starts running real-time tasks, it will never be interrupted by the hypervisor.

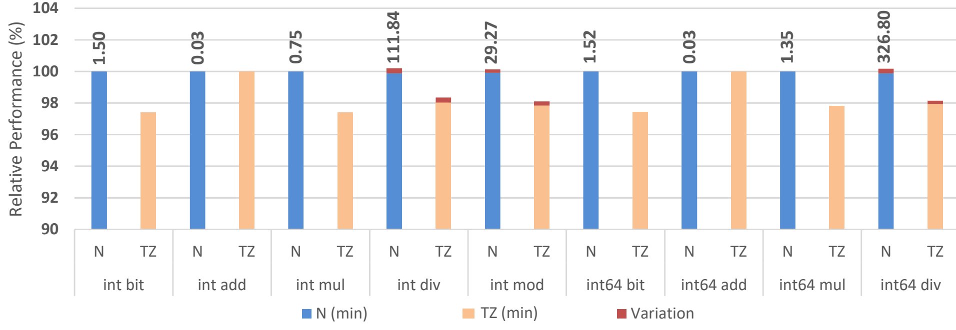
Regarding the variation, Figure 3.5b, Figure 3.5c and Figure 3.5d present the normalized variation of the collected results over time for the native, modified and virtualized versions of FreeRTOS, respectively. It is clear that the use of FIQ for handling interrupt sources slightly reduces the variation of results, and variation in the virtualized system is also in the same order of magnitude as the modified version, which means the virtualized system remains as deterministic as the (modified) native one. In sum, the asymmetric scheduling policy gives the RTOS a higher execution privilege, so it can preserve its real-time characteristics. Furthermore, the need for handling interrupts as FIQs promotes a deterministic execution, and most of the cases can either increase performance.

Non-Secure VM (GPOS)

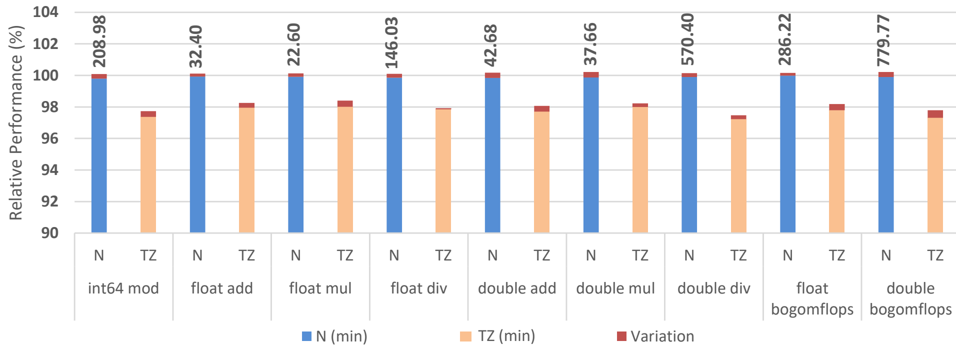
LMBench [3.17] is a widely used suite of micro-benchmarks that measure a variety of important aspects of system performance, such as latency and bandwidth. The suite is written in portable ANSI-C using Portable Operating System Interface (POSIX) interfaces and targeting UNIX systems. The LMBench 3.0 suite includes more than forty micro-benchmarks within three different categories: *bandwidth*, *latency*, and *other*. The evaluation was focused on three specific benchmarks:

- ***lat_ops:*** Arithmetic operations latency, to evaluate general CPU performance (VFP and NEON are disabled);
- ***bw_mem:*** Memory operations bandwidth for different blocks size (2K, 128K, 4M), to evaluate the interference of the TZASC as well as Level 1 (4-way set-associative 32 KB) and Level 2 (8-way set-associative 512 KB) data caches;
- ***lat_syscall:*** System calls latency, to evaluate if device virtualization using pass-through policy does not introduce an extra overhead.

For this experiment, FreeRTOS was configured with a 1ms tick rate (i.e., guest-switching rate) and no real time tasks were added to the system (i.e., the RTOS



(a) *lat_ops* benchmark results (part 1)

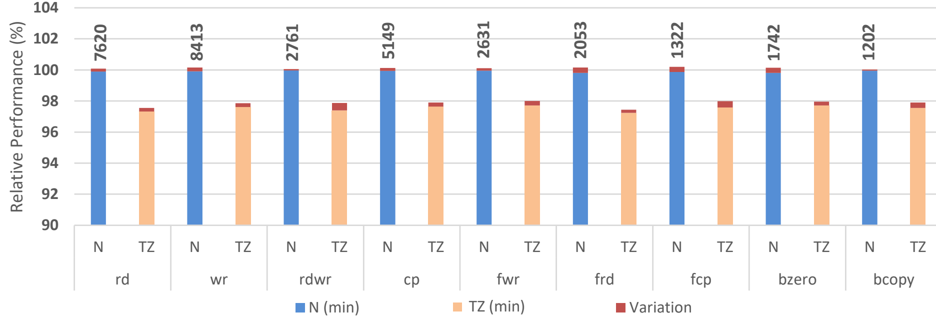


(b) *lat_ops* benchmark results (part 2)

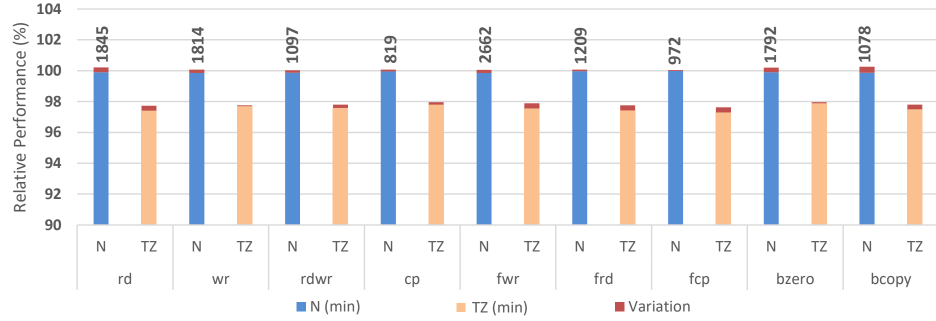
Figure 3.6: LTZVisor: arithmetic operations latency benchmark

will be infinitely executing the idle task). Micro-benchmarks were ran in the native version of Linux (N) and then compared against the virtualized version (TZ). MMU, L1 and L2 caches and branch predictor were enabled for both test case scenarios. For each micro-benchmark I performed 10 consecutive experiments. For each experiment the micro-benchmark was configured for 10 warm-ups and 100 repetitions (-W 10 -N 100). Presented results correspond to the average relative performance and variation (as well as the average absolute performance) of the 10 consecutive experiments, encompassing a total of 1000 samples.

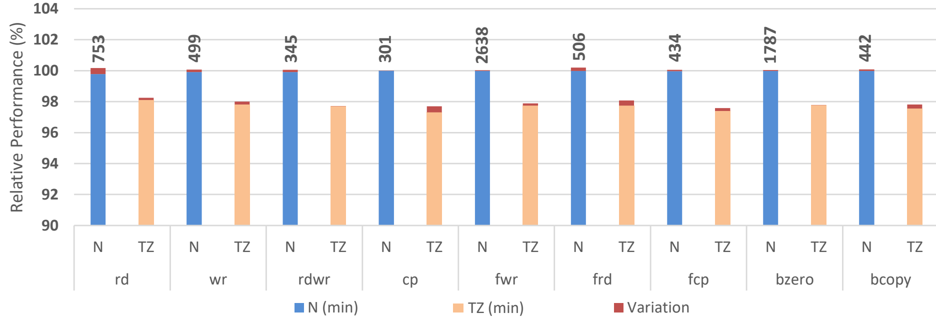
Figure 3.6 presents the achieved results for the arithmetic operations latency benchmark. The values on top of the bars correspond to the average latency, in nanoseconds. As it can be seen, the virtualized version of Linux only presents an average performance degradation of 2%, when compared to its native execution. This value is practically uniform among all micro-benchmarks (apart from the small variations due to the benchmark's lack of accuracy and the system's nonlinearities), except for the `int add` and `int64 add` cases. For these specific micro-benchmarks, the achieved results do not reflect the real performance penalty, due to the lack of precision. The assessed latency is 0.03 nanoseconds, and the minimal time unit is 0.01 nanoseconds. Regarding variation, it is clear the virtualized Linux presents a varia-



(a) *bw_mem* benchmark results (2KB)



(b) *bw_mem* benchmark results (128KB)



(c) *bw_mem* benchmark results (4MB)

Figure 3.7: LTZVisor: memory bandwidth benchmark

tion in the same order of magnitude as the native version. This means the virtualized system remains as deterministic as the native one.

Figure 3.7 presents the achieved results for the memory bandwidth benchmark. The values on top of the bars correspond to the average memory bandwidth, in megabytes per second (MB/s). Figure 3.7a, Figure 3.7b and Figure 3.7c depict the assessed results for a memory block size of 2KB, 128KB and 4MB, respectively. These memory block sizes were selected with the intention to fit and not fit within the L1 and L2 cache sizes, respectively. Looking at the three figures, it is clear the relative performance of the system is practically uniform among all micro-benchmarks, presenting an average performance degradation of 2% when comparing to the virtualized version

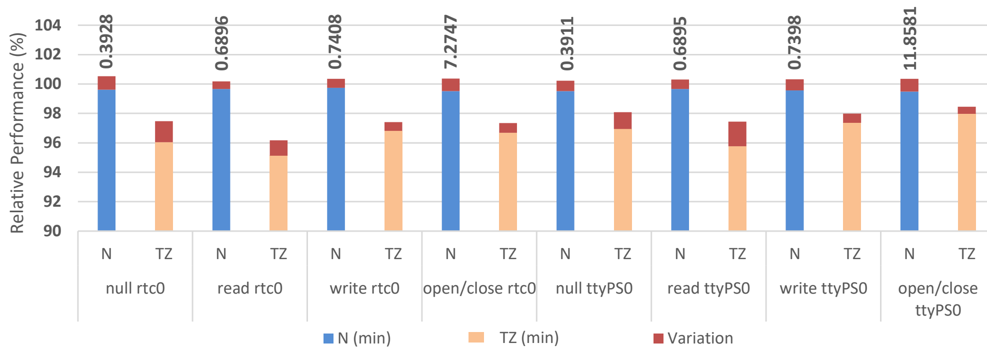


Figure 3.8: LTZVisor: system calls latency benchmark

of Linux with the native one. Contrasting these values with the results presented in Figure 3.6, three main conclusions can be drawn: first, it is clearly noticed the effect of each cache on the accessed absolute memory bandwidth results - the higher the memory block size, the lower the memory bandwidth; second, cache isolation is in fact guaranteed by hardware, and does not introduce any extra overhead neither requires any cache maintenance operation on each guest-switch; and, finally, (memory) space isolation provided by means of the TZASC has not associated any extra source of overhead. To corroborate the viability of my conclusions, experiments were performed without some of the hypervisor support (please refer to Sections 3.3.1 and 3.3.4) for caches and memory initialization. For example, one set of experiments were performed without the hypervisor having enabled L2 cache before booting the GPOS. The results were very straightforward: an abrupt decrease of performance, reaching almost 70% in some cases, happen for memory block sizes higher than 32KB and lower than 512KB. Such experiments clearly demonstrates the effect of L2 cache in the overall system, as well as the coexistence of non-secure and secure cache entries without any cache maintenance support. Despite not being presented in Figure 3.7, due to shortage of space, I also performed a larger set of experiments encompassing memory block sizes of 16KB, 64KB and 1MB. The achieved average relative performance and variation results were identical to the ones presented in Figure 3.7, which reinforces the reliability of my conclusions.

Figure 3.8 presents the achieved results for the system calls latency benchmark, performed for two different devices: *rtc0* (timer) and *ttyPS0* (serial port). The values on top of the bars correspond to the average latency, in microseconds. The assessed results demonstrate the virtualized version of Linux has an average performance degradation of 3% when compared with its native execution. In contrast with Figure 3.6 and Figure 3.7 where results are practically uniform among all micro-benchmarks (differences on the relative performance between the micro-benchmarks

are around 0.5%) and variation is less than 0.5%, the difference between the relative performance of the micro-benchmarks can reach 3% with a maximum variation of 2%. I believe this small deviation from the average relative performance achieved in the other experiments, as well as the lack of uniformity on this particular case are not being influenced by the adopted device virtualization approach (pass-through), but are being caused by the peripheral bus access concurrency. Nevertheless, without an in-depth study with relevant experiments that lead to valid conclusions, by now, it remains as an open question.

3.6 Discussion

With LTZVisor I demonstrate how hardware enhancements introduced by TrustZone technology can be adequately exploited to assist virtualization, especially in the case of two virtual machines, because this number coincides exactly with the number of isolated states directly supported by the processor. I demonstrate and explained how several TrustZone features can be adequately exploited to run an RTOS side-by-side with a GPOS.

The number of virtual machines is limited to two, one running in the secure world and another in the non-secure world. Although this is almost sufficient for several classes of current embedded applications, I still envision to overcome this limitation by multiplexing more guest OSes inside the non-secure world side. It requires carefully handling of shared hardware resources, such as processor registers, memory, caches and MMU. Processor registers can be easily saved and restored into/from a specific VMCB, while memory isolation can be achieved through the dynamic memory configuration feature of TZASC.

Spatial isolation is a major requirement for virtualization. LTZvisor implements memory isolation relying on the TZASC, which is an optional and implementation-specific component on TrustZone specification. The granularity of access restrictions depends on the SoC. Some outdated TrustZone-based SoCs are not equipped with this memory controller, and on many other the TZASC can only control some portions of the memory. For example, the Versatile Express platform provides no means to partition the DDR RAM into secure and non-secure areas. Nevertheless, when regarding the most modern TrustZone-based SoC, this is completely different, because they are totally equipped with fully featured TrustZone-aware memory controllers. This is the case of Xilinx Zynq SoCs and also the Freescale i.MX53 QSB.

Another identified limitation on the memory subsystem is related to nonexistence of a second level memory translation. There is no way to virtualize the physical memory as used by the guest OSes. The guest-physical memory always corresponds to the host-physical memory, which means all guest OSes have to co-operate with respect to the address space being used, requiring relocation and consequent re-compilation of the guest OS. This means the chance to use multiple closed-source guest OSes (only available as binary image) is very reduced, because different OS providers typically compile their software to run on the same memory address space of a specific platform. What is seen as a limitation to the system from a non-real-time perspective, is somewhat seen as an advantage from a real-time perspective. It is well-established the use of MMU and other components which introduce some nonlinearities are seen with some scepticism regarding determinism and worst-case performance requirements of many real-time systems. An important argument that supports my vision is the recent decision of ARM in introducing support for virtualization in the new ARMv8-R architecture relying on a double-stage MPU [3.18]. In the ARMv8-R architecture, operating systems running at PL1 (IRQ, FIQ, SVC, System, etc) are able to use an MPU, as well as the hypervisor running at PL2 (Hypervisor). The MPU controlled by the hypervisor restricts access of memory regions or peripherals to an individual guest, or shared between guests. This is a similar strategy to the one I use with TrustZone, and was adopted by ARM to meet the strict requirements of real-time environments.

The existence of two distinct MMU interfaces as well as secure and non-secure cache entries is also seen as an advantage due to the performance gains achieved during the partitions-switch. From a real-time perspective, the use of these features is not always desirable, which means that in many potential embedded applications the use of MMU and cache will only be exploited by the non-secure guest OS. However, if the idea is to consolidate a soft-real-time system with a general purpose, the use of these features can be helpful in terms of context-switch time and performance. The only disadvantage that arises with the TrustZone-awareness in this components, is the need of minimal hypervisor support on their initialization, as well as their inaccessibility during runtime. In this case, one possible strategy to deal with this limitation is to implement some paravirtualization support, by statically analyzing the non-secure guest OS image file, identify the opcode of the instruction, and replace it by hypercalls that request the access to those components mediated by the hypervisor.

Current device virtualization approach goes towards a pass-through model with-

out any sharing device access support. Device isolation relies on a virtual form of IOMMU provided by means of the TZPC. Similar to the limitation identified in the TZASC, the TZPC is also an optional and implementation-specific component on TrustZone specification. This means the number and type of devices that can be configured as non-secure vary from platform to platform and from vendor to vendor. For example, in Xilinx ZC702, the TTC0 is always secure and there is no way to configure its access directly form the non-secure guest OS. Despite the identified limitation on the TZPC, the pass-through policy without any support for shared devices is also somehow limited. This kind of implementation makes sense in the case of the secure VM, to promote real-time characteristics, but is very limitative in a system demanding for devices sharing among VMs, and it also disregards one of the followed design principles: the principle of least privilege. I plan to implement an hybrid approach in-between a pass-through and a paravirtualization strategy: the secure guest OS has direct and full control over the devices (pass-through model), but the non-secure VM requests access to devices via hypercalls, and the hypervisor mediates the access (paravirtualization). This model guarantees timing requirements of the real-time environment, promotes the principle of less privilege by controlling the non-secure guest OS devices' access while overcoming the dependency of the TZPC for configuring devices as non-secure.

One of the main advantages of TrustZone resides on the interrupt subsystem. The direct assignment of interrupts to each world, without intervention of the hypervisor, is a plus, but, most importantly, it does not increase the interrupt latency of the secure world once the RTOS gets executed. One small disadvantage that comes with this model is that slight modifications need to be introduced in the secure guest OS, in order to use interrupt handlers as FIQs instead of IRQs. In doing so, another problem on this specific platform arises: the decrease of performance on yield-intensive workloads. However, since this problem is very specific to this platform and precise workloads, I believe it should not be generalized.

The asymmetric design principle, which assigns to the secure VM a greater scheduling priority than the non-secure one, ensures the timing requirements of the real-time environment remains nearly intact, at the cost of integrating the hypervisor with the RTOS on the secure world side. In doing so, the RTOS has full control over the system, and can access or modify the state of the non-secure VM. Recently, Ngabonziza et al. [3.19] present some doubts about how my solution [3.20] could prevent the RTOS (secure world) from accessing the GPOS (non-secure world): in fact, it cannot; this is the price need to be paid to preserve the real-time demands of

the system, while keeping performance acceptable for low-end and low-cost devices. Anyway, two possible solutions to guarantee a higher degree of isolation on high-end devices are: run all guest OSes in the non-secure world side; or either paravirtualized the RTOS, so that it can run in the user mode of the secure world side, and mediate each memory access through the hypervisor. Another point outlined by Ngabonziza et al. is related to guest OSes preemption and consequent starvation. They argue in the proposed design "either OS cannot preempt the other OS". This is completely wrong; LTZVisor guarantees, by design, the secure guest OS (RTOS) preempts the non-secure guest OS (GPOS) as soon as a secure interrupt (FIQ) is triggered, but the reverse is not possible. So, starvation can happen, but only from the non-secure world side. However, despite this being a design decision to ensure the real-time needs, it is well-justified by the fact typical real-time applications have frequent idle times, which ensures the non-secure guest OS has enough CPU slices for execution. Ultimately, the scheduling policy can be designed accordingly to the applications needs, ensuring enough scheduling points that adequately meet the needs of both OSes, without compromising any real-time deadline; or either multicore platforms can be exploited to implement asymmetric multiprocessing (AMP) support.

3.7 Summary

Embedded real-time systems are proliferating at rapid pace in our everyday life, representing a huge part of our key infrastructures. The trend nowadays goes towards the consolidation of a wide range of functions into the same hardware platform, leading real-time requirements to coexist with non-real-time characteristics. Virtualization has been used as an enabler for platform consolidation whilst guaranteeing a robust functionality isolation, but the penalties incurred by existent software-based approaches bring forth the need of hardware-assisted solutions. Among existing COTS technologies, ARM TrustZone is attracting particular attention, due to its exclusive applicability on those ARM processors where VE are not available, while offering the best cost-benefit trade-off. The problem is that this technology is still seen with a lot of scepticism, which rose an urgent need to comprehensively examine the hype, myths, and realities of its usage for virtualization purpose.

In this Chapter, I described LTZVisor as a tool to clearly understand and evaluate how TrustZone hardware can be efficiently exploited to assist virtualization. I started to present an overview of LTZVisor, outlining its goals, design principles

and generic architecture. Then I provided concrete details about the implementation and deployment on a commercial Xilinx ZC702 board. I conducted an extensive set of experiments which demonstrated that this technology can effectively satisfy the strict requirements for virtualizing a real-time environment, while offering a low performance cost on running rich unmodified guest OSes. Finally, I presented an extensive discussion about the identified benefits and limitations, and how I think this limitations can be addressed and overcome; critics pointed by Ngabonziza et al. were also addressed and answered. Despite the identified limitations, the promising results encourage further research on this direction, mainly regarding the abolishing of the dual-OS limitation.

References

[3.1] G. Heiser, "Virtualizing embedded systems-why bother?," *Proceedings of the 48th Design Automation Conference (DAC)*, pp. 901–905, 2011.

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

[3.3] C. Lee, S. W. Kim, and C. Yoo, "VADI: GPU Virtualization for an Automotive Platform," *IEEE Transactions on Industrial Informatics*, vol. 12, pp. 277–290, Feb 2016.

[3.4] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," *Proceedings of the 11th Real-Time Linux Workshop*, 2009.

[3.5] A. Tavares, A. Didimo, T. Lobo, P. Cardoso, J. Cabral, and S. Montenegro, "Rodosvisor - An ARINC 653 quasi-compliant hypervisor: CPU, memory and I/O virtualization," in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012)*, pp. 1–10, Sept 2012.

[3.6] H. Joe, H. Jeong, Y. Yoon, H. Kim, S. Han, and H. W. Jin, "Full virtualizing micro hypervisor for spacecraft flight computer," in *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pp. 6C5–1–6C5–9, Oct 2012.

[3.7] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-based Secure Vir-

tualization Architecture," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pp. 209–222, ACM, 2010.

[3.8] C. Dall and J. Nieh, "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor," *SIGPLAN Not.*, vol. 49, pp. 333–348, Feb. 2014.

[3.9] T. Frenzel, A. Lackorzynski, A. W. H., and Härtig, "ARM TrustZone as a Virtualization Technique in Embedded Systems," *Twelfth Real-Time Linux Workshop*, 2010.

[3.10] S. Zampiva, C. Moratelli, and F. Hessel, "A hypervisor approach with real-time support to the MIPS M5150 processor," in *Sixteenth International Symposium on Quality Electronic Design*, pp. 495–501, March 2015.

[3.11] J. Winter, "Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms," in *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, STC '08, pp. 21–30, ACM, 2008.

[3.12] M. Cereia and I. Bertolotti, "Virtual Machines for Distributed Real-time Systems," *Comput. Stand. Interfaces*, vol. 31, pp. 30–39, Jan. 2009.

[3.13] D. Sangorrin, S. Honda, and H. Takada, "Dual operating system architecture for real-time embedded systems," in *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Brussels, Belgium*, pp. 6–15, 2010.

[3.14] S. Oh, K. Koh, C. Kim, K. Kim, and S. Kim, "Acceleration of dual OS virtualization in embedded systems," in *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*, pp. 1098–1101, Dec 2012.

[3.15] P. Varanasi and G. Heiser, "Hardware-supported Virtualization on ARM," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pp. 11:1–11:5, ACM, 2011.

[3.16] G. Labs, "An Exploration of ARM TrustZone Technology." Genode Operating System Framework.

[3.17] L. W. McVoy, C. Staelin, *et al.*, "lmbench: Portable tools for performance analysis.," in *USENIX annual technical conference*, pp. 279–294, San Diego, CA, USA, 1996.

[3.18] J. Taylor, "Security for the next generation of safe real-time systems," in

Proceedings of Embedded World Conference, Nuremberg, Germany, March 2016.

[3.19] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone Explained: Architectural Features and Use Cases," in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pp. 445–451, Nov 2016.

[3.20] S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares, "Towards a lightweight embedded virtualization architecture exploiting ARM TrustZone," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pp. 1–4, Sept 2014.

"Virtualization has many aspects attractive to the embedded world, but on its own is a poor match for modern embedded systems."

- Gernot Heiser

4

TZVisor: Beyond TrustZone Support

As a step towards the design of a secure and safe virtualization-based architecture, LTZVisor clearly demonstrated why ARM TrustZone technology is a key enabler for hardware-assisted virtualization. It was also possible to recognize, however, that this technologies present some limitations, specially regarding the consolidation of multiple guest OSes. This is being the main reason why some researchers still arguing that perceiving TrustZone as a virtualization mechanism is very limiting and ill-guided [4.1, 4.2].

In this Chapter, I will lift up this knowledge through the implementation of the TrustZone-assisted Hypervisor (TZVisor), a fully-featured virtualization environment providing complete hardware isolation among the multiple supported guest OSes. I describe how it is possible to multiplex more than one guest OS inside the non-secure world side, while providing all the details to handle shared hardware resources such as processor registers, memory, MMU and caches, devices and interrupts, and time. Presented use cases in the aerospace and industrial domains corroborate the viability of the proposed solution while proving the versatility for different configuration scenarios and application domains. The conducted evaluation process for each use case proved the possibility of running multiple unmodified

guest OSes on the non-secure world side with low performance cost, while proving the versatility of the generic architecture for fitting different application domains.

This Chapter is structured as follows: Section 4.1 describes the goals with the development of TZVisor, and Section 4.2 outlines its generic architecture. The generic implementation of the fully-featured virtualization architecture is described in Section 4.3. Sections 4.4 and 4.5 explains and evaluates specific implementations for two different application domains: aerospace and industrial, respectively. I discuss the main advantages and the identified limitations through an extensive discussion in Section 4.6; finally, the Chapter is summarized in Section 4.7.

Related Publications

The ideas and results presented in this Chapter have partly been published as:

- **S. Pinto**; J. Pereira; T. Gomes; M. Ekpanyapong; A. Tavares, "*Towards a TrustZone-assisted Hypervisor for Real Time Embedded Systems*", *in IEEE Computer Architecture Letters*, vol.PP, no.99, pp.1-1
- **S. Pinto**, A. Tavares and S. Montenegro, "*Space and Time Partitioning with hardware support for Space Applications*", *in Proceedings of the Data Systems In Aerospace (DASIA)*, Tallinn, 2016.
- **S. Pinto**, A. Tavares and S. Montenegro, "*Hypervisor for Real-Time Space Applications*", *in Proceedings of 4S Symposium*, Malta, 2016.
- E. Qaralleh, D. Lima, T. Gomes, A. Tavares and **S. Pinto**, "*HcM-FreeRTOS: Hardware-centric FreeRTOS for ARM Multicore*", *in Proceedings of the 2015 IEEE Conference on Emerging Technologies & Factory Automation (ETFA)*, Luxembourg, 2015, pp. 1-4.

4.1 TZVisor: Objectives

LTZVisor clearly demonstrated the feasibility of exploiting TrustZone to assist virtualization, however several limitations were identified, specially regarding the consolidation of more than two OS environments. The overall goal toward the development of TZVisor is to find a strategy that goes beyond the intrinsic TrustZone support to assist virtualization. In other words, the main goal is to provide a completely fully-featured virtualized environment which has roots on TrustZone hardware technology, but which overcomes its main identified limitations. This overall goal, can be split into several small objectives:

- ***Objective 1:*** Analyze how multiple (more than two) guest management can be achieved. Evaluate if it can be guaranteed by means of TrustZone hardware and quantify how much overhead it introduces;
- ***Objective 2:*** Analyze how memory isolation can be achieved regarding the execution of multiple guest. Evaluate which mechanisms are provided by means of TrustZone hardware, as well as the introduced overhead;
- ***Objective 3:*** Analyze how caches and MMU can be shared. Evaluate if it can be done by means of TrustZone hardware and quantify how much overhead it introduces;
- ***Objective 4:*** Analyze how device partition can be achieved. Evaluate which mechanisms are provided by means of TrustZone hardware and how much overhead it introduces;
- ***Objective 5:*** Analyze how interrupts for multiple guest OSes can be managed. Evaluate which mechanisms are provided by means of TrustZone hardware and how much overhead it introduces;
- ***Objective 6:*** Experiment how the proposed solution behave for distinct use case scenarios with different criticality requirements;

4.2 TZVisor: General Architecture

TZVisor provides a fully featured virtualization solution that allows the execution of multiple guest OSes on TrustZone-enabled platforms. The secure world is responsible

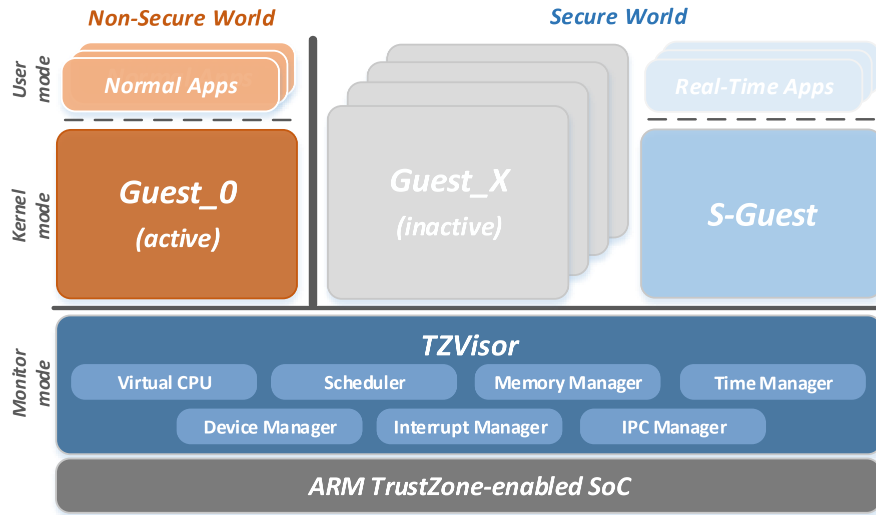


Figure 4.1: TZVisor: general architecture

for hosting the secure VM, as well as for preserving inactive non-secure VMs. The non-secure world is responsible for hosting the active non-secure VM. Figure 4.1 depicts the proposed virtualization architecture.

TZVisor runs in the highest privileged processor mode, i.e., in monitor mode. The hypervisor is composed of seven main components: the CPU manager, the scheduler, the memory manager, the time manager, the device manager, the interrupt manager, and the inter-partition communication manager. The CPU manager is responsible for managing the VMCB of each VM during a partition switch, while the memory manager is responsible for configuring memory access security accordingly to the active non-secure VM. The scheduler ensures the higher privilege of execution of the secure VM, as well as the correct temporal isolation between the multiple non-secure VMs. The device manager is responsible for configuring the security state of devices accordingly to the active non-secure VM, and the interrupt manager is responsible for managing interrupts of multiple guests. The time manager is responsible for dealing with time for all guest OSes and ensure all guests have the notion of the real passage of the time when they are launched for execution. The inter-partition communication manager provides a communication channel between the several VMs.

The secure VM runs in the supervisor mode of the secure world side. In TZVisor architecture, this VM is an optional component of the overall architecture, due to the weakness it presents in terms of spatial isolation. Since the secure VM runs in the secure world side, the processor has full view over the secure and non-secure worlds side, which can interfere with the other VMs by accessing or modifying their states or states of their resources (memory or memory mapped devices). For this reason,

the operating system hosted on the secure VM is considered part of the system's TCB, which can significantly increase the attack surface. For safety-critical systems, such as airplanes and spacecraft, this property can be seen with some scepticism, and this is why in the TZVisor system architecture the secure VM is an optional component.

The active non-secure VM runs in the non-secure world side. The OS kernel runs in the supervisor mode while the applications in the user mode. The software running on the non-secure world side is completely isolated from the privileged software running on the secure world side. When the processor is operating in a privileged mode but not in the secure state, it cannot access nor modify any state information belonging to the secure world. Any attempt from the non-secure guest OS to access any resource of the secure world side immediately triggers an exception to the hypervisor. The inactive non-secure VMs are preserved in the secure world side. Since only one guest can run at a time, there is no possibility for inactive guests (belonging momentaneously to the secure side) to change the state of another guest.

4.3 TZVisor: Implementation

TZVisor goes beyond ARM TrustZone support to provide time and space partition between multiple guest OSes. This Section provides all the details behind the hypervisor implementation, explaining how to multiplex several guest OSes inside the non-secure world side. The support is guaranteed by carefully handling shared hardware resources such as processor registers, memory, caches and MMU, and interrupts and devices. Processor registers are saved and restored into/from a specific VMCB, while memory isolation is ensured through the dynamic memory configuration feature of the TZASC.

4.3.1 Guest Management

LTZVisor relies on the TrustZone hardware support to minimize the number of registers to be saved and restored in each partition-switching operation (please refer to Section 3.3.1). The fact of the supported VM coinciding exactly with the number of isolated states directly supported by the processor, means the solution significantly rely on hardware to manage the execution context of each VM, because there is a list of registers which an individual copy of them exists for each world. TZVisor

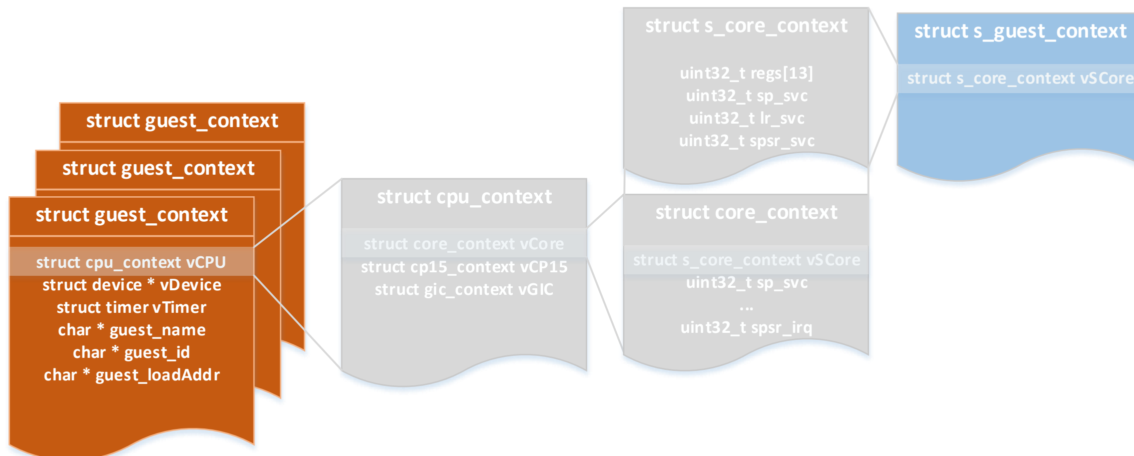


Figure 4.2: TZVisor: guest management

supports multiplexing of multiple guest OSes in the non-secure world side, which cannot rely only on the TrustZone hardware. Therefore, carefully handling of several shared resources need to be done by software.

Figure 4.2 depicts the data structure for the non-secure guest OSes as well as the secure guest OS. The secure guest OS, due to its uniqueness and exclusivity, is just composed of the secure core specific data structure. The secure core context includes 16 registers: 13 general-purpose registers (*R0-R12*), the Stack Pointer (*SP*), the Linker Register (*LR*) and *SPSR* for the System mode. Comparing with the VMCB of the secure side in LTZVisor (please refer to Section 3.3.1) no modification were required. The *SP*, the *LR* and the *SPSR* of the FIQ mode continue to be exclusively dedicated to the secure world.

On the other hand, the non-secure guest OS context needs to be managed in a completely different way. Since the non-secure OS might handle several guest OSes, not only the hardware context need to be part of the data structure, but also several guest attributes such as the guest name and identification, the list of devices assigned to a partition, the virtual timer, as well as the guest load address. This attributes are essential to handle shared resources, as well as to configure the memory during the context-switch operation. The CPU data structure includes not only the CPU core, but also the CP15 and the GIC data structures. The CPU core includes the context of the optimized secure core and some extra register for other CPU execution modes, encompassing a total of 28 registers: 13 *General Purpose Registers* (*R0-R12*), the *SP*, the *LR* and the *SPSR* for the Supervisor, System, IRQ, Abort and Undef modes. Comparing to the VMCB of the non-secure side in LTZVisor, the IRQ mode have to be preserved as several guest OSes co-exist in the non-secure world side. The CP15

data structure is composed of 20 registers, including the System and the Auxiliary Control registers, and the Translation Table registers. Listing 4.1 presents part of the CP15 data structure.

Listing 4.1: Guest management: CP15 data structure

```
typedef struct cp15_context{
    uint32_t CSSELR; // Cache Size Selection
    uint32_t SCTLR; // System Control
    uint32_t ACTLR; // Auxiliary Control
    uint32_t TTBR0; // Translation Table Base 0
    uint32_t TTBR1; // Translation Table Base 1
    uint32_t TTBCR; // Translation Table Base Control
    uint32_t DACR; // Domain Access Control
    /*{...}*/
    uint32_t TPIDRURW; // User Read/Write Thread ID
    uint32_t TPIDRURO; // User Read-only Thread ID
    uint32_t TPIDRPRW; // Privileged only Thread ID
} VCP15;
```

4.3.2 Scheduler

TZVisor extends the LTZVisor scheduler to implement a two-layered or hierarchical scheduling policy: the first layer is responsible to schedule the secure guest OS for guaranteeing its hard timing requirements are completely met; the second layer is responsible to schedule the several non-secure guest OS in such a way starvation cannot happen between the multiplexed non-secure guest OSes. The first layer implements an asymmetric or idle scheduler. This scheduling policy guarantees that the non-secure guest OS is only scheduled during the idle periods of the secure guest OS, and the secure guest OS can preempt the execution of the non-secure one. The second layer, on the other hand, implements a cyclic scheduling policy. This scheduling policy ensures a non-secure partition cannot use the processor for longer than its assigned CPU quantum. The time of each slot can be different for each partition, depending on partition criticality classification, and is configured at design time. By adopting a variable time slot strategy instead of a multiple fixed approach, the hypervisor interference is minimized and it is ensured higher performance and deterministic execution, because partition is only interrupted when the complete slot is over.

4.3.3 Memory Partition

LTZVisor provides memory isolation relying on the TZASC. Partitions are configured at design time, and the memory is configured during boot time: one memory segment for the hypervisor and the secure VM, and the remain fifteen memory segments to the non-secure VMs. As explained in Section 3.3.3, on Xilinx ZC702, memory segments can be configured with a granularity of 64MB. Due to the static nature of the system, there is no need to change the configuration of memory assigned to each guest during runtime.

TZVisor exploits the dynamic memory segmentation feature of the TZASC to implement robust (hardware-enforced) spatial isolation between the multiple non-secure guest OSes, basically by dynamically changing the security state of the memory segments of partitions. Only the partition that is currently running (in the non-secure side) must have its own(s) memory segment(s) configured as non-secure, and the remaining memory as secure. If the running partition tries to access a secure memory region (belonging to an inactive partition or either to the hypervisor), an exception is automatically triggered and redirected to the hypervisor. Since only one guest can run at a time, there is no possibility of the inactive partitions (belonging momentously to the secure side) to change the state of another partition. Figure 4.3 depicts the memory setup and respective secure/non-secure mappings, for a virtualized system consisting in the hypervisor and several partitions. In this specific configuration, the hypervisor uses the first memory segment (0x00000000 - 0x03FFFFFF), and has access to all memory. Guest OS 0 uses the second 64MB memory segment, and is only allowed to access one non-secure memory segment

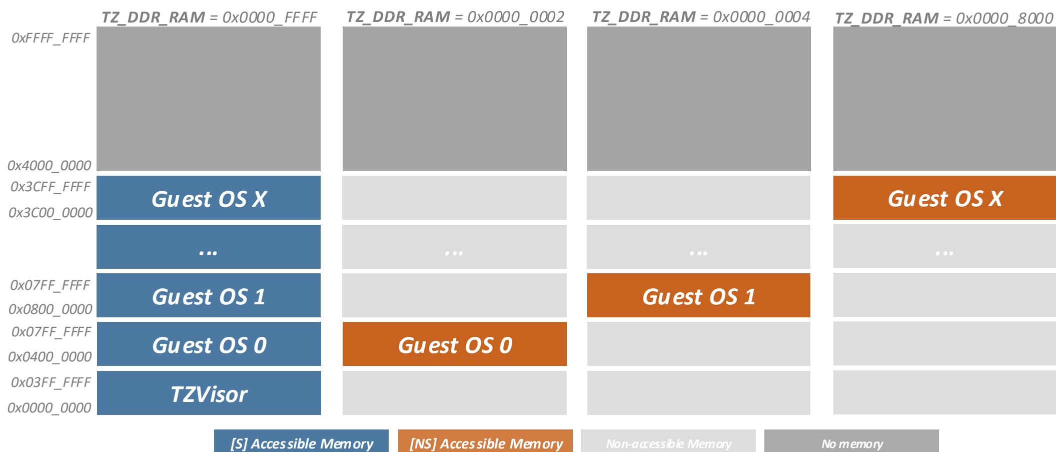


Figure 4.3: TZVisor: system memory map

(0x04000000 - 0x07FFFFFF); remaining guest OSes are mapped the same way, but within their respective memory segment.

While memory partition in LTZVisor is configured once during boot time, in TZVisor memory isolation is managed at multiple stages: it starts from the hypervisor initialization, and then continues during runtime. During the hypervisor initialization all memory segments are set as secure (`TZ_DDR_RAM` configured with 0x00000000 value). Then the hypervisor is responsible for creating all necessary partitions. This specific API is responsible for, among other operations, copying the guest OS image (included in the TZVisor image) to the specific memory segment it was assigned during compiling and linking time. The load address is verified before the memory copy starts. As explained in Section 3.3.3, TrustZone-enabled SoCs (which are not VE-enabled) only provide MMU support for single-level address translation: guests have to know the physical memory segment they can use in the system, which mean they need to be compiled and linked to a specific memory map. After all guest OSes being loaded for the specific memory segments, the hypervisor is responsible for starting the first partition. Before launching the first guest, the hypervisor changes the configuration of the `TZ_DDR_RAM` register. The load address (part of the VMCB information) is used to match with the specific memory segment, which is then used to extract the value of the memory configuration to be used from a static table which has all available configurations. The same process is used during each guest-switching operation. Listing 4.2 presents the routine responsible for configuring the memory accordingly to the configuration value extracted from the static table.

Listing 4.2: Memory partition: reconfiguring memory security

```
void memory_config(uint32_t value){
   // Unlocking SLCR register ...
   write32(SLCR_UNLOCK, SLCR_UNLOCK_KEY);
   // Configuring memory ...
   write32(TZ_DDR_RAM, value);
   // Locking SLCR register ...
   write32(SLCR_LOCK, SLCR_LOCK_KEY);
}
```

4.3.4 MMU and Cache Management

LTZVisor relies on the TrustZone hardware support to guarantee the cache and virtual space coherence between the secure and the non-secure guest OSes. The existence of two distinct MMU interfaces as well as secure and non-secure cache entries are crucial to the absence of any MMU and cache related operations during a guest switch, which keeps performance overhead really low. TZVisor cannot rely just on the TrustZone hardware support to guarantee the cache and virtual space coherence among all non-secure guest OSes. The multiplexing of several guests OSes on the non-secure world side requires the hypervisor to clean and invalidate cache information and TLB entries before changing the active non-secure guest OS. Accordingly to the TrustZone specification, for MMU and cache maintenance operations take effect over the non-secure world side they can be performed either from the non-secure or from the secure side. Therefore, I envisioned three different approaches for implementing this support:

- *Copy the maintenance routine for a specific non-secure memory segment during boot time, and during the guest switch operation run the maintenance routine from the non-secure world side* - this approach offers a compromise between performance and world-interference, however it can considerably impact the system safety;

- *Copy the maintenance routine to a non-secure memory segment on each non-secure guest OS switch, before running the maintenance routine from the non-secure side* - this approach offers a good trade-off between safety and world-interference, however increases considerably the context-switch time due to the copying process;

- *Integrate the maintenance routine into the hypervisor and execute the maintenance operation in the secure world side* - this approach offers the best trade-off between safety and performance at the cost of slightly increasing on the TCB, while affecting the use of caches on the secure world side;

Between the aforementioned approaches there are different trade-offs on performance, world-interference and safety. The first one offers a good performance-interference ratio, however it can impact significantly the system safety. This is mainly due to the need for executing the maintenance operations forces the hypervisor to momentously jump to the non-secure world side, and then it relies on the maintenance code to return to the secure world side. If a non-secure guest OS (with

complete access to all non-secure memory segments) is compromised, it can simply modify this code or even just executing the returning instruction, leading the hypervisor to enter in an undefined state. This is completely unacceptable, because it can lead the entire system to fail. The second approach solves the safety problem at the cost of performance. The need for copying the maintenance routine at every guest switch will definitively decrease the performance by a considerable factor. Taking this arguments into consideration, the last approach was considered as the one offering the best trade-off between all aforementioned metrics, specially because the system typically runs with caches disabled on the secure world side.

A particular observed phenomenon is that the use of the dynamic configuration feature of the TZASC to ensure spatial isolation imposes the maintenance operation as a requirement, instead of a prevention mechanism for avoiding cache information leakage. During the implementation of this strategy it was observed that if the caches are not flushed, the system enters in a data abort exception. This happens because of the incoherence that exists between the data in cache and the data in memory. In the deployed platform, caches follows a write-back policy: if the cache is not cleaned and invalidated the data still resides there, and is tagged as non-secure, while the same information belongs now to a secure memory area. This incoherence of the security state of data leads the processor to trigger an exception that redirects the execution flow to the data abort handler of the monitor (hypervisor) vector table.

Listing 4.3: MMU and cache management: low-level maintenance routine

```
.global nsw_inv_tzvisor
nsw_inv_tzvisor:
   bl data_l1cache_flush_all // Flush L1 D-Cache
   bl unified_l2_cache_flush_all // Flush L2 Cache
   mov r0, #0
   mcr p15, 0, r0, c7, c5, 0 // Invalidate I-Cache (ICIALLU)
   mcr p15, 0, r0, c8, c7, 0 // Invalidate I-TLB (TLBIALL)
   mcr p15, 0, r0, c8, c3, 0 // Invalidate TLB (TLBIALLIS)
   mcr p15, 0, r0, c7, c5, 6 // Invalidate BP (BPIALL)
   dsb
   isb
   bx lr
```

Listing 4.3 presents the MMU and cache low-level maintenance code. The routine is responsible for flushing (clean and invalidate) the L1 data cache, as well as the unified (data and instruction) L2 cache. The remainder of the routine include invalidating

the L1 instruction cache, translation lookaside buffers, and all entries from branch predictors.

4.3.5 Device Management

One of the identified limitations on LTZVisor was the implemented device virtualization approach which disregards the principle of least privilege. The non-secure guest OS, similarly to the secure guest, uses a pass-through policy, which means the guest has complete access and control over the device. This approach makes sense from the secure guest OS perspective, but not from the non-secure one.

TZVisor extends the previous device virtualization approach in the sense that the non-secure guest OS uses paravirtualization to access devices. All devices, except the TTC1 (please refer to Section 3.3.7 for details), are configured as secure, guaranteeing the non-secure guest OS cannot access the hardware resources directly. Instead, the access driver from the non-secure guest OS is slightly modified to send requests to the hypervisor in the secure world side, which is responsible for relegating the device access to the secure world OS or even mediate the access directly from the monitor. Frenzel et al. [4.3] proposed a paravirtualization strategy which relies on four main components: (1) the non-secure access driver to issue the requests, (2) the hypervisor to mediate the communication, (3) the unprivileged virtual machine monitor (uVMM) to provide the virtual platform for the non-secure guest OS, and (4) the secure driver to access the secure hardware device. The control flow from TZ-Linux to access a secure device encompasses five steps:

1. The access driver in the non-secure world issues a request to the uVMM in the secure-world OS;
2. The hypervisor saves the normal-world processor state into the VM state and sends a message to the uVMM;
3. The uVMM selects the virtual device and handles the read or write request through the use of a secure driver;
4. After finishing the request the uVMM sends a reply message to the hypervisor;
5. The hypervisor restores the normal-world processor state from the VM state and initiates the normal-world entry;

TZVisor implements a different strategy, following a lightweight paravirtualization

approach. The implemented approach relies on two main components: (1) the non-secure access driver to issue the requests, (2) and the hypervisor to mediate the communication and the access to the device. Since it is the hypervisor that directly accesses the secure device, no save and restore operations of the non-secure world side need to be performed. The control flow from the non-secure guest OS to access a secure device encompasses just three steps:

1. The accessed driver in the non-secure world issues a request (through an SMC, see Listing 4.5) to the hypervisor in the secure-world;

2. The hypervisor perform some verifications (e.g., verify if the guest OS can access the device) and handles the read or write request;

3. After finishing the request, the hypervisor prepares the return message and initiates the non-secure world entry;

Listing 4.4 depicts the code that should be added to each non-secure guest OS device driver in order to paravirtualize the access. The raw functions are replaced by secure functions that are responsible for explicitly trigger a request from the non-secure world side, through the use of the SMC instruction. The information regarding the address to be accessed, as well as the value (for a write operation please see Listing 4.5) is passed through the R1 and R2 registers, respectively. The R0 register specifies the SMC call id, i.e., the reason behind the SMC call which in fact indicates the action required to be performed by the hypervisor. The hypervisor receives the request, verifies if the address is a valid one in the context of the assigned devices to the active VM, and if valid then accesses the device. No save and restore operation are performed in the meantime, once is the hypervisor that directly deals with the secure device access.

Listing 4.4: Device management: paravirtualization patch

```
+#ifndef NONSECURE_HW_ACCESS
+#undef __raw_readl
+#undef __raw_writel
+#define __raw_readl(addr) \
+  secure_read(addr) // Paravirtualized read
+#define __raw_writel(val, addr) \
+  secure_write(val, addr) // Paravirtualized write
+#endif
+
+#endif
```

Concluding, while the implemented device access in LTZVisor guarantees the best performance, the paravirtualized approach proposed by Frenzel et al. [4.3] guarantees the safest approach, by ensuring, at the highest level, the principle of least privilege. The solution presents a better trade-off between performance and safety, at the cost of a slight increase of the TCB. The non-secure guest OS cannot access the devices directly, as it is the hypervisor responsible for dealing with the access.

Listing 4.5: Device management: secure write

```
+ENTRY(secure_write)
+   mov r2,r0 // Copying val to r2
+   ldr r0,=-31 // SMC call id - write
+   dsb
+   dmb
+   smc #0
+   bx   lr
+ENDPROC(secure_write)
```

4.3.6 Interrupt Management

Following the same approach implemented in the lightweight version of the hypervisor, TZVisor configures secure interrupts as FIQs, and non-secure interrupts as IRQs. Secure interrupts are redirected to the hypervisor, while non-secure interrupts are redirected to the active guest. Furthermore, while a guest partition is under execution, all interrupts belonging to the active guest are locally and directly managed by the OS without any hypervisor interference. The problem which arose with the multiplexing of several guest OSes over the non-secure world side, is how to manage interrupts of inactive non-secure guest OSes. This problem is even more critical if an interrupt for a (hard) real-time guest partition arrives when the active guest is running. To work around this problem, three main approaches were implemented to cope with different levels of criticality of non-secure guest OSes:

- *Defer the interrupt handling until the target guest is next scheduled for execution*;
- *Force immediately a context switch to the target guest to handle the interrupt*;
- *Handle the interrupt directly from the hypervisor (even as a user privilege handler)*;

Between the aforementioned approaches there are different trade-offs on interrupt latency and inter-guest interference. The first approach does not impact on guest-interference, but it considerably increases the interrupt latency. If an interrupt for an inactive guest OS arises, the interrupt will be pending for so long on the device, and will be dispatched and handled as soon as the inactive guest OS is scheduled. In the meantime, if several other inactive guest OSes are queued to be scheduled before, the interrupt latency will be directly proportional to the accumulative CPU quantum dedicated to the other guests. While this approach can be seen as acceptable when managing general-purpose guest OSes, it falls short when strict timing requirements are demanded. Taking this into consideration, the second approach presents a good ratio between interrupt latency and guest interference. If an interrupt for an inactive guest OS arises, the hypervisor take control of execution, and switches to target guest OS. This approach will affect the execution of the running guest OS, but will decrease interrupt latency to the amount of time of a full guest-switch operation. Finally, the last approach is a particular extension of the second one, addressing the reduction of interrupt latency nearly the native. If an interrupt for an inactive guest OS arises, the hypervisor take control of execution and immediately handles the interrupt at the hypervisor level. This method affects the execution of the running guest OS, but will decrease interrupt latency to the minimum possible value. However, it presents some limitations because it cannot be applied in all situations. For instance, it prevents the call of OS-specific APIs in the interrupt handler.

A virtual structure (vGIC) keeps the configuration of the GIC for each non-secure guest OS. When a guest switch operation occurs, the hypervisor is responsible for adequately enabling and disabling IRQ sources, as well as reconfigure the interrupt sources as secure or non-secure. It just depends on the interrupt handling mechanism adopted for a specific VM.

Listing 4.6: Interrupt management: restoring virtual GIC

```
void vgic_context_restore(VGIC * p_vgic){
    /*{...}*/
    for(i=0;i<GIC_NUM_REGS;i++){
        temp_en = gic_enable_get(i);
        gic_enable_set(i, (temp_en|p_vgic->vICDISERn[i]));
    }
}
```

Listing 4.6 presents part of the `vgic_context_restore(...)` routine belonging to

the hypervisor interrupt management module, which is responsible for reconfiguring the GIC according to the virtual GIC information. In this particular snippet, it encompasses reading the state of the interrupt set enable register (`ICDISERn`), for keeping the enabled FIQ sources while enabling the IRQ sources for the next running guest.

In sum, TZVisor provides a fine-grained configuration over the handling mechanism that should be used for the inactive guests according to the partition criticality level. All of those features are configured statically at design time. The hypervisor provides the means for real-time guarantees but it is not responsible for any misuse of them: the system designer is the one responsible for configuring and tuning them according the application needs.

4.3.7 Time Management

LTZVisor provides a distinct time management implementation, basically by dedicating one independent timing unit for each guest OS: secure VM uses the TTC0, and the non-secure VM uses the TTC1. While this approach seems adequately for a dual-OS configuration, it does not scale well when the number of supported VMs increase, because the available hardware resources are not enough.

TZVisor extends the previous time management approach by presenting a scalable time management strategy for dealing with the multiple non-secure guest OSes. The secure VM, due to its uniqueness regarding the overall system, has a dedicated timing unit: one timer from the TTC0. This dedicated hardware assignment ensures the real-time guarantees and even complete support for tick-driven Operating Systems. This timer is also used to provide strong temporal isolation between the multiple non-secure VMs. This means that when the secure VM is present in the system, is the secure guest OS tick that is responsible for creating scheduling points which guarantee one partition cannot use the processor for longer than its defined CPU quantum. When the secure VM is not present in the system, this timer is assigned to the hypervisor, and is responsible for creating synchronous scheduling points at the hypervisor level.

For managing the time of the multiple non-secure VMs a two-level strategy was followed. It basically consists in replicating the timing units needed by the non-secure guest OSes at the hypervisor level. At the partition level, whenever the active guest is executing, timers belonging to the guest are directly managed and

updated by the guest OS. The problem lies in how to deal and handle time from inactive guests. For inactive guests the hypervisor implements a virtual tickless timekeeping mechanism based on time unit(s) that measures the passage of time. Therefore, when a guest is rescheduled, its internal clocks and related data structures are updated with the elapsed time since its previous execution: this ensures all guest OSes have the real notion of the passage of the time. The replicating strategy also ensures if a compromised guest intentionally changes the state of the timing facilities, or even configures with different properties, such actions do not interfere with others VMs because related timers and data structures are updated in every guest switch. This strategy imposes, however, two limitations: first, the non-secure guest OSes have to follow a tickless time management strategy; second, the time management strategy is very guest-specific and require some adaption from OS to OS.

4.3.8 Inter-VM Communication

Inter-VM communication provides a transparent virtual mechanism for implementing communication between different VMs. In contrast with other solutions, which follow an non-standard approach [4.4, 4.5, 4.6], TZVisor uses the standardized VirtIO [4.7] as a transport abstraction layer. VirtIO has been used in several implementations targeting I/O virtualization [4.7, 4.8], and has recently started being adopted to implement inter-guest [4.9] and inter-processor communication on multicore platforms (e.g., Texas Instrument Remote Processor Messaging and Mentor Graphics MEMF) [4.10].

TZVisor implements an adaptation of the Remote Processor Messaging (RPMsg) API from the Texas Instrument and OpenAMP group to a supervised single-core architecture. The implementation from Texas provides the foundation for implementing communication on top of general-purpose guest OSes, while the implementation from OpenAMP provides the foundation for a bare-metal approach. The main modifications encompass: (1) the complete elimination of the remote processor executable loader and processor life cycle management since it is supported by the hypervisor; (2) the refactoring of VirtIO device configuration as it is implemented statically and configured at boot time; and (3) the implementation of the RPMsg slave mode support following also the VirtIO standard.

Figure 4.4 depicts the communication architecture, where a data path and an event channel provides the means to implement point-to-point or guest-to-guest communication. As it can be seen, the data path is completely isolated from the event path, a

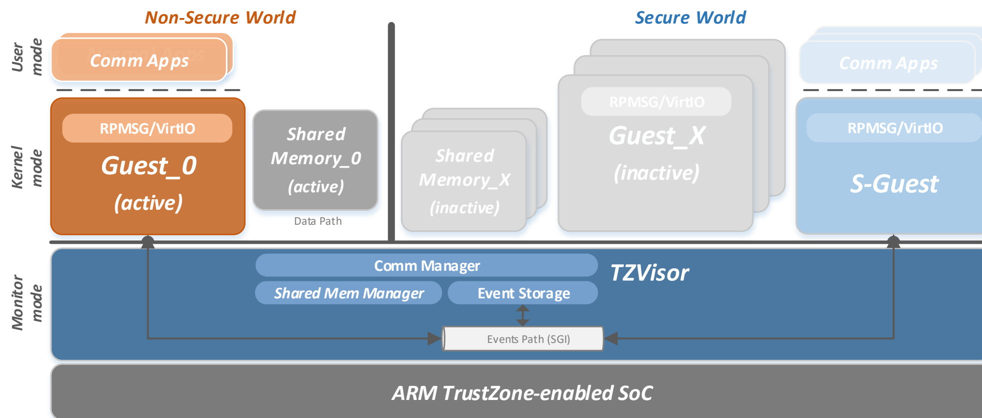


Figure 4.4: TZVisor: inter-VM communication

design decision that promotes asynchronous communication, essential to guarantee the timing requirements of the secure VM. The data path is defined by a shared block of memory. Each guest OS has a dedicated shared memory block, which is only visible for the pair of guests that establishes a communication channel. Isolation at the data path is enforced, once again, by exploiting the dynamic memory segmentation feature of the TZASC to configure memory regions as secure or non-secure during runtime. This design decision introduces a higher level of security at the cost of memory footprint, once the granularity of DRAM memory segments is too high (64MB). The OCM RAM memory can also be used to implement smaller memory blocks, because granularity at the OCM level is 4KB. The event path is defined by SGIs routed through the hypervisor. This mechanism is based on requests from guest OSes to the hypervisor, via the SMC instruction. All requests are stored in a circular buffer, following a first-in, first-out policy. During each partition switch, TZVisor triggers SGIs to the respective guest OSes, enabling asynchronous notifications. In spite of a slight degradation of the partition-switching time, this trade-off guarantees the reliability of the communication as the hypervisor has full control over every transaction.

At the hypervisor level, the inter-VM communication support encompassed the addition of three main building blocks: the communication manager, the shared memory manager, and the event storage. The communication manager is responsible for managing all the logic for sending and receiving messages to and from different guest OSes. The shared memory manager is responsible for configuring the accessibility of the shared memory blocks according to the active guest OS. The event storage module works as a buffer that caches all SGIs that must be sent to the respective guest OS. Listing 4.7 presents part of the communication manager module called during each guest-switch, which is responsible for triggering an event to the next

active guest OS notifying that a message need to be managed at guest level.

Listing 4.7: Inter-VM communication: triggering SGI routine

```
void send_monitor_sgi(guest_context *guest)
{
   /*{...}*/
   if( guest->vcpu_comm->idx ==  guest->vcpu_comm->tail)
      return; //empty circular buffer

   idx_tmp = guest->vcpu_comm->buffer[guest->vcpu_comm->idx]
   guest->vcpu_comm->on_buffer[idx_tmp] = false;
   _send_monitor_sgi(idx_tmp, guest);
   guest->vcpu_comm->idx++;
   guest->vcpu_comm->idx &= ~(SGI_NR);
}
```

4.4 Aerospace Safety-Critical Use Case

The market of complex and safety-critical systems like airplanes and spacecraft have experienced unprecedented growth over the last few years and is expected to continue growing exponentially for the foreseeable future [4.11]. The number and complexity of desired functions evolved in such a way that fully federated architectures, where each function is implemented in its own embedded controller became completely impracticable. Naturally, industries rapidly tried to find other alternatives, and aeronautics pioneering the shift from traditional federated architectures to an integrated modular avionics (IMA) [4.12] architecture. By combining several applications into one generic powerful computing resource, they were able to get a reduction on size, weight, power and cost.

As space domain typically shares the same basic needs of aeronautics, they rapidly concluded that IMA strategy could be spun-in to the space domain. The problem was that the use of generic platforms altogether with several COTS-based components with different criticality and from several suppliers, dictated integration challenges namely in terms of reusability and safety. The introduction of space and time partition (STP) [4.13, 4.14], for separation of concerns between functionally independent software components, was the solution to achieve higher level of integration while maintaining the robustness of federated architectures. By containing and/or isolating faults, STP approach eliminates the need to re-validate unmodified

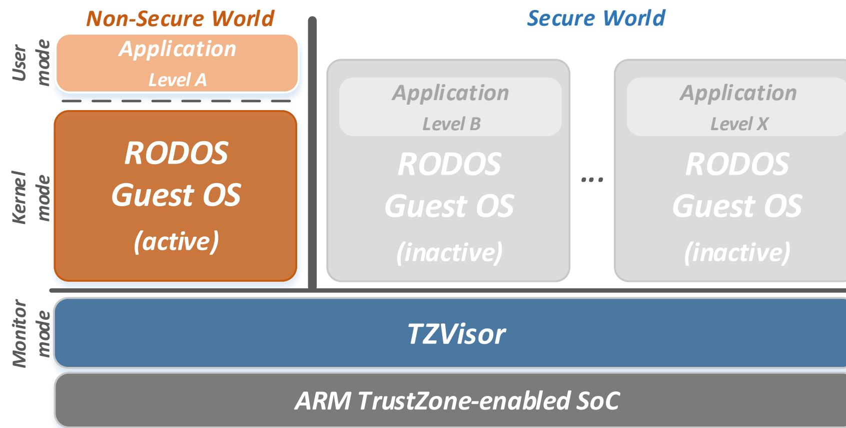


Figure 4.5: TZVisor for Aerospace: system architecture

applications on an IMA system, as the guaranteed isolation it provides limits re-certification efforts only at the partition level. Virtualization technology has been used as an implementation technique to provide STP. Over the last few years several works have been proposed in the aerospace domain [4.15, 4.16, 4.6, 4.17, 4.18], but, to the best of my knowledge, none of them assisted by COTS hardware.

This Section presents TZVisor for Aerospace, a specific configuration of TZVisor general architecture to meet the strict safety requirements of the Aerospace industry. It is demonstrated the effectiveness of the presented solution by running several RODOS OS partitions on the Xilinx ZC702 board. An extensive evaluation process is conducted to access the virtualized TCB and the performance overhead.

4.4.1 Implementation

Figure 4.5 presents the architecture of TZVisor for Aerospace. Comparing this specific configuration with the TZVisor general architecture, the first visible change is the complete elimination of the secure VM. The strict and robust isolation requirements of aerospace industry impose some restrictions and also limit some decisions. The secure world side is exclusively dedicated to the hypervisor and the several inactive VMs. This configuration decision reduces the TCB of the system to the hypervisor component, because is the only piece of software that has complete access to the system. The complete elimination of the secure VM also has impact on the scheduling policy, which better fits the strong temporal isolation demands of safety-critical systems. The hypervisor scheduler implements only the cyclic scheduler, ensuring a partition cannot use the processor for longer than its defined CPU quantum. The time of each slot can be different for each partition, depending on

partition criticality classification, and it is configured at design time.

To ensure strong spatial isolation between partitions only the active partition (in the non-secure side) has its own memory segment configured as non-secure, and the remaining memory as secure. Caches, TLBs and branch predictors interfaces on the non-secure side are clean and invalidated at every guest switch, when the support for guest OSes is enabled at design time. To achieve also strong isolation at device level, devices are not shared between partitions, and are assigned at design time. For interrupts, secure interrupts are redirected to the hypervisor, while non-secure interrupts are redirected to the active guest (without hypervisor interference). When a partition is under execution, only the interrupts managed by this partition are enabled, which minimizes inter-partition interferences through hardware. Interrupts of inactive partitions are momentaneously configured as secure, and consequently redirected to the hypervisor.

For time management purpose the hypervisor implements two levels of timing. The hypervisor manages two timers: one 32-bit timer unit (TTC0) for the hypervisor tick, and one 64-bit timer (Cortex-A9 MPCore Global Timer) to keep track of the wall-clock time. Timers dedicated to the hypervisor are configured as secure devices, i.e., they have higher privilege of execution than timers dedicated to the active guest. This means that despite of what is happening in the active guest, if an interrupt of a timer belonging to the hypervisor is triggered, the hypervisor takes control of the system. The partition manages two timers: one 32-bit timer unit (TTC1) to keep track of the wall-clock time (altogether with a 32-bit soft-time unit for building a virtual 64-bit timer), and one 32-bit timer unit (TTC1) for the RODOS tick. Whenever the active guest is executing, timers belonging to the guest are directly managed and updated by the guest OS (the timer units dedicated to the partition are the only devices configured as non-secure). For inactive guests the hypervisor implements a virtual tickless timekeeping mechanism based on a time-base unit that measures the passage of time (the aforementioned 64-bit timer). Therefore, when a guest is rescheduled, its internal clocks and related data structures are updated with the time elapsed since its previous execution.

The Health Monitor (HM) component is the module responsible for detecting and reacting to anomalous events and faults. Although at an early stage of development, once an error or fault is detected, the hypervisor reacts to the error providing a simple set of predefined actions. The complete list of events and pre-defined actions can be seen in Table 4.1.

Table 4.1: TZVisor for Aerospace: health monitoring events and actions

Event name	***Hypervisor*** *pre-def. action*	***Guest*** *pre-def. action*
Guest triggered		
DATA_ABORT	—	Reboot
PREF_ABORT	—	Reboot
UNDEF_INST	—	Reboot
Hypervisor triggered		
MEM_VIOL	Log	Reboot
DEV_VIOL	Log	Reboot
NO_GUESTS	Reset	—

4.4.2 Execution Flow

The system starts with the secure world boot process. This procedure is responsible for a set of operations which includes specific processor and coprocessor registers initialization, vector tables setting, stacks configuration, and MMU, cache and branch predictor initialization (disable). Once the secure world boot process is finished, the hypervisor starts executing.

The hypervisor starts by initializing and configuring the platform-specific hardware. This procedure is responsible for configuring memory, devices and interrupts: all memory segments are configured as secure; all devices are configured as secure except the TTC1, because it will be shared among all non-secure guest OSes for time management purpose (as explained in Section 4.3.7); the GIC is initialized, secure interrupts are configured to use the FIQ interrupt mechanism of the processor, and all interrupt sources are configured as secure. Then, the hypervisor initialize some internal data structures and control variables, and creates the respective VM(s). The VM creation includes the initialization of the VMCB, as well as the loading of the respective guest OS image to the specific memory segment it was assigned during compilation and linking time. Once the hypervisor is initialized and the VM(s) created, the scheduler is started.

The scheduler starting routine is responsible for selecting the first guest to run (the first guest created) and to start the hypervisor timing units. The next step performs the non-secure world restore operation. This includes restoring (initializing for the first time) CP15- and GIC-specific registers, as well as configure the security state of the memory, and update the guest-specific timing structures. The last part of the restore operation ensures the core registers are correctly loaded with a VMCB information. At the end, the hypervisor enables the NS bit and jumps to the non-secure world address. Once on the non-secure side, the guest OS will run until the

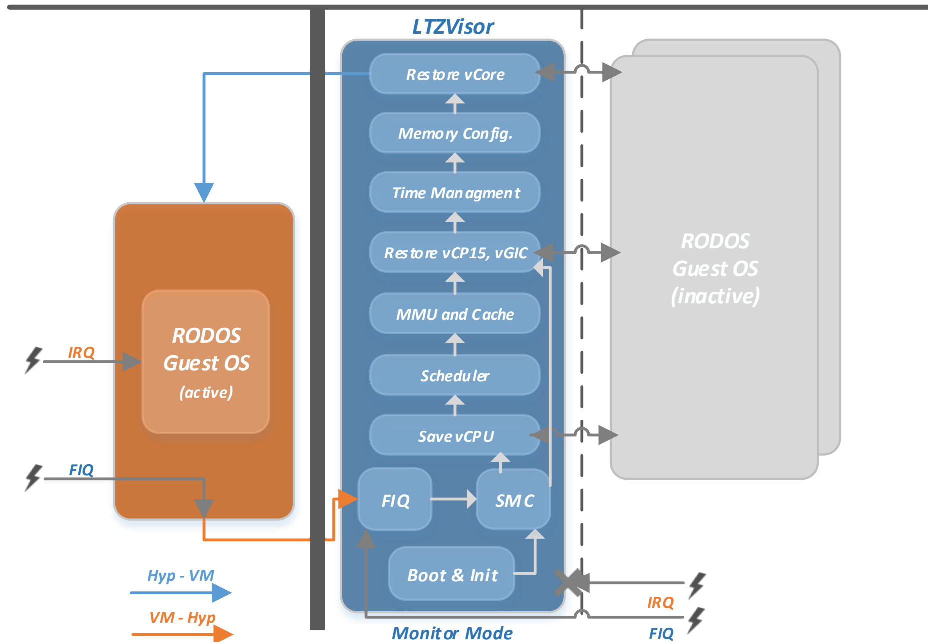


Figure 4.6: TZVisor for Aerospace: execution flow

instant that a FIQ is triggered. The arrival of a FIQ request from the hypervisor timer (tick) brings the processor into monitor mode, jumping to the FIQ handler of the monitor vector table. At this time, the hypervisor will prepare the non-secure world save operation, by first acknowledging and handling the interrupt.

The saving operation is very straightforward: it includes saving the core-, CP15- and GIC-specific registers into the VMCB of the active VM. After saving the state of the active VM, the hypervisor invokes the scheduler to select the next running VM. As explained in Section 4.3.2, the scheduler follows a cyclic policy, which ensures VM selection follows a consecutive order. After selecting the next active VM, the hypervisor performs all MMU and cache maintenance operations and then prepares the restoring process of the new VM. When it happens, the processor performs all previously described steps again. Figure 4.6 summarizes the TZVisor for Aerospace execution flow.

4.4.3 Evaluation

TZVisor for aerospace was evaluated on a Xilinx ZC702 evaluation board for a single-core configuration. The ARM Cortex-A9 was configured to run at 600 MHz. The evaluation focused on the following metrics: memory footprint, partition context switch time and partition performance loss. To evaluate partition context-switch

time and performance loss two different experiments were specified:

1. ***Experiment 1*** - MMU, data and instruction cache as well as branch predictor (BP) support for partitions were disabled;

2. ***Experiment 2*** - MMU, data and instruction cache as well as branch predictor support for partitions were enabled;

In both experiments software stacks were compiled using the ARM Xilinx toolchain with compilation optimizations disabled (except for the experiment which expresses the correlation between guest-switching rate and performance overhead). RODOS, a tickless RTOS already in use in several satellites, was used as guest OS.

Memory Footprint

To access memory footprint results, the size tool of ARM Xilinx toolchain was used. Table 4.2 presents the collected measurements, where boot code and drivers were not taken into consideration. As it can be seen, the memory overhead introduced by the hypervisor - and in fact the TCB of the system - is around 6 KB. The main reasons behind this low memory footprint are the hardware support of TrustZone technology altogether with the careful design and static configuration of each TZVisor component.

Table 4.2: TZVisor for Aerospace: memory footprint (bytes)

	.text	***.data***	***.bss***	***Total***
Hypervisor	5568	192	0	**5760**

Partition Context Switch

To evaluate the partition context switch time the PMU component was used. To measure the time consumed by each internal activity of the context-switch operation, breakpoints were added at the beginning and end of each code segment under measurement. Results were gathered in clock cycles and converted to microseconds accordingly to the processor frequency (600MHz). Each value represents an average of ten collected samples.

The list of internal activities to perform a full guest-switch are:

1. ***FIQ handling*** - The non-secure guest OS is running while a secure interrupt

is triggered (hypervisor timer tick). It is given by the time since the processor enters the monitor's vector table until TZVisor acknowledges and handles the FIQ;

2. ***Save vCore context*** - TZVisor saves the core context of the active non-secure guest OS. It is the time to save the core state of the active non-secure guest OS to its respective VMCB;

3. ***Save vCP15 context*** - TZVisor continues saving the context of the active non-secure guest OS, specifically the CP15. It is the time to save the CP15 state of the active non-secure guest OS to its respective VMCB. This operation is not executed if no cache support for non-secure guest OSes is enabled at design time;

4. ***Save vGIC context*** - TZVisor concludes saving the context of the active non-secure guest OS with the GIC. It is the time to save the GIC state of the active non-secure guest OS to its respective VMCB.

5. ***Scheduler*** - TZVisor concludes saving the context of the active non-secure guest OS. The hypervisor scheduler is invoked to select the next guest to run, following a cyclic policy. It is expressed as the time consumed by the hypervisor during the scheduling action;

6. ***MMU and cache maintenance*** - TZVisor selects the next guest OS ready-to-run and then (optionally) performs the MMU and cache related operations. It expresses the time to perform all MMU and cache related maintenance operations. This operation is not executed if no cache support for non-secure guest OSes is enabled at design time;

7. ***Restore vCP15 context*** - TZVisor select the next guest OS to run and then starts it by restoring its CP15 state. It is given by the time to restore the CP15 state of the selected guest OS from its respective VMCB. This operation is not executed if no cache support for non-secure guest OSes is enabled at design time;

8. ***Restore vGIC context*** - TZVisor restores the CP15 of the new guest OS and continues the restoring operation with the GIC-related registers. It is expressed as the time to restore the GIC state of the selected guest OS from its respective VMCB.

9. ***Time management*** - The context of the new guest OS is partially restored

by the hypervisor. The guest OS needs to be aware of the real passage of the time, before continue executing. So, the hypervisor updates the timing structures and devices with the secure timer information. It is given by the time to load the non-secure guest OS timing structures with information from the hypervisor timing structures that keep track of the real passage of the time;

10. ***Memory configuration*** - TZVisor updates the new guest OS with the timing information and then configure the memory of the previous executing guest as secure, and the new guest as non-secure. It is the time to change the memory configuration;

11. ***Restore vCore context*** - TZVisor restores the core context of the new non-secure guest OS. It is expressed as the time to restore the core state of the active non-secure guest OS to its respective VMCB;

The list of activities, as well as the measured time for each test case scenario are presented in Table 4.3. As it can be seen, in the first test case scenario (MMU and caches disabled), the activities which present higher consuming time are the virtual GIC context-switch (save and restore) and the time management. The virtual GIC context-switch operation takes approximately 15.3μs and the time management takes approximately 53μs. Nevertheless, there is a chance to optimize the time management operation if more hardware timers are provided by the target platform. In the second test case scenario (MMU and caches enabled), since MMU and cache support for guest OSes are needed, the major source of overhead (approximately 114μs) is related with the MMU and cache management. On this case, there is also

Table 4.3: TZVisor for Aerospace: performance statistics

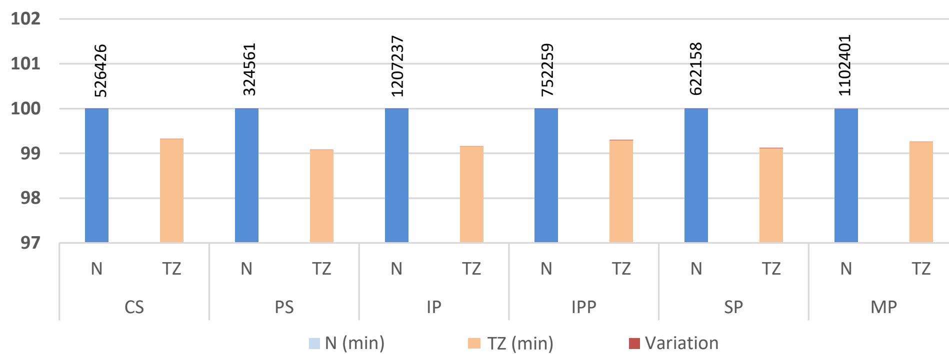
Operation	***Caches disabled***		***Caches enabled***	
	Performance	***Time***	***Performance***	***Time***
	$\overline{x}$	@600MHz	$\overline{x}$	@600MHz
(1) FIQ handling	972	1.620μs	975	1.625μs
(2) Save vCore	1124	1.873μs	1120	1.867μs
(3) Save vCP15	-	-	1441	2.402μs
(4) Save vGIC	3884	6.473μs	3887	6479μs
(5) Scheduler	2400	4.000μs	2402	4.003μs
(6) MMU and cache	-	-	68650	114.417μs
(7) Restore vCP15	-	-	1493	2.488μs
(8) Restore vGIC	5290	8.817μs	5281	8.802μs
(9) Time management	31820	53.033μs	31791	52.985μs
(10) Memory configuration	632	1.053μs	631	1.052μs
(11) Restore vCore	1178	1.963μs	1178	1.963μs
Total	**47299**	**78.832μs**	**118850**	**198.083μs**

a chance to optimize this operation, based on the exploration of the cache locking mechanism provided by the hardware platform (please refer to Section 4.6).

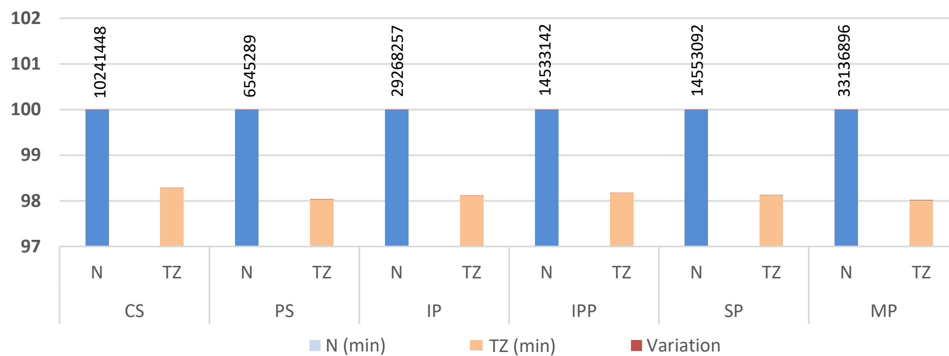
Performance

To access performance results, the Thread-Metric Benchmark Suite was used. Benchmarks were executed in the native version of RODOS (N), and then compared against the virtualized version (TZ). I split the performance evaluation experiment in two parts: the first one evaluates the performance for a specific guest-switching rate (10 milliseconds); and the second one evaluates how the guest-switching rate correlates with the guest performance.

For the first part of the experiment the hypervisor was configured with a 10 milliseconds guest-switching rate. The system was set to run one single guest partition, and the hypervisor scheduler was forced to reschedule the same guest, so that results can translate the full overhead of the complete guest-switching operation. Benchmarks were executed in the native version of RODOS and then compared to the virtualized version. Figure 4.7 presents the achieved results, corresponding to the



(a) Caches disabled

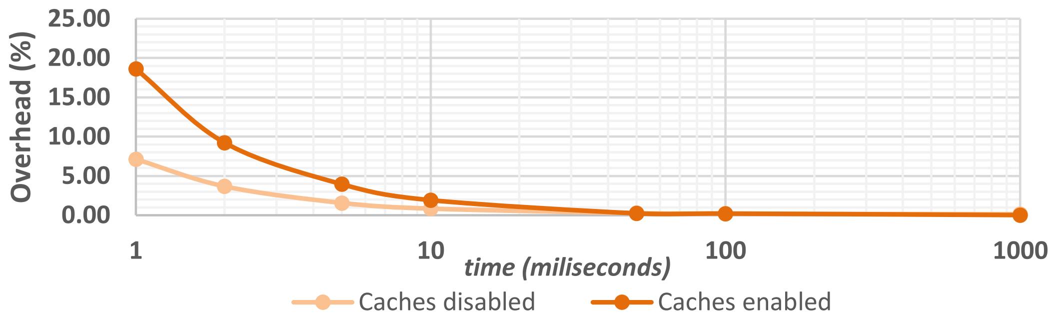


(b) Caches enabled

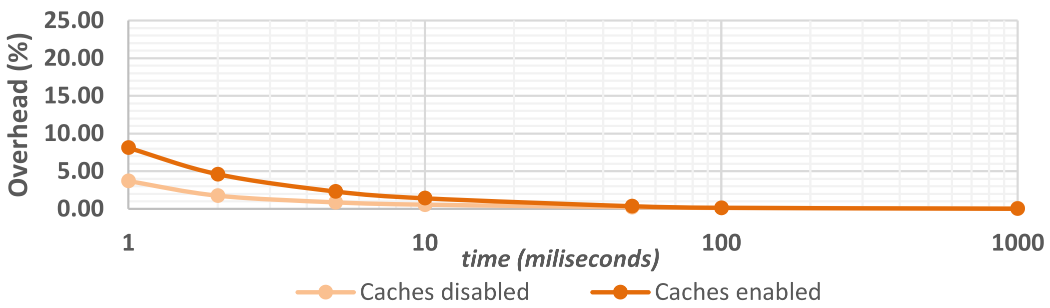
Figure 4.7: TZVisor for Aerospace: Thread-Metric benchmarks

relative performance and variation (as well as the average absolute performance) of 50 collected samples for each benchmark. In both test case scenarios - Figure 4.7a and Figure 4.7b -, it is clear that the virtualized version of RODOS only presents a very small performance overhead when compared to its native execution - <0.9% and <2.0%, respectively -, as well as a variation in the same order of magnitude as the native version. This means the virtualized system remains as deterministic as the native one. In Figure 4.7a (MMU and caches disabled) the performance overhead is smaller, because, as aforementioned, the guest-switching operation does not require the execution of several operations such as cleaning and invalidating data and instruction caches.

The focus of the second part of the experiment was on how the guest-switching rate correlates to guest performance. Instead of fixing the guest-switching rate in 10 ms, the same experiments were repeated for a guest-switching rate between 1 to 1000 ms. Figure 4.8 shows achieved results, where each mark corresponds to the average performance overhead of measured results for the 6 benchmarks. Figure 4.8a presents the achieved results with compilation optimizations disabled (-O0). The performance overhead of the virtualized RODOS ranges from 7.13% to 0.13% and 18.61% to 0.02% with caches disabled and enabled, respectively. When caches are enabled the significant rise of overhead above 5 milliseconds is mainly explained by



(a) Compilation optimizations disabled (-O0)



(b) Compilation optimizations enabled (-O2)

Figure 4.8: TZVisor for Aerospace: guest-switching rate vs performance

two reasons: firstly, as previously explained, when MMU and caches are enabled, the list of internal activities of the context switch operation is higher; and secondly, since caches have to be cleaned and invalidated each time a partition is rescheduled, partitions will not take advantage of them until they are filled. Nevertheless, guest-switching rate should be tuned accordingly to the maximum acceptable latency among each guest, otherwise the real-time characteristics of the system can be compromised.

Figure 4.8a depicts the correlation between the guest-switching rate and performance overhead for the typical compilation optimizations configuration that were presented in all previous experiments (optimizations disabled, -O0). Figure 4.8b, on the other hand, depicts the assessed results for a system configuration with compilation optimizations enabled. The reason behind this decision is related to a comment of a reviewer during the evaluation process of my journal paper entitled "*Towards a TrustZone-assisted Hypervisor for Real Time Embedded Systems*". The reviewer said that the experiments with compilation optimizations disabled were not valid, justified by the fact the use of TrustZone hardware shrunk the overheads when compared to the native OS execution:

- "*The evaluation is flawed to the point that it is impossible to draw conclusions from it, for a very simple reason: all tests are performed using code without compiler optimizations enabled. That means that the impact of hardware overheads (and the overheads of the techniques described in this paper) are shrunk relative to the longer run-time of the straight-line code that doesn't use the author's techniques.*"

In my opinion, I think the comment from the reviewer was inadequate and too strong for two reasons: first, because my goal while presenting all experiments without compilation optimizations disabled was to evaluate the system without the interference of software optimizations; second, because when the code is compiled with optimizations enabled, it means all software stack are affected, inclusively the hypervisor code. This will be obvious in the guest context-switch code, speeding-up this process, which will be translated in a decrease of the performance overhead. In fact, the results presented in Figure 4.8b corroborate my predictions as the performance overhead is smaller, and for low values of guest-switching rate ($<$ 5 milliseconds) the decrease is considerable. The performance overhead of the virtualized RODOS, in this particular case, ranges from 3.69% to 0.06% and 8.10% to 0.01% with caches disabled and enabled, respectively.

4.5 Industrial Mixed-Critical Use Case

With the advent of the Fourth Industrial Revolution, initiatives such as Industry 4.0 or industrial Internet of Things are dramatically changing the way modern automation and industrial control systems (ICS) are conceived and designed [4.19, 4.20]. The industry is embracing an unprecedented technological trend for connecting billions of devices, while converging multiple discrete systems into a single unit [4.21, 4.22]. This strong initiative to connect the unconnected, altogether with a tight system integration, are key-enablers to effectively monitor and optimize complex industrial processes, with important economic advantages due to the capital (CAPEX) and operational expenditure (OPEX) reductions [4.21].

The increasingly need for connectivity and integration raises, however, several safety, reliability and security concerns [4.22, 4.23]. While in the context of industrial control systems the notion of security has traditionally almost the same meaning as safety (that is, protection of human lives and machines against system failures), with integration of information technology, industrial controllers need to guarantee functionality isolation and real-time behavior, while protecting their integrity against unauthorized modification and restricting access to production-related data (company know-how) [4.22, 4.23]. Stuxnet cyber-attack [4.24] clearly demonstrated there is an urgent need to ensure that software components with different criticality do not interfere each other, and that these systems work reliably and robust as specified.

Isolation is a well-established strategy for achieving separation of concerns between functionally independent software components. Software-based approaches such as microkernels and virtualization have been used as an enabler for safety and security in several domains [4.15, 4.25, 4.4], but it is proven these methods, when supported only by software, fail in providing the desired security level [4.26]. In the industrial automation context, adoption of virtualization technology has been limited due to unavailability of mature solutions, as well as the imposed strict timing requirements of control systems [4.27]. However, the advances in hardware, as well as in software, opens up for new virtualization architectures.

This Section presents TZVisor for ICS, a specific configuration of TZVisor general architecture to meet the mixed-critical requirements of the modern automation and industrial control systems. It is demonstrated the effectiveness of the presented solution by running an RTOS side-by-side with two GPOS instances on the Xilinx ZC702

board. The evaluation process is conducted to access the performance overhead in running multiple guest OSes in the non-secure world side.

4.5.1 Implementation

Figure 4.9 presents the architecture of TZVisor for industrial control systems. This configuration follows the generic TZVisor architecture as the secure VM is part of the overall system. The secure VM is essential to guarantee the strong timing requirements of real-time control applications. The hypervisor scheduler implements the two-layered approach: the first layer ensures the asymmetric design principle and guarantees the RTOS has the processor as long as it needs; and when the secure VM is idle the second layer ensures a fair temporal isolation between the multiple non-secure guest OSes. Spatial isolation, as aforementioned, is implemented through the dynamic re-configuration of the memory security state. The hypervisor and the secure VM use the same memory segment, configured as secure. The state of this memory segment is preserved (soft-coded) during runtime. On the other hand, non-secure VMs have their memory segments configured as secure or non-secure, depending if the VM is active or not. Caches, TLBs and branch predictor interfaces of the non-secure side are clean and invalidated at every guest switch. This is a mandatory procedure on this specific configuration, due to the use of GPOSes that requires the use of a virtual space.

Devices assigned to the secure VM are configured as secure and their state will never change. To achieve device isolation between the non-secure VMs, they are dynamically configured as non-secure or secure, depending on partition state (active

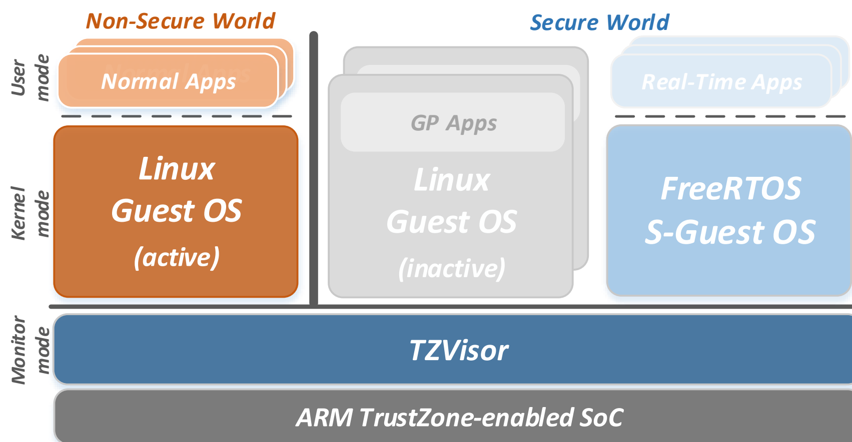


Figure 4.9: TZVisor for ICS: system architecture

or inactive). For interrupts, secure interrupts are directly handled by the secure VM, if it is executing, or redirected to the hypervisor. Non-secure interrupts are redirected to the active guest (without hypervisor interference). When a partition is under execution, only the interrupts managed by this partition are enabled. Interrupts of inactive partitions are momentaneously configured as secure, but disabled.

Time is also managed following a two-layered approach, but with slightly differences when compared to the previous use case. The hypervisor manages two timers: one 32-bit timer unit (TTC0) for the hypervisor tick, and another 32-bit timer (TTC0) to keep coherency over guest OSes absolute time. The timer dedicated to the hypervisor, i.e. TTC0 is configured as secure devices and will have an higher privilege of execution than timers dedicated to the active guest. This means that once an FIQ triggered by the hypervisor tick timer arises, the hypervisor takes control of the system despite of what is happening in the active guest. The partition (Linux in this specific case) manages two timers: one 32-bit timer unit (TTC1) for managing absolute time (real time clock) and another 32-bit timer unit (TTC1) for managing relative time (system timer). As a complement to the hardware timers, Linux implement also Jiffies as global variables that hold the number of ticks that have occurred since the system booted. This data helps to keep track of absolute time. When the active guest is executing, timers belonging to the guest are directly managed and updated by the guest OS. For inactive guests the hypervisor implements a timekeeping mechanism based on a time-base unit that measures the absolute time. When a guest is rescheduled, its internal clocks and related data structures are updated with the time elapsed since its previous execution.

4.5.2 Execution Flow

The system starts with the secure world boot process. This procedure is responsible for a set of operations which includes specific processor and coprocessor registers initialization, vector tables setting, stacks configuration, and MMU, cache and branch predictor initialization (disable). Once the secure world boot process is finished, the hypervisor starts executing.

The hypervisor is then responsible for a set of initialization of platform-specific hardware. This process includes configuring memory, devices and interrupts: all memory segments are configured as secure; all devices are configured as secure, except the TTC1; the GIC is initialized and configured so that secure interrupts use the FIQ interrupt mechanism of the processor, and all interrupt sources are

configured as secure (except the TTC1). Then, the hypervisor is responsible for the initialization of some internal data structures, and creates and loads the respective VM(s) and corresponding binary images. Once the hypervisor is initialized and the VM(s) created, control is passed to the RTOS.

After the complete hypervisor initialization, the RTOS is booted and starts scheduling its own tasks. When the ready-to-run task list becomes empty, the idle task performs a system call to explicitly invoke the hypervisor. Immediately, the processor enters the monitor mode, saves the secure guest OS context, and then goes through the scheduler.

The second layer of hypervisor scheduler is responsible for selecting the next running non-secure VM. As previously explained, the scheduler follows a cyclic policy for managing non-secure guest OSes. After selecting the next active VM, the hypervisor performs all MMU and cache maintenance operations. This maintenance operations are followed by the VM restoring process. This step encompasses restoring CP15- and GIC-specific registers, as well as configuring the security state of the memory, and updating the guest-specific timing structures. The last part of the restore operation ensures the core registers are correctly loaded from a VMCB. At

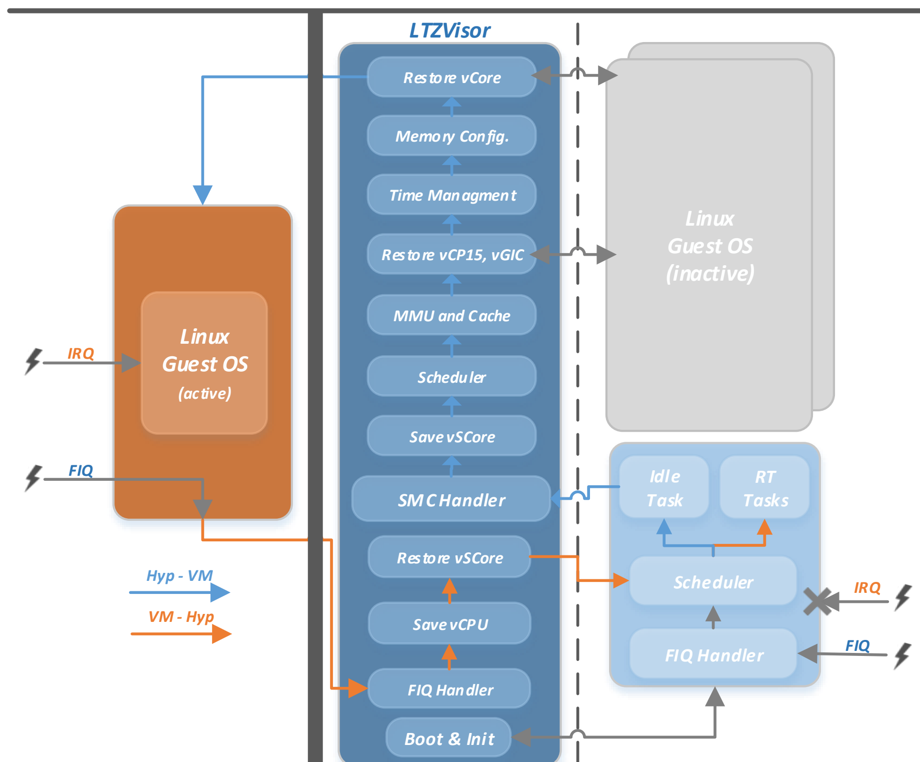


Figure 4.10: TZVisor for ICS: execution flow

the end, the hypervisor enables the NS bit and jumps to the non-secure world side. Once the non-secure guest OS starts executing, it will run until the instant that a FIQ is triggered. The arrival of a FIQ request brings the processor into monitor mode, jumping to the FIQ handler of the monitor vector table. At this time, the hypervisor will prepare the non-secure world save operation.

The saving operation is very straightforward: it includes saving the core-, CP15- and GIC-specific registers into the VMCB of the active VM. After saving the state of the active VM, the hypervisor restores the context of the secure guest OS, and the RTOS starts executing. When it happens, the processor performs all previously described steps again. Figure 4.10 summarizes the TZVisor for ICS execution flow.

4.5.3 Evaluation

TZVisor for ICS was evaluated on a Xilinx ZC702 evaluation board targeting a dual ARM Cortex-A9 running at 667MHz. The performed evaluation focused on three metrics: memory footprint, and performance overhead. TZVisor and both OS partitions were compiled using the ARM Xilinx toolchain, with compilation optimizations disabled (-O0). Linaro Linux (v3.3.0) and FreeRTOS (v7.0.2) were used as non-secure and secure partitions, respectively. MMU, data and instruction cache and branch predictor were disabled on the secure world side.

Memory Footprint

To access memory footprint results, the size tool of ARM Xilinx toolchain was used. Table 4.4 presents the collected measurements, where boot code and drivers were not taken into consideration. As it can be seen, the memory overhead introduced by the hypervisor is around 6.5 KB.

Table 4.4: TZVisor for ICS: memory footprint (bytes)

	.text	***.data***	***.bss***	***Total***
Hypervisor	4612	1092	608	**6312**

Partition context switching

To evaluate the guest context switch time I used the PMU component. To measure the time consumed by each internal activity of a round-trip world switch, a PMU-

specific instruction was added at the beginning and end of each code portion to be measured. Results were gathered in clock cycles and converted to microseconds accordingly to the processor's frequency (667MHz). Each value represents the average and the standard deviation of ten collected samples.

The list of internal activities to perform a full switch between secure to non-secure and non-secure to secure worlds are:

1. ***SMC handling*** - The secure guest OS schedules the idle task. The idle task performs a secure call that is responsible for invoking the hypervisor (*SMC*). It is given by the time since the processor enters in the monitor's vector table until TZVisor completes the SMC handling;

2. ***Save vSCore context*** - TZVisor handles the SMC request and saves the context of the secure guest OS. It is the time to save the current state of the secure guest OS to its respective VMCB;

3. ***Cyclic Scheduler*** - TZVisor concludes to save the context of the secure guest OS. The hypervisor scheduler is invoked to select the next non-secure VM to run. It is the time spent by the hypervisor during the scheduling action;

4. ***MMU and cache maintenance*** - TZVisor selects the next guest OS ready-to-run and then performs the MMU and cache related operations. It is the time to perform all MMU and cache related maintenance operations;

5. ***Restore vCP15 and vGIC context*** - TZVisor performs the MMU and cache related maintenance operations, and then restores partially the context of the new non-secure VM. It is given by the time to restore the CP15 and GIC state of the selected guest OS from its respective VMCB;

6. ***Time management*** - The context of the new guest OS is partially restored by the hypervisor. The guest OS need to be aware of the real passage of the time, before continue executing. It is given by the time to update the timer units;

7. ***Memory configuration*** - TZVisor updates the new guest OS with the timing information and then configure the memory of the previous executing guest as secure, and the new guest as non-secure. It is the time to change the security state configuration of the memory;

8. ***Restore vCore context*** - TZVisor restores the core context of the new non-

secure guest OS. It is the time to restore the core state of the active non-secure guest OS from its respective VMCB;

9. ***FIQ handling*** - The non-secure guest OS is running while a secure interrupt is triggered (e.g., hypervisor tick). It is given by the time since the processor enters in the monitor's vector table until TZVisor completes the FIQ handling;

10. ***Save vCPU context*** - TZVisor saves the CPU (core, CP15 and GIC) context of the active non-secure guest OS. It is the time to save the CPU state of the active non-secure guest OS to its respective VMCB;

11. ***Restore vSCore context*** - TZVisor saves the context of the non-secure guest OS and then restores the context of the secure guest OS. It is the time to restore the state of the secure guest OS from its respective VMCB.

12. ***Asymmetric Scheduler*** - TZVisor restores the execution of the RTOS. The RTOS continues the execution of the idle task, and verifies if there are real-time tasks to run. If not, the idle task performs a system call (*SMC*) that is responsible for invoking the hypervisor. It is given by the time since the processor restores the idle task execution until it enters in the monitor's vector table.

The list of activities as well as the corresponding measured time are presented in Table 4.5. A quick look over presented results clearly demonstrate the major source of overhead (approximately 374μs) is related with the MMU and cache management. This value is even more expressive than assessed results for Aerospace use case, mainly because the enabling of L2 cache. Nevertheless, I believe optimizations can be achieved by exploring the cache locking mechanism, which allows system software to lock certain cache ways. An in-depth study will be addressed in the future.

Performance

To evaluate the non-secure guest OS overhead, the LMBench3 suite were used. Two specific benchmarks of the LMBench3 suite (*lat_ops* and *bw_mem*) were executed in the native version of Linux (N), and then compared against the virtualized version (TZ). The performance evaluation experiment was split into two parts: the first one evaluates the performance for a specific guest-switching rate (10 milliseconds); and the second one evaluates how the guest-switching rate correlates to the guest performance. No real-time tasks were added to the system, i.e. the RTOS will

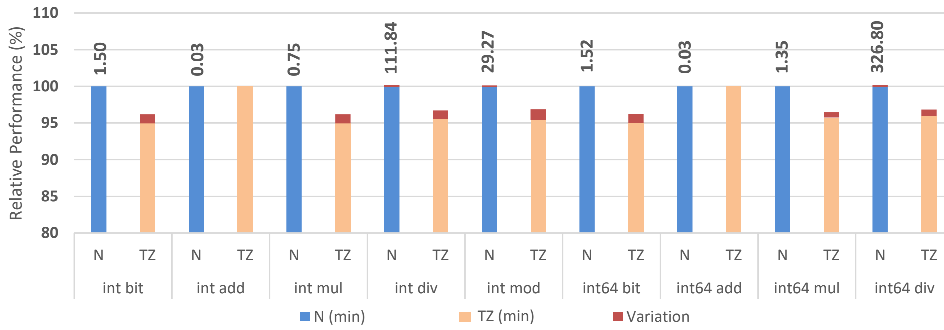
Table 4.5: TZVisor for ICS: performance statistics

Operation	***Performance*** $\overline{x}$	***Time*** @667MHz
(1) SMC handling	575	0.862μs
(2) Save vSCore	420	0.630μs
(3) Cyclic Scheduler	2390	3.583μs
(4) MMU and cache	249324	373.799μs
(5) Restore vCP15 and vGIC	6797	10.190μs
(6) Time management	11820	17.721μs
(7) Memory configuration	632	0.948μs
(8) Restore vCore	1181	1.771μs
(9) FIQ handling	975	1.462μs
(10) Save vCPU	6602	9.898μs
(11) Restore vSCore	241	0.361μs
(12) Assymetric Scheduler	7632	11.442μs
Total	**288589**	**432.667μs**

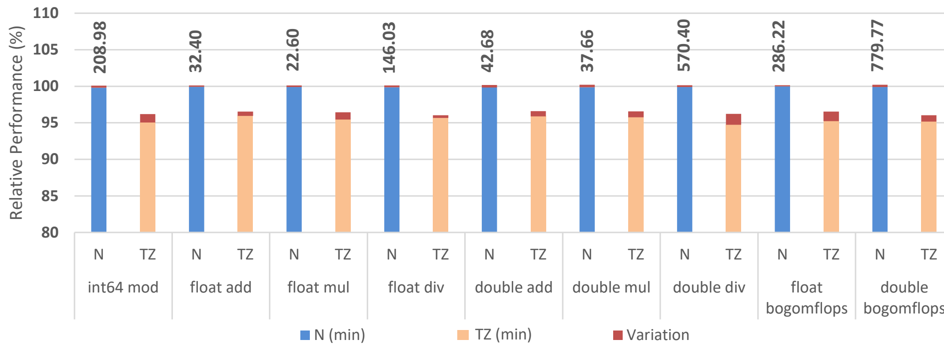
be always running the idle task. MMU, L1 and L2 caches and branch predictor were enabled for both test case scenarios. For each micro-benchmark 10 consecutive experiments were performed. For each experiment the micro-benchmark was configured for 10 warm-ups and 100 repetitions (-W 10 -N 100). Presented results correspond to the average relative performance and variation (as well as the average absolute performance) of the 10 consecutive experiments, encompassing a total of 1000 samples.

For the first part of the experiment the hypervisor was configured with a 10 milliseconds guest-switching rate. The system was set to run one single guest non-secure partition, and the hypervisor scheduler was forced to reschedule the same guest, so that results can translate the full overhead of the complete guest-switching operation. Figure 4.11 presents the achieved results for the arithmetic operations latency benchmark. The values on top of the bars correspond to the average latency, in nanoseconds. The virtualized version of Linux only presents an average performance degradation of 3%, when compared with its native execution. This value is nearly uniform among all micro-benchmarks, except for the `int add` and `int64 add` cases. For this specific micro-benchmarks, the achieved results do not reflect the real performance penalty, due to the lack of precision. Regarding variation, the virtualized Linux presents a slight increase when comparing to the native one. I believe this effect is the result of caches are being flushed in every guest-switch, due to the nonlinearities they introduce in the system.

Figure 4.12 presents the achieved results for the memory bandwidth benchmark. The values on top of the bars correspond to the average memory bandwidth, in



(a) *lat_ops* benchmark results (part 1)

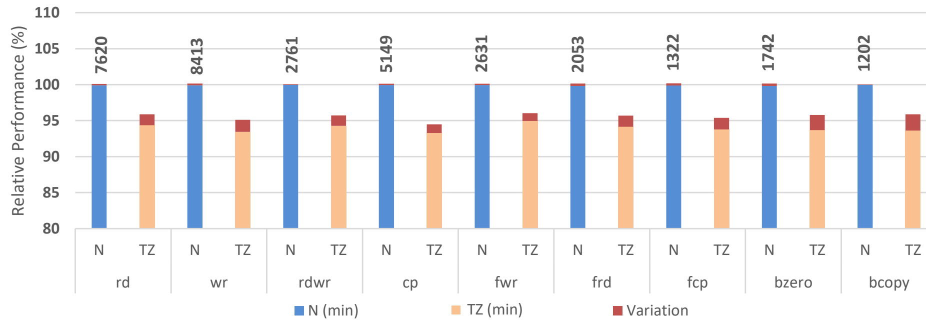


(b) *lat_ops* benchmark results (part 2)

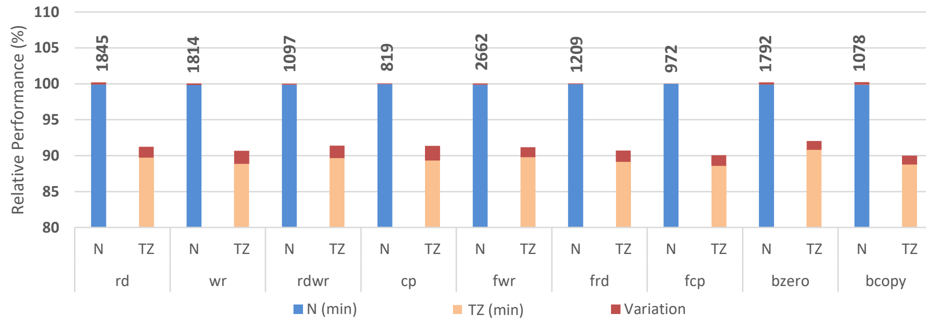
Figure 4.11: TZVisor for ICS: arithmetic operations latency benchmark

megabytes per second (MB/s). Figure 4.12a, Figure 4.12b and Figure 4.12c depict the assessed results for a memory block size of 2KB, 128KB and 4MB, respectively. These memory block sizes, as already explained in the previous Chapter, were selected aiming to fit and not fit within L1 and L2 cache sizes. Looking at the three figures, it is clear the relative performance of the system is not uniform among all scenarios, presenting a slight increase in the performance degradation for the case the memory block size is 128KB. Contrasting these values with the results presented in Figure 4.11, two main conclusions can be drawn: first, it is clearly noticed the effect of cache flushing in memory-intensive workloads, which directly translates, on average, in a slight decrease of performance when comparing, for example, with computing-intensive workloads; second, the bigger the cache size, the bigger the performance penalty, because partitions will not take advantage of caches until they are filled. I also performed a larger set of experiments encompassing memory block sizes of 16KB, 64KB and 1MB. The achieved average relative performance results were identical to the ones presented in Figure 4.12, which reinforces the reliability of described conclusions.

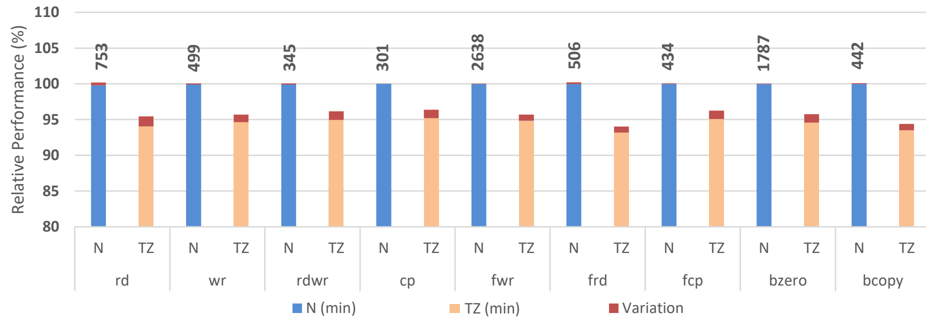
The focus of the second part of the experiment was on how the guest-switching rate correlates to guest performance. Instead of fixing the guest-switching rate in 10 mil-



(a) *bw_mem* benchmark results (2KB)



(b) *bw_mem* benchmark results (128KB)



(c) *bw_mem* benchmark results (4MB)

Figure 4.12: TZVisor for ICS: memory bandwidth benchmark

liseconds, the same experiments were repeated for a guest-switching rate between 1 to 100 milliseconds. Figure 4.13 shows achieved results, where each mark corresponds to the average performance overhead for the arithmetic operations latency benchmark. The performance overhead of the virtualized Linux ranges from 45.50% to 0.50% when the guest-switching rate ranges from 1 to 100 milliseconds, respectively. The performance overhead increases significantly when the guest-switching rate decreases below 5 milliseconds. This is the direct consequence of flushing a significant size of L2-cache for a short period of guest execution time.

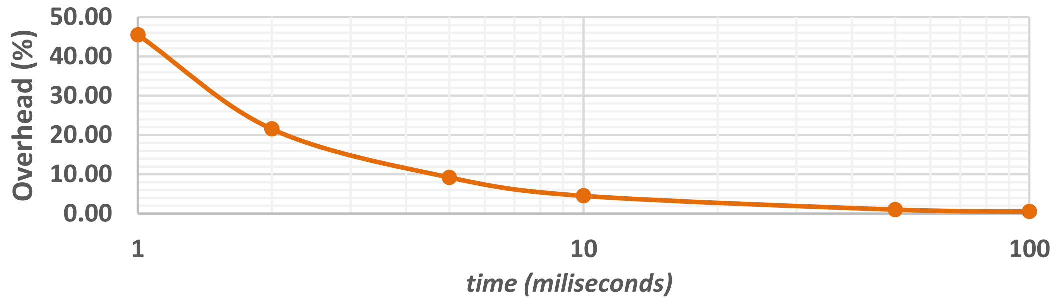


Figure 4.13: TZVisor for ICS: guest-switching rate vs performance

4.6 Discussion

With TZVisor I demonstrated how hardware enhancements introduced by TrustZone technology can be exploited to implement a novel fully-featured virtualization solution that supports the execution of an arbitrary number of guest OSes. I explained how is it possible to multiplex more than one guest OS inside the non-secure world side, by adequately handling shared hardware resources such as processor registers, memory, MMU and caches, and devices and interrupts.

The main reason for researchers considering TrustZone as an ill-guided virtualization mechanism was completely refuted. The limitation of supporting just two VM was abolished. TZVisor demonstrated the non-secure world side can afford as much guest OSes as the ones that can be isolated in different memory segments. This means the number of supported VMs is just limited by the number of memory segments available in the hardware platform, as well as the granularity which the TZASC offers to configure those segments. The TZASC available on the target platform should also offer the dynamic memory configuration feature, in order to allow the reconfiguration of the security state of memory during runtime. It is true that some outdated TrustZone-based SoCs are not equipped with such kind of support, but it is also true that regarding the most modern TrustZone-based SoCs, this is completely different, because they are totally equipped with fully featured TrustZone-aware memory controllers. For example, Sun et al. [4.28] demonstrates the use of the same feature to create TrustICE, a framework that uses the hardware-assisted Watermark feature, available on Freescale i.MX53 QSB, to dynamically protect the memory regions of the suspended secure code (ICEs).

MMU and cache management was identified as the major bottleneck of the developed solution, due to the high performance overhead it introduces in every guest switch. This effect is clearly observed when comparing the results assessed by the

Aerospace and ICS use cases. For Aerospace the L2-Cache was disabled while for ICS it was enabled. The effect of flushing 512KB of memory directly translates into a considerably increase in performance overhead, specially when the guest switching rate reaches just a few milliseconds. What I think would be really interesting was to investigate a correlation between the cache size, guest-switching rate and performance overhead. Depending on the nature of the guest OS, the type of workload, and the frequency caches are flushed, maybe it would be worth to have small memory caches and take less time flushing them during the guest context-switch. I am aware that several ARM processors of ARMv7 family, including the Cortex-A8 and Cortex-A9 (L2Cpl310), offer a coarse-grained cache control that allows system software to lock certain cache ways. The feasibility of such method was already demonstrated by Zhang et al. [4.29], but within a different scope and with a different purpose. Nevertheless, I strongly believe this feature could be used to dedicate some portions/ways of cache to specific guest OSes, and thus eliminating the need of flushing them. During a guest switch operation, instead of cleaning and invalidating the cache, the cache management operation will be just resumed to a simple reconfiguration of the cache lockdown registers [4.30]. It would be interesting to find a pattern for the optimal cache configuration under a specific set of conditions. This way, it could be taken into consideration at the system design time, in order to tune the system to achieve the best performance for a specific application.

Regarding device virtualization, TZVisor introduced a particular paravirtualization approach for managing non-secure guest OS access to devices. In spite of supporting a new mechanism for handling non-secure devices in a more secure way, shared device access is not yet supported. Existent TrustZone-based architectures that implement such support follow essentially three different approaches: emulation, paravirtualization, and re-partitioning. Device emulation follows the classical Popek and Goldberg's trap-and-emulated approach. GPOS accesses to the virtual device are trapped by the hypervisor, which is responsible for handling the physical device. This method brings platform independence and flexibility, with an expense in the TCB size and execution overhead. The paravirtualization approach, as already explained, consists in the slight modification of the GPOS driver (i.e., front-end driver) to sending requests to the hypervisor (back-end). This method presents less execution overhead than emulation, but still requires a considerable engineering-effort and presents limitations in the number of functionalities. The re-partitioning method [4.31], implemented in SafeG, consists of modifying the assignment of devices to each OS during runtime. Implemented in a *pure* and *hybrid* form, the main difference among them is a trade-off between the higher performance of the pure approach and

the lower device latency of the hybrid mechanism. However, this solution presents a big bottleneck: security. Once the device is assigned to the GPOS, the GPOS has complete access to the device, which mean that if it is compromised, the device can be intentionally manipulated to cause a failure. Among the existent shared device access mechanism, there is no one-size-fits-all solution that fully and simultaneously offers a secure method with high flexibility and performance, low latency, and no TCB size expense. My intention is to go beyond state-of-the-art and introduce the concept of self-secured devices. The idea is to extend the duality concept of TrustZone technology to the hardware devices, and endow each physical device with two different logical interfaces: one for each world. Each OS might access the physical device at native performance, and without being aware of the underlying interface. To achieve an higher level of security, configuration of the device should be done in a secure state. The cost of such an advantageous approach is simple: hardware. My main task will be to verify if is it worth.

Time management of non-secure guest OSes was, definitely, one of the most challenging parts when implementing TZVisor, due to the need for ensuring guest OSes have the real notion of the passage of the time. It would be easier to manage virtual time, since the timer units just needed to be paused and resumed at each guest switch, but when shifting for real-time environments this is completely unacceptable. Deadlines would be easily missed, because the notion the RTOS has about the passage of the time is completely different from the real one. As already pointed, the time management support is, therefore, very guest-specific, and it requires a complete understand about the time management internals at the OS level. The number of timing resources is different from OS to OS, and the logic is completely particular. For example Linux uses two 32-bit hardware timers for managing time, while RODOS needs a 64-bit (either implemented as 32-bit hardware-timer and a 32-bit sotfware-timer) and 32-bit timer. While for Linux the solution was much easier and based on the update of the hardware timer units, for RODOS it was much tricky because apart from updating the hardware-timers, it required also the hypervisor to know the position of the software-timer at the guest OS level, in order to update it with the absolute time. Furthermore, despite the complete success in guaranteeing an effective time management for Linux and RODOS, this was possible because of the tickless nature of both OSes. In fact, current implementation still presents some limitations, because there is no way to guarantee the real notion of time for tick-driven OSes such as FreeRTOS. Nevertheless, this problem is a well-know problem among the virtualization community, which represents timekeeping issues in the virtualized world, and still remains as an open problem. I plan to address

this limitation in the near future, but, by now, I honestly do not have any optimal solution in mind.

4.7 Summary

Virtualization technology starts becoming more and more widespread in the embedded space. The penalties incurred by standard software-based virtualization is pushing research towards hardware-assisted solutions. Among the existing commercial off-the-shelf technologies for secure virtualization, ARM TrustZone is attracting particular attention. However, it is often seen with some scepticism due to the dual-OS limitation of existing state-of-the-art solutions.

In this Chapter I presented a novel TrustZone-assisted virtualization solution that allows the execution of an arbitrary number of guest OSes. I demonstrated how is it possible to multiplex several guests OSes inside the non-secure world side, by adequately handling shared hardware resources. The implemented use case scenarios targeting different embedded industries, which impose different timing and safety requirements, demonstrated the viability and versatility of the proposed solution. The conducted evaluation process proved the possibility of running multiple and different unmodified guest OSes with a low performance overhead and memory footprint cost.

References

[4.1] P. Varanasi and G. Heiser, "Hardware-supported virtualization on ARM," *Proceedings of the Second Asia-Pacific Workshop on Systems*, 2011.

[4.2] G. Labs, "An Exploration of ARM TrustZone Technology."

[4.3] T. Frenzel, A. Lackorzynski, A. W. H., and Härtig, "ARM TrustZone as a Virtualization Technique in Embedded Systems," *Twelfth Real-Time Linux Workshop*, 2010.

[4.4] U. Steinberg and B. Kauer, "NOVA: a microhypervisor-based secure virtualization architecture," *Proceedings of the 5th European conference on Computer systems*, pp. 209–222, 2010.

[4.5] D. Sangorrin, S. Honda, and H. Takada, "Dual operating system architecture for real-time embedded systems," in *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Brussels, Belgium*, pp. 6–15, 2010.

[4.6] A. Tavares, A. Didimo, T. Lobo, P. Cardoso, J. Cabral, and S. Montenegro, "Rodosvisor - An ARINC 653 quasi-compliant hypervisor: CPU, memory and I/O virtualization," *IEEE 17th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2012.

[4.7] R. Russell, "Virtio: Towards a De-facto Standard for Virtual I/O Devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 95–103, July 2008.

[4.8] C. Dall and J. Nieh, "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor," *SIGPLAN Not.*, vol. 49, pp. 333–348, Feb. 2014.

[4.9] S. Patni, J. George, P. Lahoti, and J. Abraham, "A zero-copy fast channel for inter-guest and guest-host communication using VirtIO-serial," in *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, pp. 6–9, Sept 2015.

[4.10] F. Baum and A. Raghuraman, "Making Full use of Emerging ARM-based Heterogeneous Multicore SoCs," in *8th European Congress on Embedded Real Time Software and Systems*, 2016.

[4.11] J. Abella, F. J. Cazorla, E. Quinones, A. Grasset, S. Yehia, P. Bonnot, D. Gizopoulos, R. Mariani, and G. Bernat, "Towards improved survivability in safety-critical systems," in *2011 IEEE 17th International On-Line Testing Symposium*, pp. 240–245, July 2011.

[4.12] RTCS, "DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations," in *Washington DC, USA*, 2005.

[4.13] N. Diniz and J. Rufino, "Arinc 653 in space," in *DASIA 2005 - Data Systems in Aerospace*, vol. 602, 2005.

[4.14] J. Windsor and K. Hjortnaes, "Time and space partitioning in spacecraft avionics," in *2009 Third IEEE International Conference on Space Mission Challenges for Information Technology*, pp. 13–20, July 2009.

[4.15] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," *Proceedings of the 11th Real-Time Linux*

Workshop, 2009.

[4.16] S. H. VanderLeest, “Arinc 653 hypervisor,” in *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, pp. 5–E, IEEE, 2010.

[4.17] H. Joe, H. Jeong, Y. Yoon, H. Kim, S. Han, and H. W. Jin, “Full virtualizing micro hypervisor for spacecraft flight computer,” in *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pp. 6C5–1–6C5–9, Oct 2012.

[4.18] A. Crespo, M. Masmano, J. Coronel, S. Peiró, P. Balbastre, and J. Simo, “Multicore partitioned systems based on hypervisor,” *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 12293–12298, 2014.

[4.19] Z. Bi, L. D. Xu, and C. Wang, “Internet of things for enterprise systems of modern manufacturing,” *IEEE Transactions on Industrial Informatics*, vol. 10, pp. 1537–1546, May 2014.

[4.20] R. Drath and A. Horch, “Industrie 4.0: Hit or hype? [industry forum],” *IEEE Industrial Electronics Magazine*, vol. 8, pp. 56–58, June 2014.

[4.21] C. Perera, C. H. Liu, S. Jayawardena, and M. Chen, “A survey on internet of things from industrial market perspective,” *IEEE Access*, vol. 2, pp. 1660–1679, 2014.

[4.22] L. D. Xu, W. He, and S. Li, “Internet of things in industries: A survey,” *IEEE Transactions on Industrial Informatics*, vol. 10, pp. 2233–2243, Nov 2014.

[4.23] A. R. Sadeghi, C. Wachsmann, and M. Waidner, “Security and privacy challenges in industrial internet of things,” in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2015.

[4.24] R. Langner, “Stuxnet: Dissecting a cyberwarfare weapon,” *IEEE Security Privacy*, vol. 9, pp. 49–51, May 2011.

[4.25] G. Heiser, “Virtualizing embedded systems-why bother?,” *Proceedings of the 48th Design Automation Conference (DAC)*, pp. 901–905, 2011.

[4.26] S. Jin, J. Ahn, J. Seol, S. Cha, J. Huh, and S. Maeng, “H-svm: Hardware-assisted secure virtual machines under a vulnerable hypervisor,” *IEEE Transactions on Computers*, vol. 64, pp. 2833–2846, Oct 2015.

[4.27] N. Mahmud, K. Sandstrom, and A. Vulgarakis, “Evaluating industrial applicability of virtualization on a distributed multicore platform,” in *Proceedings of*

the 2014 IEEE Emerging Technology and Factory Automation (ETFA), pp. 1–8, Sept 2014.

[4.28] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pp. 367–378, IEEE Computer Society, 2015.

[4.29] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou, "Cachekit: Evading memory introspection using cache incoherence," in *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 337–352, March 2016.

[4.30] Xilinx, "Zynq-7000 All Programmable SoC: Technical Reference Manual." UG585 (v1.11), September 2017.

[4.31] D. Sangorrin, S. Honda, and H. Takada, "Reliable device sharing mechanisms for dual-os embedded trusted computing," in *Trust and Trustworthy Computing*, vol. 7344 of *Lecture Notes in Computer Science*, pp. 74–91, Springer Berlin Heidelberg, 2012.

"At the end of the day, the goals are simple: safety and security"

- Jodi Rell

5

T-TZVisor: No Safety without Security

The Internet has changed the way we live, and the Internet of Things is making the Internet even more immersive and pervasive [5.1, 5.2]. The ability to connect, manage, and control a device from anywhere and at any time leads IoT systems to generate, process, and exchange vast amounts of security-critical and privacy-sensitive data, turning them into attractive cyber-attack targets [5.3, 5.4]. Traditional protection mechanisms such as cryptographic algorithms and security protocols have proven inefficient [5.5, 5.6], because security is being misconstrued as the addition of features in a late stage of system development. The strong connectivity of IoT environments requires a holistic, end-to-end security approach, addressing security and privacy risks at all abstraction levels [5.4, 5.5, 5.6].

Security by isolation is a well-established strategy for achieving security goals such as data confidentiality, integrity, and availability (CIA). The problem is the way security is being conceived in the information technology sector cannot be directly shifted to the operational technology context, as in this context the main focus is not on digital information protection, but on how to control processes and change of states in a safe and secure way [5.7]. In the OT context several software-based approaches such as microkernels, sandboxes, and virtualizations have been used

[5.8, 5.9, 5.10], but these methods fail in providing the desired security level. In the IT sector several trusted execution environments have been also proposed [5.11, 5.12, 5.13, 5.14, 5.15], but they do not take into account how processes are controlled and which timing guarantees they need. To achieve effective security, IT and OT can no longer be siloed functions - they must be aligned [5.7].

In this Chapter, I present Trusted TZVisor (T-TZVisor) as a complete TrustZone-assisted virtualization solution which addresses security, without risking real-time and safety. Security starts by assuring a root of trust as the basis for a secure boot process, and continues by establishing a chain of trust which validates, at boot time, all levels of secure software running on the device. Runtime security is guaranteed by the implementation of an enhanced trusted execution environment at the guest OS level, with back-end support from the trusted real-time environment. The conducted evaluation process demonstrates how security is assured while the system's real-time properties remain nearly intact.

This Chapter is organized as follows: Section 5.1 defines the goals with the development of T-TZVisor, and Section 5.2 describes the proposed general architecture. The implementation of the security features in the overall system architecture is described in Section 5.3. The evaluation process, presented in Section 5.4, encompasses three main stages: real-time evaluation, security analysis, and experimental validation. The benefits and limitations of T-TZVisor are discussed in Section 5.5, and, finally, Section 5.6 summarizes the Chapter.

Related Publications

The ideas and results presented in this Chapter have been partly published as:

- **S. Pinto**, T. Gomes, J. Pereira, J. Cabral and A. Tavares, "*IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices*", in IEEE Internet Computing, vol. 21, no. 1, pp. 40-47, Jan.-Feb. 2017.
- **S. Pinto**; D. Oliveira, J. Pereira, J. Cabral and A. Tavares, "*FreeTEE: When real-time and security meet*", *in Proceedings of the 2015 IEEE Conference on Emerging Technologies & Factory Automation*, Luxembourg, 2015.

5.1 T-TZVisor: Objectives

The idea of using TrustZone for implementing a trusted execution environment is not new. In fact, it has been a widely used approach for addressing security at the device level, with some associations, such as GlobalPlatform, providing standardization support. While in the context of the information technology sector this approach has been satisfying the CIA needs, in the recent (Industrial) IoT era, this is not completely true [5.7]. While more connected devices are being integrated in the OT context, risks of a security breach can have serious consequences in the physical environment. OT security and IT security are different, and TEEs, as they are being conceived, do not fully and simultaneously satisfy the control, availability, integrity and confidentiality (CAIC) requirements. For this reason, the overall goal with the development of T-TZVisor is to study, evaluate and understand the feasibility of extending TZVisor to include security without risking the real-time and safety properties of the system. This overall goal, can even be split into several more specific objectives:

- ***Objective 1:*** Evaluate which hardware entities can provide the system root of trust;
- ***Objective 2:*** Investigate how to guarantee a complete chain of trust. Enumerate which secure storage components exist and investigate how they must be used;
- ***Objective 3:*** Investigate how to implement a secure software architecture which scales up security without risking real-time and safety;
- ***Objective 4:*** Experiment the proposed solution to measure the impact on the real-time properties of the system while leveraging the four fundamental elements of CAIC.

5.2 T-TZVisor: General Architecture

A TEE is a secure area ensuring that sensitive data is stored, processed and protected in an isolated and trusted environment. Typically, TrustZone-based TEE solutions embody a small secure kernel responsible for managing secure services, on the secure world side, and a rich OS responsible for managing non-secure client applications, on

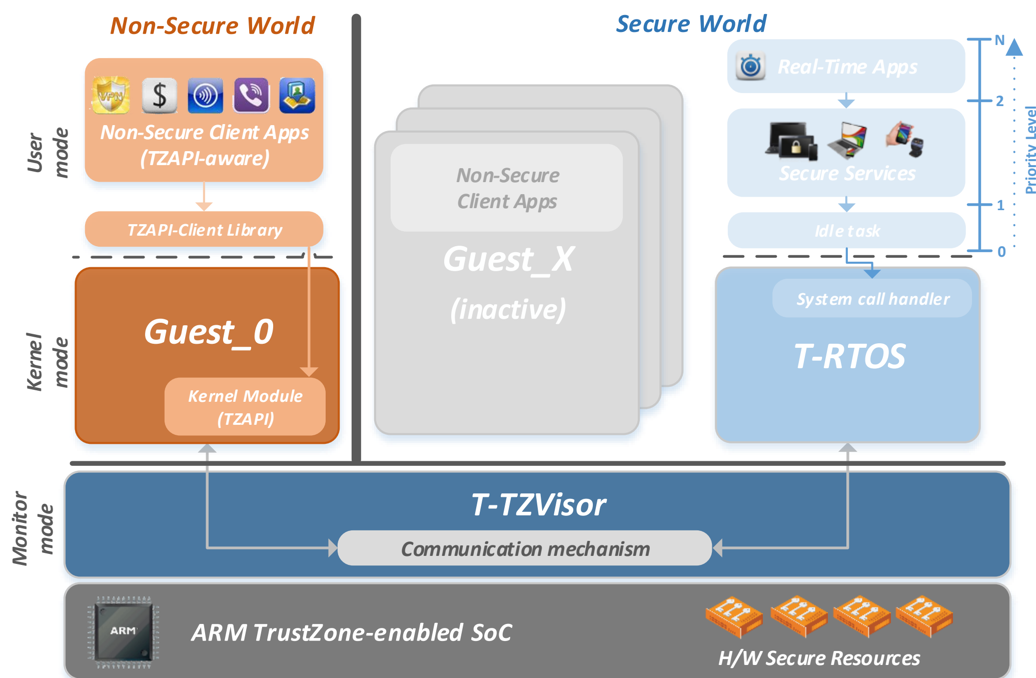


Figure 5.1: T-TZVisor: general architecture

the non-secure world side. The secure OS is only explicitly scheduled under request of the rich environment when an NSCApp needs to access sensitive data. While this approach perfectly fits in several application domains, where real-time is not a concern (e.g., mobile phones), it is not well-suited for a multitude of domains where real-time is a key requirement (e.g., industrial IoT).

Figure 5.1 depicts the proposed T-TZVisor architecture. The secure and safe software architecture relies on TrustZone hardware, as well as other hardware trust anchors to ensure safety is not compromised by a security breach. The proposed system architecture provides a safe and secure environment completely isolated from the rich execution environment, which protects the integrity and confidentiality of secure-sensitive processing while enhancing availability by isolating real-time and critical processing from the non-critical one. As it can be seen, security-related operations as well as the real-time processing are performed on the secure world side, while the general purpose and rich environment are provided by the guest OSes that run, in a round-robin fashion, in the non-secure world side.

The software running in the secure world consists of the T-TZVisor, the Trusted RTOS (T-RTOS) and its corresponding real-time tasks and secure services. T-TZVisor runs in the highest privileged processor mode, i.e., in monitor mode. It is responsible for providing all the support needed for running and managing multiple

guest OSes on the non-secure side, with additional responsibility for providing run-time security by controlling and managing the interaction between the NSCApps of the REE and the secure services of the TEE. The T-RTOS, running in supervisor mode, implements an extended version of an RTOS, providing the basic building blocks of a TEE as a lower-priority thread. In doing so, the T-RTOS allows not only the execution of real-time tasks, but also (low-priority) secure services. The inactive non-secure guest OSes are also preserved in the secure world side.

The software running in the non-secure world side consists of the active guest OS. The OS kernel runs in the supervisor mode, while the applications and libraries execute in user mode. When a guest OS is TrustZone-aware, it also implements additional software modules to guarantee a correct interaction between the NSCApps and the secure services. The TZAPI-dependent software encompasses a privileged TrustZone kernel module, as well as an unprivileged TrustZone API library.

5.3 T-TZVisor: Implementation

T-TZVisor implements a secure TrustZone-assisted virtualization solution that fully and simultaneously addresses security, safety and real-time processing. It starts from the root of trust for a secure boot process, and continues through the complete chain of trust for validating the software components responsible for guaranteeing runtime security, as well as the safe and real-time processing for the entire lifetime. This Section provides all the details behind the implementation, explaining which trusted storage components exist on the Zynq device, describing the complete secure boot process, and explaining the main software extensions at the hypervisor, RTOS and non-secure guest OS level.

5.3.1 Trusted Storage

Ensuring a complete chain of trust is not a trivial task. Firstly, a simple breach in an isolated stage of the complete chain can compromise the integrity of the overall system, and, secondly, some hardware trust anchors are needed.

Secure storage is the basis of hardware support needed for guaranteeing the establishment of a chain of trust. To achieve this, one of the first tasks encompasses the identification of the main (volatile or non-volatile) memory components that

provide such security guarantees. Among the several storage sources, is considered secure storage on-chip memory which is inaccessible to an adversary, i.e. memory that resides within the security perimeter of the SoC device. In Zynq devices, the BootROM, OCM, L1 and L2 cache, AXI block RAM, PL configuration memory, BBRAM, and eFUSE array are hardware components that can be classified as secure storage. Among them, the following storage components need to be highlighted, due to they vital role in the secure boot process:

- The BootROM is 128K mask programmed boot read-only memory, which contains the BootROM code. The BootROM is not visible nor writable. This memory is responsible for preserving the code in charge of performing essential initializations at startup or power on reset, and for copying the FSBL partition from the specified non-volatile memory (NVM) to the OCM memory;

- The OCM is 256K of memory RAM inside the Zynq device. Since the OCM memory has no address or data lines at Zynq device pins, OCM is considered secure storage. The primary function of the OCM is to store the FSBL when the Zynq device is booted. In this case the maximum allowable size of the FSBL is 192K. In addition, the OCM can also be used as secure storage for sensitive software after boot. After boot, the full 256K OCM is available.

- The PL eFUSE array is an on-chip one time programmable NVM used to store the 256-bit AES key. The PS eFUSEs stores the RSA enable bit and the hash of the primary public key (PPK) used in RSA authentication.

- The BBRAM is an on-chip alternative to eFUSE for non-volatile AES key storage. BBRAM is reprogrammable and zeroizable NVM. BBRAM is NVM when an off-chip battery is connected to the Zynq device. The ZC702 board provides the battery while the Zedboard does not.

Secure memory organization

Identifying the possible sources of secure storage is one of the first tasks while trying to ensure a complete chain of trust in a secure system. If the secure storage sources are not adequately established, the chain will be worthless, and, at the end, the system can be compromised.

The BootROM is the root of trust of the system. It is an on-chip read-only memory, which cannot be accessed or even updated, and it is, therefore, critical for the security

of the system. It is responsible for storing a small portion of code responsible for bringing the hardware from the reset into a secure state, and decrypt, authenticate and load the FSBL to the OCM memory.

The OCM memory has been provided to be large enough (256KB) to run the FSBL from an internal location, which is immune to any external probing attack. The OCM is also large enough to securely store TrustZone software routines once the system is booted. This means particular attention should be paid to the memory positions at which T-TZVisor and T-RTOS code is compiled and later loaded. As demonstrated in previous chapters, TZVisor memory footprint is around 6/7KB, while FreeRTOS needs around 20KB of memory. The complete FSBL provided by Xilinx needs about 157KB, including support for all memory interface controllers and code for managing authentication and decryption of the secure system image. The sum of the three pieces of software can perfectly fit into the OCM memory. The application at the RTOS level (which will include real-time tasks and secure services) was not considered, but the OCM still has enough space to store it. Nevertheless, once T-TZVisor is loaded, the space consumed by the FSBL in the OCM memory can be freed.

The types of NVM used to boot Zynq devices are Secure Digital (SD), Quad Serial Peripheral Interface (QSPI) flash, NAND, and NOR. The ZC702 supports only SD and QSPI. If the complete system image fits within the QSPI flash (16MB in ZC702), it is a preferable external boot source than the SD Card, because it is less prone to modification. If the final system image does not fit the QSPI, obviously the SD card should be used. This is the case when a GPOS such as Xilinx Linux is chosen as non-secure guest OS. Nevertheless, since each layer of the final secure system image is encrypted and requires authentication (please refer to Figure 5.2 for more details), even by using the SD card as the main NVM source for booting the system software, the chain of trust is still guaranteed.

In sum, the following considerations should be used when deploying the several layers of software:

- The BootROM, as an on-chip non-volatile read-only memory, should store the small software code responsible for bringing the hardware from the reset in a secure state, and attest and load the FSBL to the OCM memory;
- The OCM memory, as an on-chip volatile memory, should store the FSBL during boot time, as well as the TrustZone critical code (T-TZVisor, T-RTOS, and any other security-critical software component) during runtime;

- The secure system image should be stored preferentially in off-chip non-volatile QSPI flash, but when it does not fit, the SD card should be used;
- The DRAM memory, as an off-chip volatile memory, should only store the non-secure guest OSes as well as other non-security-critical software components. The inactive guest OSes should use the secure memory segments, while the active non-secure guest OS the non-secure ones.

5.3.2 Trusted Boot

A device secure boot involves several stages and encompasses the use of several systems contained within the Zynq SoC device. The secure boot process is always initiated by the BootROM (system root of trust), continues through the OCM with the FSBL and then the T-TZVisor, and ends when non-secure client applications are loaded and running from the external DRAM.

After the power-on and reset sequences have been completed, the on-chip BootROM begins to execute. It starts the whole security chain by ensuring that first-stage boot loader is signed and verified. The BootROM code is a tiny program. It reads the boot mode settings specified by the bootstrap pins, and then reads the boot header, from the specified external non-volatile memory, to determine whether the boot is non-secure or secure. If secure, the key source can be the BBRAM or eFUSE. Table 5.1 presents the existent Boot Header formats.

The next steps in the secure boot process are responsible for the authentication and decryption of the FSBL image. For RSA authentication, the BootROM will use the public key to authenticate the FSBL before it is decrypted or executed. The public key is loaded from the boot image (Figure 5.2 illustrates the secure boot image format) and validated by calculating a SHA-256 signature and then comparing it to the hash value stored in the eFUSE. If both values match, the BootROM calculates the signature for the FSBL and authenticates it with the public key. If the public key signature does not match the hash value stored in eFUSE or the authentication fails on the FSBL, the BootROM enters in a secure lockdown state. In a security

Table 5.1: BootROM header summary

BootROM Header Value	***Description***
0xA5C3C5A3	Encrypted image using eFUSE key.
0x3A5C3C5A	Encrypted image using BBRAM key.
All others	Non-encrypted image.

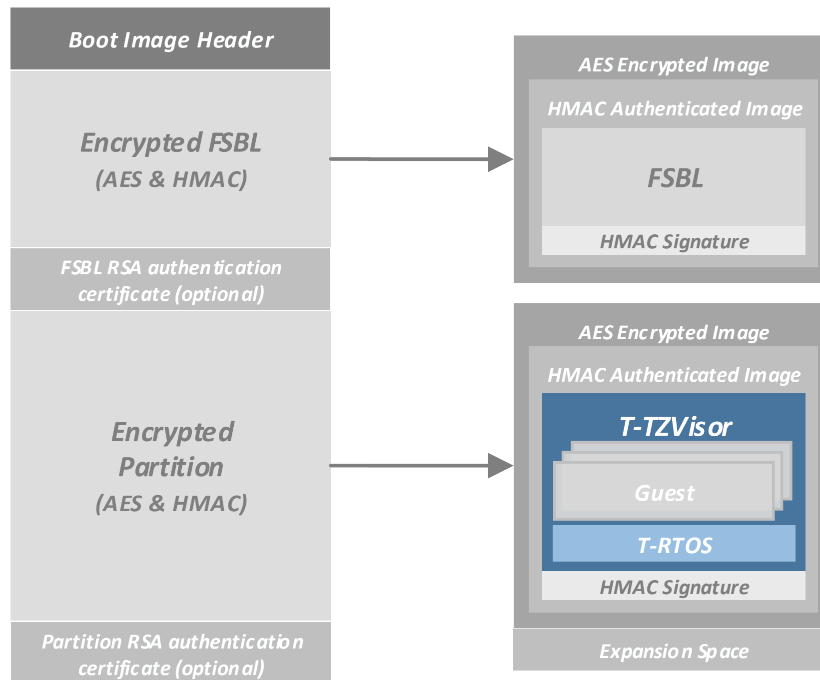


Figure 5.2: T-TZVisor: secure boot image format

lockdown, the on-chip RAM is cleared along with all the system caches; the PL is reset, and the PS enters a lockdown mode that can only be cleared by issuing a power-on reset. If the public key signature matches, the encrypted FSBL is then sent by the BootROM to the AES and hash-based message authentication code (HMAC) hardware. These components are hardened cores within the PL. The FSBL image is decrypted and sent back to the PS, where it is loaded into the OCM for execution. The BootROM also monitors the HMAC authentication status of the FSBL and if an authentication error occurs, the BootROM puts the PS into a secure lockdown state.

Once the FSBL has been successfully loaded and authenticated, control is turned over to the decrypted FSBL code which now resides in the OCM. The FSBL code is then responsible for the authentication, decryption and loading of the T-TZVisor system image. This image contains the critical code of the T-TZVisor and the T-RTOS, as well as the system guest OSes images. The binary images of the guest OSes are individually compiled for the specific memory segment they should run, and then attached to the final system image through the use of specific assembly directives. Initially, they are positioned in consecutive (secure) memory addresses, and, later, the hypervisor is the one responsible for copying the individual guest images for the correct memory segment they should run. Therefore, at this stage, the system image is attested as a whole, and not individually. A similar procedure

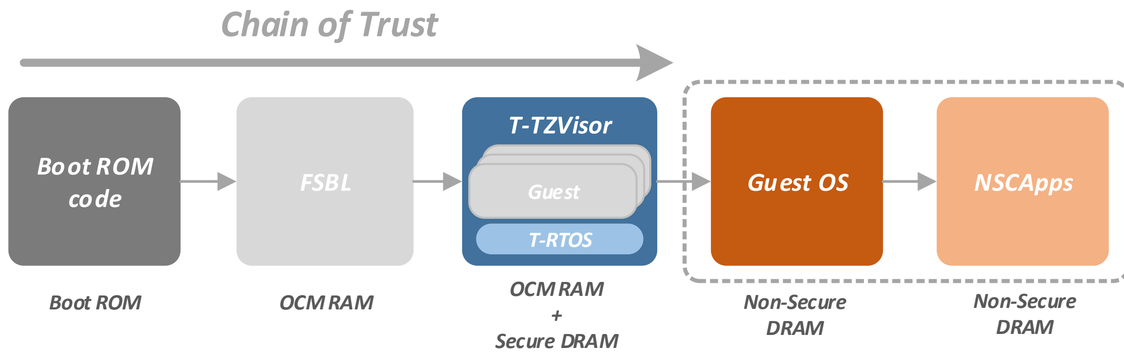


Figure 5.3: T-TZVisor: secure boot process

as the one previously described is then triggered. The FSBL will use the public key to authenticate the T-TZVisor image before it is decrypted or executed. If the authentication is successful, the system image is then sent for decryption, and sent back to the PS, where it is partially loaded into the OCM (T-TZVisor, T-RTOS and other security-critical software), and the remaining code into the external DRAM (guest OSes images and other non-critical software). In the meantime, if the authentication and decryption is not successful, the FSBL puts the PS into a secure lockdown state.

Once the T-TZVisor system image has been successfully loaded and authenticated, control is turned over to the decrypted T-TZVisor, which resides in the OCM. The T-TZVisor is then responsible for configuring specific hardware for the hypervisor, and for loading the guest OS images to the respective memory segment. Guest OSes images are not individually encrypted. As aforementioned, they are part of the overall system image, and once they are loaded from the FSBL, they will reside in the (secure) on-chip or off-chip RAM in a raw format. The memory to which they are loaded depends of their individual image size. If the guest OS is a GPOS, obviously it is loaded to the external DRAM. Assuming that guest OS images are loaded to the external DRAM, there is no more guarantees about the OS integrity. T-TZVisor does not check the integrity of the non-secure guest OSes binaries when they are loaded. This means the chain of trust ends when the critical software is securely running. It is assumed everything that goes outside the perimeter of the secure world side can be compromised, and therefore is outside of the scope of this work. Nevertheless, the addition of another stage of verification, at the guest OS level, will help to achieve a higher level of runtime security for the entire system lifetime. By including an attestation service, it would be possible to check and attest the VMs identity and integrity at boot time, as well as other key components during runtime.

The complete secure boot sequence is summarized and depicted in Figure 5.3. It should be noted that all the support for generating the secure boot image, as well as to program the eFUSE and other hardware components necessary for the secure boot process is guaranteed by the Xilinx Tools. More technical details about the secure boot process on Zynq devices can be found in [5.16, 5.17].

5.3.3 Trusted TZvisor

The Trusted TZvisor is the extended version of TZVisor with facilities for guaranteeing the communication between NSCApps and the secure services. All other facilities provided by TZVisor for managing the different guest OSes are also preserved. It should be clear that the introduced support for the TEE communication is different from the inter-VM communication presented in Section 4.3.8, although both are supported. The extension performed at the hypervisor level encompasses the addition of hypercalls handling mechanisms, for dealing with issued requests that arrive from the non-secure TrustZone kernel driver.

Communication

T-TZVisor implements a remote procedure call (RPC) style communication interface to establish a communication channel between the NSCApps and secure services. RPCs are always initiated in the non-secure world side, where NSCApps use the available system call interfaces (kernel module) to explicitly invoke the SMC instruction. This SMC instruction will trap the execution flow into the monitor mode, where the hypervisor component is responsible for restoring the execution of the T-RTOS, as well as for forwarding the information of the allocated message buffers passed through the core registers. Once the T-RTOS is restored, the idle task is recovered and the secure service dispatcher will forward the incoming request to the respective secure service. The communication follows a blocking-implementation strategy (from a non-secure perspective), which means the non-secure world side will only be recovered on completion of the RPC request. Once it happens, the secure service dispatcher notifies the T-RTOS, which in turn goes through the SMC handler and returns to the last well-known execution point of the non-secure world side.

5.3.4 Trusted RTOS

Typical existent trusted execution environments rely on a secure operating system to provide the facilities to simulate concurrent execution of multiple independent secure services. Developing a secure kernel would require some time, and obviously is not the main focus of this thesis. Furthermore, existent TEEs embody a secure OS that, for the goal it is designed, it completely disregards the timing requirements imposed by a real-time environment. The ultimate goal of this thesis is to engineer a scalable TrustZone-assisted virtualization solution which scales up security without risking safety and real-time. With this objective in mind, I made the decision of using the traditional RTOS, used in TZVisor architecture as the secure guest OS, to provide the foundation for managing the TrustZone API interaction. Consequently, the RTOS needs to be slightly extended in order to implement the TrustZone API back-end support. If adequately implemented, the real-time properties of the system remain the same, the security is integrated into the system, and the engineering effort for implementing such support is low.

Hence, T-FreeRTOS is the modified version of FreeRTOS, which includes the addition of the TEE module. The main services provided by the RTOS (e.g., task management and memory management) were leveraged to manage the secure services, and a small kernel module, to provide support for the TZAPI communication, was implemented. One of the advantages of a design based on operating system principles is the use of the processor MMU to separate the memory space into multiple user space sandboxes. If this feature is implemented, secure services, from independent stakeholders, can execute at the same time without needing to trust each other. The kernel design enforces the logical isolation of secure tasks from each other, preventing one secure task from tampering with the memory space of another. FreeRTOS, natively, does not implement such support. In fact, the implementation of such virtual memory space could be easily done, but, once again, the trade-off for scaling up security without risking real-time restricted such a decision.

The TEE module is responsible for interpreting the commands/data received from the NSCApps and acting accordingly to the desired operation. It acts as a counterpart or a back-end of the normal world kernel module. The TEE module implements a main interface that is responsible for decoding the received commands, and call the specific function developed to handle the request. Several functions were implemented, including for opening and closing sessions with the NSCApps, as well as dispatching the requested secure service. Secure services are hard-coded and cannot

be loaded or removed at runtime. The conditional code of the TEE module for handling the secure services is also static. This design decision reduced the complexity of the developed code, at the cost of flexibility and upgradability. Secure services are managed through the set of APIs provided by the FreeRTOS. Once again, for promoting the real-time behavior of the system, I decided to classify the priority of the secure services as the lowest of the system. This will guarantee and promote higher priority execution for real-time tasks. Kernel modifications were also carefully implemented to first privilege the execution of the real-time features. For example, in conditional switch statements, secure features were checked after real-time related statements, just to not compromise the execution flow.

Listing 5.1: Secure service dispatcher

```
int tz_ss_dispatcher(void *param)
{
   /*{...}*/
   struct tz_smc_cmd *cmd = NULL;
   cmd = (struct tz_smc_cmd*)param;

   switch(cmd->context){
      case ECHO_SS_ID:{
         xTaskCreate( prvEchoSS, (signed char *) "echo_ss",
            configMINIMAL_STACK_SIZE, (void*)param, SS_PRIO,
            &xEchoSS_Handle );
         break;
      }
      /*{...}*/
   }
   /*{...}*/
}
```

Listing 5.1 presents the dispatcher implemented by the TEE module. The dispatcher receives a structure with information about the requested service. The information contains the identification of the requested service. In this specific case, the ECHO service is responsible for retrieving the information sent by the NSCApps, and was implemented just for validation purposes. The conditional switch statement verifies which service matches the one requested, and once identified, a new OS task is created. It should be noted that the priority of the task (`SS_PRIO`) is kept to one (in FreeRTOS a lower value means a lower priority). By now, as it can be seen, the implementation follows a static approach, where the addition of another service will force to add, at least, another case to the conditional switch statement. A

dynamic approach will be implemented in the future, which will open the possibility for downloading secure services during runtime.

5.3.5 TrustZone-aware GPOS

The TrustZone-aware GPOS provides the foundation for application developers to design and implement standard NSCApps that interact with secure services. The GPOS provides a rich and flexible environment by which NSCApps, following the TrustZone API specification (TZAPI library), interact with secure services through the TrustZone kernel module.

TrustZone API Library

Recognizing that developing security software ecosystem has been hindered by the lack of common standards for software development, ARM has released the TrustZone API as a public specification that can be used by anyone as an interface to their underlying security solution.

The TZAPI is a programming interface that enables a NSCApp to access a security environment for managing and using secure services. It enables a client to connect to a service and send commands to the linked service. A command is an abstract message which instructs the service to perform some useful work on behalf of the client. A client can also query installed services and, if the implementation allows it, install new services at run-time.

The TrustZone API specification, as a standardized software API, defines a set of types, constants, data structures, and functions used by NSCApps to interact with the secure services. Table 5.2 presents the main data structures defined by the TrustZone API. The functions are grouped into three main categories. Control

Table 5.2: TrustZone API: main data structures

Name	***Description***
Data Structures	
`tz_device_t`	The structure used to contain control information related to the device.
`tz_session_t`	The structure used to contain control information related to a session between a client and a service.
`tz_operation_t`	The structure is used to contain control information related to an operation that is to be invoked with the security environment.
`tz_shared_memory_t`	The structure is used to contain control information related to a block of shared memory that is mapped between the client and the service.

functions form the main body of the API and deal with the creation of a session between a client and a service, the issuing of commands in that session, and the creation of shared memory mappings. The encoder and decoder functions of the API are used to encode and decode structured messages exchanged between the client and the service. Table 5.3 presents and describes the control functions, and Table 5.4 the encoder and decoder functions specified by the TrustZone API. The Service manager API provides functions that allow a client to enumerate the services installed on the device, to obtain their properties, and (optionally) to download or remove services at run-time. The functions used for installing and removing new services as well as querying the installed services dynamically at run time were not implemented.

In order to access the security environment a client should first open a connection with the underlying device. To achieve this, the client must call the `TZDeviceOpen` function. Then, to interact and use a service, a client must first open a session with it. This action encompasses three main steps: prepare the open operation by calling the function `TZOperationPrepareOpen`; call the function `TZOperationPerform` to connect to the service and send the opening message; and, finally, release the operation

Table 5.3: TrustZone API: control functions

Name	***Description***
Control functions	
`TZDeviceOpen`	This function opens a connection with the device in the underlying operating environment that represents the secure environment.
`TZDeviceClose`	This function closes a connection with a device, freeing any associated resources.
`TZDeviceGetTimeLimit`	This function generates a device-local absolute time limit.
`TZOperationPrepareOpen`	This function is responsible for locally preparing an operation that can be used to connect with the service.
`TZOperationPrepareInvoke`	This function is responsible for locally preparing an operation that can be used to issue a command to a service with which the client has already created a session.
`TZOperationPrepareClose`	This function is responsible for locally preparing an operation that can be used to close a session between the client and a service.
`TZOperationPerform`	This function performs a previously prepared operation - issuing it to the secure environment.
`TZOperationRelease`	This function releases an operation, freeing any associated resources.
`TZOperationCancel`	This function requests the cancellation of an operation in an asynchronous manner.
`TZSharedMemoryAllocate`	This function allocates a block of memory, which is shared between the client and the service it is connected to.
`TZSharedMemoryRegister`	This function registers a block of memory, which is shared between the client and the service it is connected to.
`TZSharedMemoryRelease`	This function marks a block of shared memory associated with a session as no longer shared.

Table 5.4: TrustZone API: encoder and decoder functions

Name	***Description***
Encoder and decoder functions	
`TZEncodeUint32`	This function appends a single data value to the end of the encoded message.
`TZEncodeArray`	This function appends a binary array to the end of the encoded message.
`TZEncodeArraySpace`	This function appends an empty array to the end of the encoded message and returns the pointer to this array to the client.
`TZEncodeMemoryReference`	This function appends a reference to a range of a previously created shared memory block.
`TZDecodeUint32`	This function decodes a single item from the current offset in the structured message returned by the service.
`TZDecodeArraySpace`	This function decodes a block of binary data from the current offset in the structured message returned by the service.
`TZDecodeGetType`	This function returns the type of the data at the current offset in the decoder stream.
`TZDecodeGetError`	This function returns the error state of the decoder associated with the given operation.

context using the function `TZOperationRelease`. Once a client session is opened, the client may then invoke one or more service commands and receive the responses thereof. To invoke a service command, the client must perform the following steps: prepare the invoke operation using the `TZOperationPrepareInvoke` function; call the function `TZOperationPerform` to send the command to the service; and, finally, release the operation context using the function `TZOperationRelease`. Finally, when all commands have been issued, the client must close the session. To close a session, the client must perform the following steps: prepare the close operation using the `TZOperationPrepareClose` function; call the function `TZOperationPerform` to connect to the service and send the close message; and, finally, release the operation context using the function `TZOperationRelease`.

A client can use the TZAPI to encode and decode structured messages exchanged with a service. Structured messages are also a convenient way to develop a robust protocol between the client and a service, enabling an implementation to provide type safety and defensive protection against buffer overflow issues. A specific function is defined for each data type to be encoded; the names of the encoder functions start with the prefix `TZEncode*`. A specific function is defined in the TZAPI for each data type which can be decoded; the names of the decoder functions start with the prefix `TZDecode*`. Table 5.4 presents all the encoding and decoding functions specified by the TrustZone API.

The usage of structured messages may add a significant overhead when transferring large quantities of data between a client and a service. This may be unacceptable

in some use-cases where a minimum data bandwidth is required to achieve data streaming. To overcome this bandwidth problem, the TZAPI provides the capability to designate blocks of memory that are shared between the client and the service and directly accessible to both. The client can explicitly cancel any outstanding operation. Additionally, operations may be given an explicit timeout period when they are prepared - after this time, if the operation has not already completed, it will be automatically canceled.

TrustZone kernel module

The TrustZone kernel module, in its generic concept and despite the specificities of the targeting OS, is a piece of privileged software responsible for interpreting the system calls issued by the NSCApps at user level, and managing the establishment of a communication channel with secure services, at kernel level. The current solution presented in this thesis assumes the implementation of TEE support on the non-secure world side for a Linux guest OS. All technical descriptions and terminology are restricted to Linux systems. The kernel module was not implemented from scratch. Instead, a slight refactoring was done to the loadable kernel module that acts as a TEE driver in the Open Virtualization framework. The refactoring encompassed also the addition of extra type-error checking mechanisms and the fixing of some bugs.

The TrustZone kernel module provides a pseudo-character device that implements a logical communication channel (between the normal world and the secure world) on top of the real communication channel, and provides the functional foundation to implement the normal world TZAPI library. It provides a set of specific IOCTLs that semantically understands parameters, allocates memory buffers, encodes and decodes data, prepares the requests and establishes the communication (through SMC instruction). Among existent IOCTLs, `TZ_IOCTL_SES_OPEN_REQ` and `TZ_IOCTL_ENC_UINT32`, for example, are invoked when the API `TZOperationPerform` for opening a session and the API `TZEncodeUint32` for encoding a message are called, respectively.

5.4 Evaluation

The implemented solution was evaluated on a ZC702 evaluation board targeting a dual ARM Cortex-A9 running at 600MHz. In spite of using a multicore hardware

architecture, current implementation only supports a single-core configuration. I focused the evaluation on real-time (experiment 5.4.1) and security (experiment 5.4.2). To evaluate the real-time properties of the system, the Thread-Metric benchmark was used. To evaluate security, I conduct a discussion around how T-TZVisor has fully and simultaneously achieved control, availability, integrity and confidentiality.

5.4.1 Real-time

In order to measure the impact on real-time properties of the system, in terms of performance and determinism, I compared the modified native version of the FreeRTOS (where interrupts are handled as FIQs) against T-FreeRTOS, using the Thread-Metric Benchmark Suite. I collected 50 samples for each benchmark, corresponding to a total of 700 collected samples for both test case scenarios. MMU, caches, branch predictor and others dynamic architectural features were disabled in the secure world side.

As demonstrated in Figure 5.4, assessed results present a negligible overhead when comparing the native execution of FreeRTOS to the modified one. Regarding determinism, the assessed variance was in the same order of magnitude in both test scenarios. This is perfectly understandable because once T-FreeRTOS starts running real-time tasks, it will never be interrupted by any security-related feature. Furthermore, all introduced kernel modifications were carefully implemented to first privilege the execution of real-time features. As it was previous explained, in conditional switch statements, for example, secure features were processed after real-time related statements, just to not compromise the execution flow.

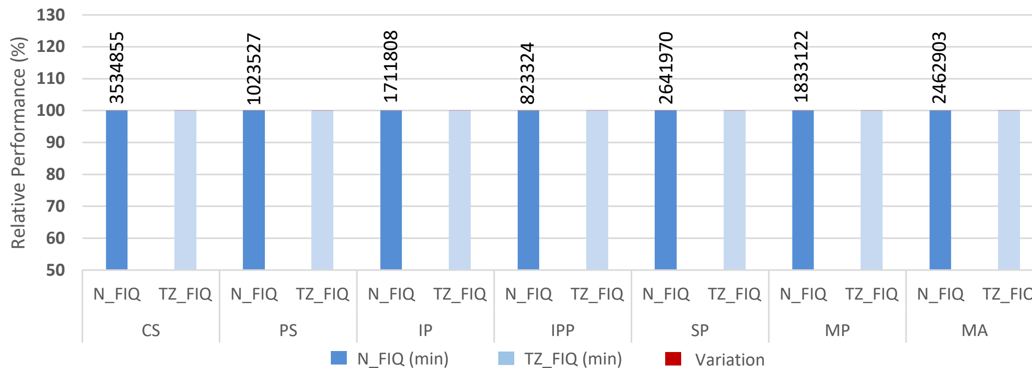


Figure 5.4: T-TZVisor: Thread-Metric benchmark

5.4.2 Security Analysis

In this Section, I analyze the security properties of the developed solution by summarizing which security guarantees are provided by T-TZVisor regarding the four fundamental elements of CAIC. I also describe why the system is not able to provide effective mechanisms against side-channel attacks, and how I plan to address them later.

Security Guarantees

T-TZVisor has fully or partially achieved the four fundamental elements of CAIC:

- ***Control*** refers to the ability to control a process and change a state, when needed, in a safe and secure manner, and without impacting people, safety, and assets. T-TZVisor provides safe and secure control over processes by implementing critical operations as real-time tasks managed by the T-RTOS. The software implemented in the secure world side is considered secure and safe, and cannot be compromised by any external influence from the non-secure guest OSes. Even if a GPOS guest OS is completely compromised, the strong time and spatial isolation ensures critical control processes have a higher privilege of execution and isolation from the non-critical. When implementing an industrial control application requiring mixed criticality decoupled among both worlds, i.e., the graphical user interface for monitoring and control purposes running on the non-secure GPOS, while the control applications running on the T-RTOS, the graphical user interface can issue requests to modify the state of processes but they must be validated at the hypervisor or T-RTOS level;
- ***Availability*** refers to the ability that authorized parties are able to access the information when needed. T-RTOS proved to have a high-level of availability, guaranteed by the strong temporal isolation (asymmetric scheduling policy), as well as by the co-existence of privileged (FIQs) and unprivileged (IRQs) interrupt sources. By scheduling the GPOS only on the idle periods of the T-RTOS, as well as pre-empting its execution once an FIQ is triggered, we were able to guarantee a high-level of availability at the secure world side. Our experiments focused on performing some tests/attacks to the Linux system running on the non-secure side, and observe how they could disturb the correct behavior of T-RTOS. The first experiment consisted in forcing several reboots

to Linux. We have observed that the non-existence of services from the GPOS while rebooting does not affect any kind of service provided by the T-RTOS. Then, we injected a device driver on Linux to re-configure the MMU interface of the non-secure world side to try to access a memory area outside the boundary of the non-secure memory area. Due to the existence of one MMU interface for each world, as well as the strong memory isolation provided by the TZASC, the attempt was completely unsuccessful. At last we have connected a radio transceiver to the system, linked and managed by Linux, that is able to receive data packets from several sensors on a sensor hub. We have tested the behavior of a compromised sensor by repeatedly sending data bursts to our device, which generated repeatedly interrupt requests on Linux. This experiment simulates a DoS attack to the system. Due to the co-existence of privileged (FIQs) and unprivileged (IRQs) interrupt sources, FIQs belonging to the T-RTOS were able to preempt the execution of Linux, even when executing an IRQ request;

- ***Integrity*** enforces the consistency, accuracy, and trustworthiness of data and system over its entire life cycle. T-TZVisor provides integrity only at boot time, through the secure boot process. Once the system is booted, TrustZone, per se, does not provide any hardware or software mechanism to assure the integrity of data over time. A software-based solution for attestation and/or introspection will be implemented, however I believe hardware trust anchors, such as security controllers, will better fit in the Industrial IoT domain. A hybrid approach using TrustZone and security controllers, as envisioned by Winter's research group [5.18], will assure a continuously checking of component authenticity, as well as data and system integrity to prevent manipulation.

- ***Confidentiality*** is the ability to restrict data to those authorized to access it. T-TZVisor partially provides confidentiality by means of TrustZone' strong spatial isolation mechanisms. The GPOS cannot access any memory segment allocated to the T-RTOS, because the TZASC traps any unauthorized memory access. The GPOS with a separated MMU and cache interface nullifies any cached information leakage. The MMU and cache maintenance operations performed during every non-secure guest OS switch also prevents any cached information leakage. Moreover, non-secure guest OSes cannot access any device assigned to the T-RTOS, because the TZPC also traps any unauthorized device access. The only possible access path is through the communication channel, where lies one well-known security breach of TrustZone [5.19]. The current design of TrustZone's architecture does not authenticate

access to resources, enabling man-in-the-middle attacks and so, interception and manipulation of messages transferred through the channel.

Side-Channel Attacks

Side-channel attacks are performed based on observing properties (e.g. timing, and power consumption) of the system, while it performs cryptographic operations. This kind of attacks is out of the security spectrum of TrustZone hardware technology, which, per se, does not provide any mechanism or countermeasure to prevent them. Zynq platform, despite providing some hardware trust anchors that extend the Trust-Zone security spectrum, does not provide any mechanism for dealing with such logical attacks. So, due to the lack of hardware protection, T-TZVisor does not provide a guarantee against side-channel attacks. Nevertheless, T-TZVisor has been designed and deployed in a platform to be compliant with additional hardware accelerators and security modules. The PL of the Zynq device can be perfectly exploited to incorporate randomness into cryptography, as well as use fixed-time algorithms to reduce data-related timing signatures. The benefit of using liquid hardware is the easy upgradability this technology provides.

5.4.3 Experimental Validation

To evaluate the effectiveness of the TEE in providing the security facilities to the NSCApps, a proof of concept application was developed. This proof of concept emulates a basic trusted storage system. The pair of applications is named ECHO_-CLI and ECHO_SS, because it consists of a NSCApp that sends critical data to be securely stored in the secure world, and a secure service that stores the data, and then sends it back to the NSCApp, similar to an echo mechanism. The described experiment assumes the secure boot process was successful, and a complete chain of trust was established to guarantee the secure world software was not compromised.

The execution flow of the interaction of such application can be summarized in three stages. The first one is responsible for establishing the session between the pair of applications. The second one is responsible for performing the operation, i.e., for moving the data from the non-secure to the secure world, store it securely in a trusted region, and then send it back to the non-secure world. Finally, in the last stage the connection is closed.

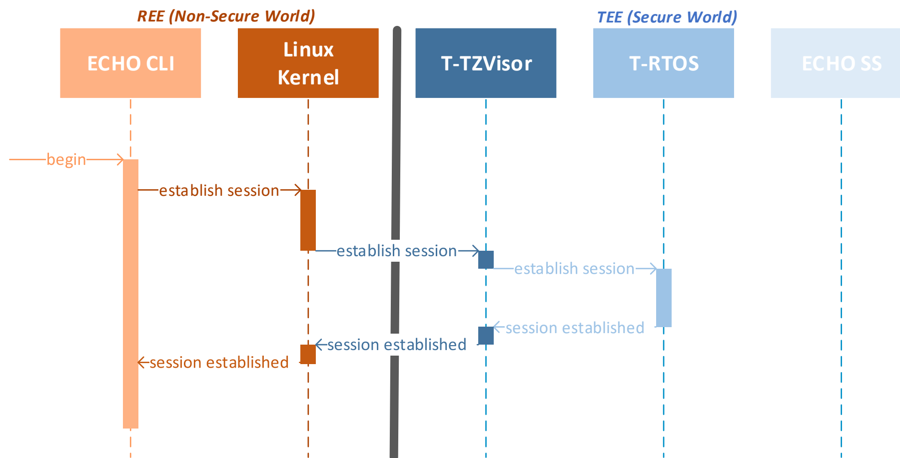


Figure 5.5: T-TZVisor: establishing a secure session

The first stage is then responsible for establishing the session between the pair of applications. The ECHO_CLI NSCApp triggers the communication by providing the necessary information (TZ-API data structure), to establish a session with the secure service. The kernel module is then responsible for keeping this information, and forward the issued request to the T-TZVisor by triggering a specific hypercall (SMC). The hypervisor handles the hypercall, identifying if the issued request corresponds to a valid operation (i.e, in this specific case, a communication request to establish a valid session). If the request is valid, the hypervisor performs a world switch (saving and restoring the context of each world), and sends the request to the main interface of the TEE module running on T-RTOS. The TEE module is then responsible for analyzing the data structure, and verify if the information matches. The session is valid if the context and the universally unique identifier (UUID) of the secure service are valid. Figure 5.5 shows a sequence diagram for establishing a session between the ECHO_CLI application and the TEE module running on the T-RTOS.

The second stage is the one which effectively performs the necessary operation for moving and securing the data from the non-secure side to a trusted memory region. The NSCApp starts by creating a data buffer for sharing data with the secure service. To accomplish this, it requests some facilities provided by the Linux kernel module. The buffer will be created within the non-secure memory area belonging to the Linux non-secure guest OS, and, thus, accessible to both sides. The buffer will be used for sending and receiving data from the ECHO_SS, but within different offsets. This means that, in theory, just one generic data buffer exists, but, in fact, with two different interfaces: one for sending, and another for receiving. As previously explained in Section 5.3.5, the mechanism for sharing data can be a single 32-bit

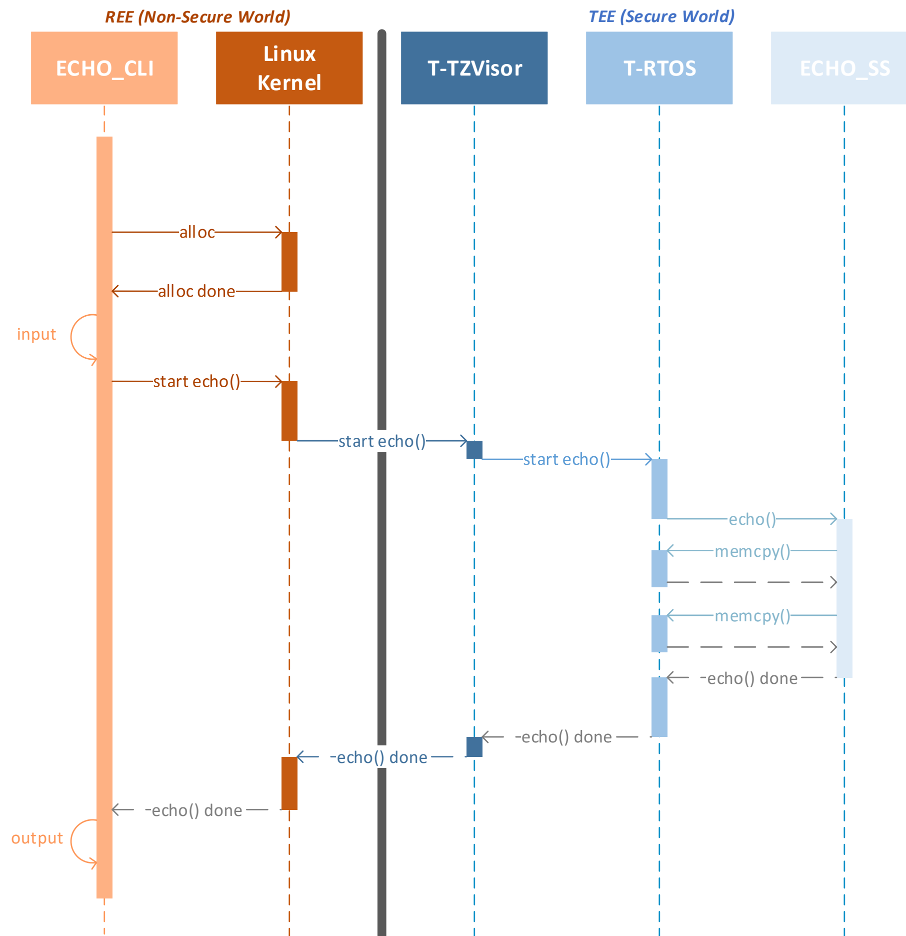


Figure 5.6: T-TZVisor: performing a secure operation

value, an array, or even a shared memory block. All facilities are specified from the TrustZone API and corresponding support at kernel level (for memory allocation) is provided by the kernel module. For the purpose of this demonstration an array is used to emulate an encryption key.

Once the TEE module dispatches the ECHO_SS, the secure service will be added to the T-RTOS ready-to-run task list. If no real-time task is running, the ECHO_SS is then executed. The execution flow of the ECHO_SS service is simple: it copies the data from the non-secure buffer (sender) to a secure memory region, and once the data is securely stored, it copies the secure data to non-secure buffer, but for a different interface (receiver). During this process of copying memory from the shared buffer to the secure memory region and from the secure memory region to the buffer, the ECHO_SS requests T-RTOS memory management facilities at the kernel level. Once the ECHO_SS finishes, the service is deleted and the T-RTOS is responsible for forwarding the answer to the REE, mediated through the hypervisor. Figure 5.6

shows a sequence diagram that describes the operations between the ECHO_CLI NSCApp and the ECHO_SS service to securely store a specific data in the secure world side.

Finally, the last stage is responsible for closing the session between the pair of applications. Interactively, the execution flow is similar to the one for establishing a session, but with slight differences. The ECHO_CLI triggers the communication by sending the session information to the TrustZone kernel module. This information does not include the UUID of the service, because the session is already established. The kernel module is then responsible for issuing a request to T-TZVisor for closing the session between the pair of services. The hypervisor handles the hypercall, and, if valid, performs a world switch and sends the request to the main interface of the TEE module running on T-RTOS. The TEE module is then responsible for analyzing the session context, and verifying if the information matches. The session is closed if the context and the information of the secure service are valid.

5.5 Discussion

With T-TZVisor I demonstrated how TrustZone technology, supported by other hardware trust anchors, can be adequately exploited to implement a secure virtualization solution which addresses security, without risking real-time and safety. Security starts by assuring a hardware root of trust as the basis for a secure boot process, and continues by establishing a chain of trust which validates, at boot time, all secure software components before executing them. In the case of authentication or decryption, at some point, if not successful, the device immediately enters in a secure lockdown state. During runtime, the real-time and safety properties of the system are not affected, because the main building blocks of a trusted execution environment are implemented as a lower-priority thread of the secure world RTOS.

The secure boot process, although successfully implemented for the developed solution, is a very platform-specific process, which needs to be supported by a hardware root of trust. The RoT provides a way to establish trust in an execution environment. Only an isolated execution environment equipped with a root of trust is a real "trusted" execution environment. Unfortunately, contrarily to Intel and AMD, ARM does not specify the root of trust for TrustZone. The existent TrustZone-assisted solutions [5.20, 5.11, 5.12, 5.13, 5.14, 5.15] usually assume the availability of a unique device key which is accessible only inside the secure world of TrustZone,

and use the device key to serve as the root of trust. Unfortunately, such device keys are not always available on many platforms. For example, Nuno Santos et al. developed a trusted language runtime [5.12] which required a device key to serve as the platform identity, but the platform where the solution was deployed, Nvidia Tegra 250, does not provide such hardware. Apart Zynq-7000, I am just aware that Samsung Exynios 5, FreeScale i.MX53 and OMAP 3 and 4 families provide such facilities. Based on the aforementioned facts, the work proposed by Zhao et al. which provides the root of trust for TrustZone-enabled platforms using SRAM physical unclonable functions (PUFs) [5.21] can be extremely useful and advantageous, specially on those platforms where secure storage for device key identity is not available.

As T-TZVisor is currently implemented, in a single-core configuration, the non-secure guest OSes and, consequently, the NSCApps only run when there is no real-time ready-to-run tasks in the system. Considering the complexity and integration level of the system, this can be somewhat limiting, specially if the real-time workload is very demanding, because it can lead to the starvation of the non-secure world side. Migration to multicore will help overcome this drawback. Several multicore configurations targeting asymmetric and symmetric (SMP) multiprocessing will be exploited and experimented in the future, to conclude which one better fits the FreeTEE requirements and use-cases. For example, an AMP approach will be adequate to run each OS simultaneously, however if the guest GPOS request a secure service to the T-RTOS while it is running a real-time task, the response will be delayed until the OS finishes executing such task, and the advantage will be neglected. An interesting approach could be the decoupling of the secure world software by an AMP configuration. One RTOS managing the real-time tasks running in one core, while one T-RTOS managing the secure services running in a different core, all under supervision of the T-TZVisor. Since the T-RTOS will just manage secure services, the dedicated core will be always available for answering the issued requests sent by the NSCApps running in the non-secure guest OS.

The TrustZone API, despite being a specification from ARM, is somewhat outdated. For this reason, I have already started the implementation of both GlobalPlatform TEE Client and GlobalPlatform TEE Internal specifications. The TEE client API, like TrustZone API, defines a set of interfaces for connecting to and invoking a secure service, from the non-secure side. The TEE internal API, on the other hand, defines the runtime support for the development of secure services running inside the TEE. Since the GlobalPlatform consortium is not only leading in providing specifications

and standards for the development of security solutions but also providing a more extensive specification than TrustZone API, this will guarantee a higher level of interoperability and standardization in the developed system.

The performed security analysis identified weaknesses of this solution regarding protection against inter-world communication and side-channel attacks. Communication between the secure and non-secure world is an essential part when implementing a TEE. The problem is no message-protection mechanism exists in TrustZone, which means man-in-the-middle attacks can be performed to manipulate the messages transferred through the channel (i.e., shared memory). Security analysts have proven the vulnerabilities on the TrustZone insecure channel, and to ameliorate this problem Jang et al. proposed a framework called SeCRet [5.19]. This framework builds a secure channel between the REE and TEE, by enabling REE processes to use session keys. SeCRet is a software framework that introduces a performance penalty into the system. My idea is to develop a secure hardware-based communication mechanism, or either a hardware monitor for supervising communication. Regarding side channel attacks, the presented solution does not have the resources to prevent such attacks. Nevertheless, T-TZVisor has been designed and deployed in a platform to be compliant with additional hardware accelerators and security modules. For example, SecBus [5.22] can be used to protect the system against on-board probing of the external memory bus, and physical attacks on the memory components (e.g., cold boot).

"*Security is not a product, but a process*". This famous quote by the "security guru" Bruce Schneier, clearly demonstrates that security is much more than the addition of protection mechanisms into the devices and products. The only way to effectively do business in an insecure world is to put processes in place, since the early beginning, which recognize the inherent insecure nature of developed solutions. The key is to start reducing the risk of exposure even during the development process. This means security should be considered at all stages of system development, starting even by establishing a secure software development process. All software components must be designed taking into consideration the Principles of High Assurance Software (PHASE) such as minimal implementation, least privilege, modular implementation, and following coding standards for security, and ultimately be validated by independent experts. As explained in the Chapter 3, I always follow the design principles for minimal implementation and least privilege. The modularity of the software components was also taken into consideration, but the use of an object-oriented approach will introduce a higher degree of modularity. The use of coding

standards, such as MISRA C/C++ and CERT C [5.23, 5.24], as well as the use of static analysis tools will help improve software quality, and finding as well as eliminating defects and possible breaches in sources. Such an approach is already in movement, and I have already, with the help of other researchers and students from my research group, started re-factoring the hypervisor code and using static analysis tools from PRQA [5.24].

Finally, it should be highlighted that the implemented solution is similar to Trusted Kernel-based Virtual Machine (T-KVM) [5.25], but with presented advantages in terms of scalability. T-KVM relies on ARM VE for implementing the virtualization support. T-TZVisor relies on ARM TrustZone, which is a technology which scales across the different ARM processor families. Regarding the novelty of integrating the building blocks of a TEE as a lower-priority thread of an RTOS, I have to claim it as mine, because despite both solutions present the same idea my work in progress paper FreeTEE [5.26] was released first. Anyway, T-KVM is one of the main artifacts and outcomes of the TRESCCA European Project [5.27]. This project received a total funding around 4M€. Virtual Open Systems was the research group responsible for its development, receiving a partial funding of 405K€. These arguments demonstrate, first, the huge interest of both academia and industry in solutions of this nature, and, second, the amount of monetary resources applied in this context.

5.6 Summary

The Internet of Things is an emerging key technology that paves the way for the next generation of smart connected systems. This explosion in connectivity created a larger attack surface area, and today's IoT systems are not completely prepared to fully and simultaneously fulfill the desired security level without risking the safe and real-time operation. The reason is because the way security is being conceived in the IT sector for satisfying the CIA triad requirements cannot be directly shifted to the OT context, where the main focus is not on digital information protection, but on how to control processes and change of states in a safe and secure way.

In this Chapter, I described T-TZVisor as a secure virtualization solution which addresses security without risking the real-time and safety properties of the system. I started by describing the desired goals with the development of T-TZVisor, and explaining its generic architecture. Then, I provided concrete details about the im-

plementation, namely on how to address security and guarantee a complete chain of trust since the early power-on reset until runtime. I conducted a set of experiments to evaluate the real-time behavior of the system, as well as an extensive security analysis and experimental validation. The experiments demonstrated that the proposed solution can effectively satisfy the strict requirements of a real-time environment, while offering a secure and safe operation during lifetime. Finally, I presented an extensive discussion about the identified benefits and limitations, and how I think these limitations will be, in the future, addressed and overcome.

References

[5.1] L. Tan and N. Wang, "Future Internet: The Internet of Things," in *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, vol. 5, pp. V5–376–V5–380, Aug 2010.

[5.2] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010.

[5.3] S. L. Keoh, S. S. Kumar, and H. Tschofenig, "Securing the Internet of Things: A Standardization Perspective," *IEEE Internet of Things Journal*, vol. 1, pp. 265–275, June 2014.

[5.4] A.-R. Sadeghi, C. Wachsmann, and M. Waidner, "Security and Privacy Challenges in Industrial Internet of Things," in *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pp. 54:1–54:6, ACM, 2015.

[5.5] D. Bodeau and R. Graubart, "Cyber resiliency engineering framework," *MTR110237, MITRE Corporation*, 2011.

[5.6] IIC, "Industrial Internet of Things - Volume G4: Security Framework." Industrial Internet Consortium, Version 1.0, Sept 2016.

[5.7] W. S. Technologies, "An Executive Guide to Cyber Security for Operational Technology." Wurldtech Executive Guide, 2016.

[5.8] A. Tavares, A. Carvalho, P. Rodrigues, P. Garcia, T. Gomes, J. Cabral, P. Cardoso, S. Montenegro, and M. Ekpanyapong, "A customizable and ARINC 653 quasi-compliant hypervisor," in *2012 IEEE International Conference on Industrial Technology*, pp. 140–147, March 2012.

[5.9] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive Formal Verification of an OS Microkernel," *ACM Trans. Comput. Syst.*, vol. 32, pp. 2:1–2:70, Feb. 2014.

[5.10] F. Armand and M. Gien, "A Practical Look at Micro-Kernels and Virtual Machine Monitors," in *2009 6th IEEE Consumer Communications and Networking Conference*, pp. 1–7, Jan 2009.

[5.11] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pp. 90–102, ACM, 2014.

[5.12] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications," *SIGARCH Comput. Archit. News*, vol. 42, pp. 67–80, Feb. 2014.

[5.13] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices," in *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pp. 367–378, IEEE Computer Society, 2015.

[5.14] A. Fitzek, F. Achleitner, J. Winter, and D. Hein, "The ANDIX research OS - ARM TrustZone meets industrial control systems security," in *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pp. 88–93, July 2015.

[5.15] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[5.16] Xilinx, "Zynq-7000 All Programmable SoC: Technical Reference Manual." UG585 (v1.11), September 2016.

[5.17] L. Sanders, "Secure Boot of Zynq-7000 All Programmable SoC." XAPP1175 (v2.0), April 2015.

[5.18] C. Lesjak, D. Hein, and J. Winter, "Hardware-security technologies for industrial IoT: TrustZone and security controller," in *Proceedings of the 41st IEEE*

IECON, pp. 002589–002595, Nov 2015.

[5.19] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment," in *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[5.20] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala, "On-board Credentials with Open Provisioning," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pp. 104–115, ACM, 2009.

[5.21] S. Zhao, Q. Zhang, G. Hu, Y. Qin, and D. Feng, "Providing Root of Trust for ARM TrustZone Using On-Chip SRAM," in *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, TrustED '14, pp. 25–36, ACM, 2014.

[5.22] J. Brunel, R. Pacalet, S. Ouaarab, and G. Duc, "SecBus, a Software/Hardware Architecture for Securing External Memories," in *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pp. 277–282, April 2014.

[5.23] PRQA, "Developing Secure Embedded Software." White Paper, 2016.

[5.24] PRQA, "Addressing Security Vulnerabilities in Embedded Applitications using Best Practice Software Development Processes and Standards." White Paper, 2016.

[5.25] M. Paolino, A. Rigo, A. Spyridakis, J. Fanguede, P. Lalov, and D. Raho, "T-KVM: A Trusted Architecture for KVM ARM v7 and v8 Virtual Machines," in *The Sixth International Conference on Cloud Computing, GRIDs, and Virtualization*, pp. 39–45, March 2015.

[5.26] S. Pinto, D. Oliveira, J. Pereira, J. Cabral, and A. Tavares, "FreeTEE: When real-time and security meet," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pp. 1–4, Sept 2015.

[5.27] CORDIS, "TRustworthy Embedded systems for Secure Cloud Computing Applications." European Commission, FP7-ICT, 318036.

"The most powerful tool we have as developers is automation."

- Scott Hanselman

6

Design Automation: It's Not Just about Technology

The IoT paradigm is driving the next wave of technological and business transformation. The first wave of IoT mainly focused on technology and validated the power of connectivity. The next wave of transformation, which will definitively led companies to achieve significant market success, will be completely value-oriented, leveraging technology-design dynamic to play a key role for the success of IoT 2.0.

In this Chapter, I present a domain-specific language which mainly decouples the building blocks of virtualization-assisted TCB, leveraging easy customization towards target platforms and applications. I describe how a service-oriented programming model can help automating the generation of a customizable TCB system, ensuring correctness by design while powering components development based on service compositions, and boosting the development time due to the high abstraction level of the process.

This Chapter is organized as follows: Section 6.1 motivates the need of such a design automation support, and Section 6.2 presents a brief introduction to domain-specific languages. Section 6.3 introduces the generic elaboration DSL and its design work-

flow, and Section 6.4 describes vEL DSL as an entailment of EL DSL to assisted hypervisor design. The advantages and disadvantages of the presented DSL technology are discussed in Section 6.5, and, finally, Section 6.6 ends this Chapter with a brief summary and final conclusions.

Related Publications

The ideas and results presented in this Chapter have been accepted for evaluation and are under review as:

- **S. Pinto**, J. Martins, J. Cabral and A. Tavares, "*Hyper-Language: A Domain Specific Language for easing design, integration and configuration of embedded hypervisors*", in Journal of Systems Architecture, 2017.

6.1 Motivation

According to [6.1], IoT 1.0 has been excessively technology-oriented which led to a very high expectation, exceeding the performance and giving rise to speculative business bubble. It has been reported 50 billion of connected devices and 1.5 trillion dollars of IoT value in the coming years, driving IoT to the top of Gartner hype cycle. Figure 6.1 shows, according to Gartner Inc. [6.2], technology trends of highest priority for organizations facing rapidly accelerating digital business innovation.

A real shifting to the IoT 2.0 will be value-oriented, with value created for and by users, through the leverage of design-technology partnership as the design plays a key role in moving the wave of IoT technology forward along with the right supporting processes and enabling technology [6.1]. That is to say, due to its innate technological integration and new user experiences, the IoT demands a significantly higher level of effective technology-design dynamic to move beyond its current hype bubble over the next few years [6.1]. Mainly concerning the security domain, there are several technologies for detection, prevention and response which are essential for basic security but must be strategically and synergistically integrated to foster a holistic and robust IoT security value or solution. Therefore, to build such an effective

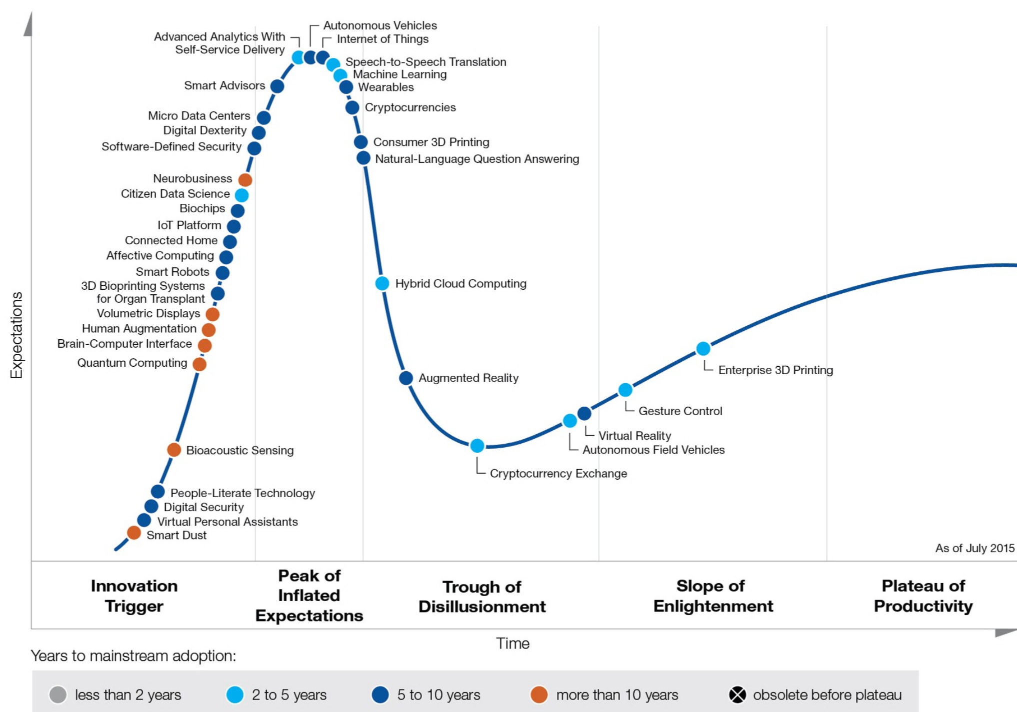


Figure 6.1: Gartner' emerging technology hype cycle (2015)

technology-design dynamic and consequently succeed in IoT 2.0, the following five ways were proposed in [6.1]:

1. **Agree to a clear problem statement** that assess value to the user and driven by a professional experienced in user-centered design and design-thinking processes;
2. **Appoint a systems lead who understands design** from both technology stack and user perspectives, as well as user experience;
3. **Work with designers who understand technology** to leverage a technology-aware approach to design which tackles product lifecycle management, inter-operability with existing and new products, as well as personalization and big data;
4. **Follow a build-test-learn process** to mitigate churn by building desired users' experience, observing their behavior, and sustaining that experience based on what they learn.
5. **Simplify for success** by approaching IoT development with a "less is more" mandate and so, avoiding friction of any kind in the user experience.

I agree on the above premises and also believe they will be achieved with some automation level or design agility to easily and quickly promote customization that scale according to resource-constrained devices, as well as lifecycle upgrading under users' demand. Specifically, the virtualization layer on the IoT endpoint device stack should be easily customized to better tackle different use cases scenarios in terms of security, safety, real-time and functionalities consolidation under continuous learning and improvement. To leverage virtualized TCB technology based on an incremental approach as suggested on Chapter 1 under "Research Questions and Methodology", edge paradigms for automated software design and software modularity such as generative programming, compositional programming, domain-specific language and program refactoring have been explored and applied in the implementation of this thesis' proposed solution.

6.2 Domain-Specific Languages

A DSL is a custom language targeting small problem domain by describing and validating it in terms native to the domain, i.e., DSL not only raises the level of

abstraction but also provides domain-specific abstractions [6.3]. Therefore, DSLs provide programmers with the ability to program more directly in the domain and also in a more declarative way (i.e. specifying what to do) than the imperative one (specifying how to do) as happens with general-purpose languages. DSLs have been proposed as a solution that can provide productivity, performance, and portability for high-level programs in specific domains such as high performance systems [6.4], dynamic binary translation (DBT) [6.5, 6.6], robotic [6.7, 6.8], Cryptography (Cryptol[1]), simulation of system-of-systems [6.9] and software-defined radio (SDR) [6.10], just to name a few. DSLs can support automated design process through well-defined design flow with clear and unambiguous abstraction levels, models, and transformations.

6.2.1 Related Work

Before going deeper into DSL' issues, benefits and concepts, let's introduce some of the above mentioned DSLs. LLDSAL [6.5] and EBT [6.6] are two DSLs designed to specify dynamically generated code, and to support the development of DBT-assisted code analysis tools, respectively. In [6.7, 6.8] ReApp project is presented along with an ontological model for industrial robotics which leverages computing compatibility within a robotic system. Domain ontologies are used to exploit the implicit semantics contained in Automation Markup Language (AML). AML descriptions will be uplifted into semantic models for automatic reasoning and represented using the Resource Description Framework in conjunction with the Web Ontology Language. Cryptol is a domain-specific language for specifying cryptographic algorithms and a Cryptol-based algorithm implementation resembles its mathematical specification more closely than an implementation in a general purpose language. In [6.4] is described Delite, a modular compiler framework targeted from embedded Scala front-end, and DSLs developed from it. Delite is essentially a Scala library used to build intermediate representation, performance optimizations and generate parallel code for multiple hardware targets like CPUs and graphics processing units (GPUs). DSLs are developed by extending reusable Delite components with domain-specific semantics. Any service provided by Delite can be overridden by a particular DSL with a more customized implementation.

In [6.9] is proposed an architecture-driven modeling method which conforms to the principle of architecture-driven development, uses ontology techniques to build

[1] http://www.cryptol.net/

equipment system-of-systems architecture model and sub-domain ontology. The proposed method performs architecture driven simulation modeling to realize the transformation from architecture models to simulation model frameworks; employs ontological metamodeling to design domain-specific modeling languages based on the comprehensive usage of architecture models, sub-domain ontologies and formalisms; integrates domain-specific simulation models from various domains using the model framework, and supports the composable development of simulation applications. In [6.10] is described the application of model-driven development, and more specifically, domain-specific modeling to the software defined radio domain. The presented approach raised the abstraction level of the radio platform beyond operating systems and middleware, increasing productivity, correctness and robustness of new designs of SDR systems. OptiSDR [6.11] is a DSL developed by extending reusable Delite components with SDR domain-specific semantics. Basically it matches high level digital signal processing routines for software defined radio to their generic parallel executable patterns targeted to heterogeneous computing architectures, including combination of hybrid GPU-CPU and DSP-FPGA. Ontology has been used to leverage models and DSLs interoperability, as well as to formally and uniformly exploit and integrate domain knowledge in DSL implementations [6.7, 6.8, 6.9, 6.12]. In [6.12] is described a novel DSL implementation paradigm using an ontology-assisted knowledge base to formally and uniformly exploit the knowledge needed for optimizations.

6.2.2 The Domain-Specific Development Process

Based on the observation that many software development problems can more easily be solved by designing a special-purpose language, Domain-Specific Development (DSD) applies such an approach to lowering the complexity of the system under development [6.3].

Figure 6.2 [6.3] depicts the DSD process consisting basically in the following steps:

1. Identify the variable part of the problem (i.e., the domain variability) and represents it by a DSL;
2. Identify the fixed part of the problem (i.e., the domain commonality) and address it using classic design, coding and testing methods;
3. For each instance of problem in a domain, create a model or expression using

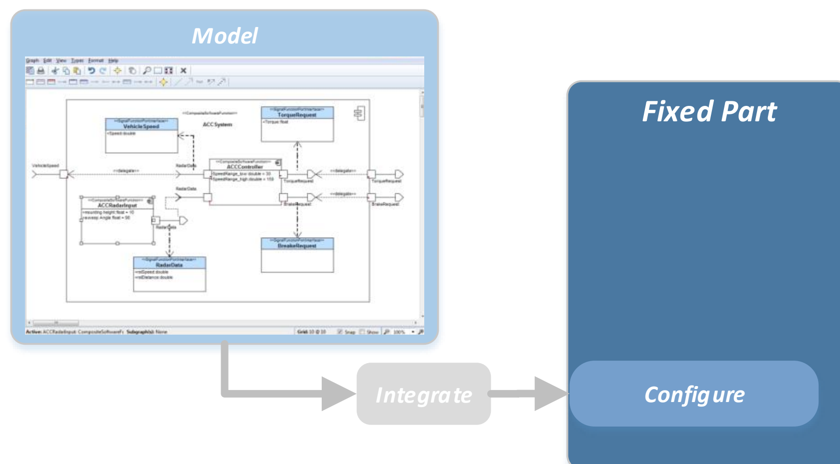


Figure 6.2: Domain-specific development

the DSL;

4. For each instance of a problem, integrate its previous created model or expression with the fixed part of the solution.

According to the size and shape of the domain under study, the domain commonality which captures the architectural patterns and exposes extension points to the domain variability can be implemented as a framework, a platform, an interpreter or an API. Two possible approaches for the integration of the fixed and variable parts of the domain are:

- *Interpretative approach*, where the fixed part contains an interpreter for the DSL used to express the variable part;
- *Code-generation approach*, where code generator or model compiler is used to fully convert a particular model or expression into imperative code that can be compiled together with the remainder of the solution for the resulting application.

6.2.3 The Domain-Specific Development Benefits

By applying the above steps, DSD offers an increased design agility, mainly due to the combination of the following factors [6.3]:

- A DSL gives the ability to work in terms of the problem space and DSL-assisted approaches are becoming particularly attractive to systems integrators;

- Working in terms of the problem space instead of in solution space can make the models more accessible to those not familiar with the implementation technology;
- Models expressed using DSLs can be validated at the level of abstraction of the problem space thus, left-shifting potential errors to the modeling or compilation phase instead of allowing them to creep into the runtime system;
- Models can be used to configure an implementation consisting of multiple technologies of different types;
- A DSL can improve developer productivity by providing a set of domain-specific APIs for models manipulation;
- A DSL can leverage portability once important domain knowledge is captured into a model. For instance, it will simplify the migration of a solution from one technology to another, or between versions of the same technology by simply modifications to DSL back-end (i.e., the generator or interpreter).

6.2.4 DSL Implementation Approaches

Existing implementation choices for DSLs range from internal DSLs (i.e. purely embedded in a host language) to language workbenches [6.3, 6.4]:

- *Internal* or *Purely embedded DSLs* are implemented as libraries in a flexible host language and emulate domain-specific syntax. Its main benefit is ease with build and compose, since they can interoperate freely within the host language. However, as an interpreted DSLs they suffer from high overhead;
- *Parser-generator*, *stand-alone* or *external DSLs* are implemented with an entirely new compiler that performs both front-end tasks such as parsing and type checking, as well as back-end tasks like optimization and code generation;
- *Compiled embedded DSLs*, occupies a middle-ground between the internal and external approaches as they embed their front-end in a host language like internal DSLs, but use compile- or run-time code generation to optimize the embedded code;
- *Languages workbenches* are further option to textual DSLs which define DSL

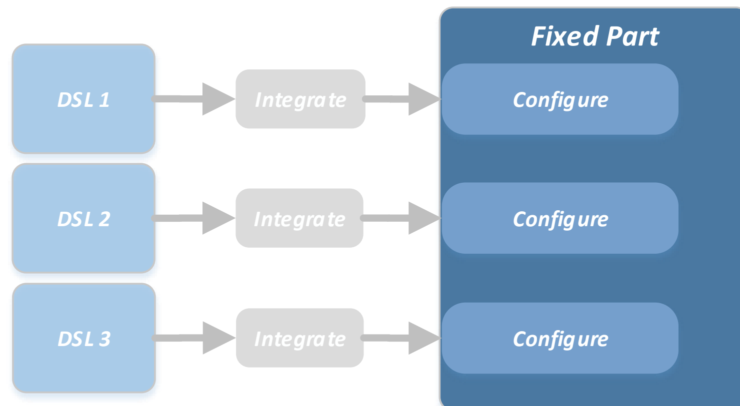


Figure 6.3: Multiple DSL integration

tools targeting textual DSLs by allowing developers to define a DSL and its graphical editor and model compiler or generators.

Well-known languages workbenches are Eclipse Modeling Framework and the Eclipse Graphical Editor Framework (EMF/GEF) [6.13, 6.14], Microsoft's Visual Studio Team System Domain Specific Language Tools [6.3], MetaCase MetaEdit+ [6.15], Xtext/Xtend [6.16] and the Generic Modeling Environment (GME) [6.17].

Using a single DSL to fully model or express a very complex problem domain can become a daunting task, if such domain touch several and different concerns. Hence, it is suggested in [6.3, 6.4, 6.9, 6.18] to compose multiple subsystems in their own different domains and DSLs, each one handling a different dimension of complexity in the problem domain, while devising a mechanism for the integration and interoperability of individual DSLs (see Figure 6.3 [6.3]).

6.3 EL: The Elaboration Language and Workflow

EL is a small declarative DSL assisting fast, effective and flexible design of custom frameworks following a generative approach for code generation. It is implemented with Xtext and Xtend [6.16] and it approaches à la Service Component Architecture (SCA), a programming model for building service-oriented architecture (SOA)-based applications and systems [6.19], extended with the assignment abstraction to specify dependencies among component's properties. Component is the basic EL model entity, and it is used to describe services compositions relying on a well-defined set of abstractions such as services, promotes, references, properties and bindings to specify its interactions with other components.

Xtext [6.16] is an Eclipse-assisted framework that allows implementation of DSLs together with their integration in the Eclipse IDE. Besides allowing a simple implementation, the Xtext covers every aspects of language infrastructure such as, lexer, parser, abstract syntax tree, scoping, linking, code generator or interpreter. These runtime components are based on Eclipse Modeling Framework (EMF). Although Java can be used for customizing the implementation of a DSL, Xtext leverages the use of Xtend, a Java-like programming language completely interoperable with the Java type system, which features a more compact and easier to use syntax and advanced features such as type inference and lambda expressions.

Due to its SOA-based programming model, EL can be seen as a horizontal parser-generator kind of DSL which can be used for modeling nearly any domain. It also assists automated code generation by focusing on domain engineering and feature modeling, as it follows a generative approach. Based on configuration knowledge it automates the selection and assembly of components which will describe the system architecture. Hence, the system' designer only needs to specify in abstract terms "*the what*" and the EL' back-end generates the desired system or component by using the configuration knowledge. In doing so, EL decouples models from generated files, leveraging an effortless modification of models while the back-end will automatically incorporate changes. However, the domain logic must be beforehand captured in a factory of artefacts (i.e., artefacts that fits the domain under study), consisting of implementation and simulation views, as well as elaboration artefacts which states how abstract requirements will be translated into concrete set of components.

According to EL DSL specification, an architecture is a set of several and different kind of components, and how they can depend on each other. It will be setup following a divide and conquer strategy, and through explicitly combined usage of EL' binding and assignment abstractions. To leverage a sustainable or clean architecture, it must exist in the early stage of the project while being completely recognizable after successive refinement along the system' lifecycle. Below are some recommendations concerning the way an EL-assisted architecture must be setup, as well as promoting benefits by following them:

1. Split the system under study into several components;

2. Specify component's dependencies using EL' binding and assignment abstractions and also provide a hierarchical and recursive elaboration algorithm;

3. Entail each component with views according to several kinds of artefacts, such as implementation, simulation and/or elaboration views. Notice that a com-

ponent can have multiples implementation, simulation and elaboration views;

4. Populate the component factory by expressing architectural grouping in your package names, according to the targeted/generated language. For instance, in C++, it can be achieved through the combined usage of nested directories and namespaces. Start by organizing first at component-level (i.e., grouping related components into packages) and only then at package-level, depending on the architecture of the domain under study (e.g., by layers or by subsystems to separate system' functionalities according to the size and organization of the domain logic).

The fixed-part of an EL-assisted architecture is given by the Elaborator and EL models in a top-level SCA composite. The Elaborator is represented by a Java class which offers a `generate(...)` method, hard-coding in its body a hierarchical and recursive elaboration order, and scope for patching and configuration of different EL models in a top-level SCA composite. The elaboration order and scope is established based on the dependencies of components and component's properties, being the latter as expressed by EL' assignments. Each selected designer-defined elaboration class is loaded to extract its `generate(...)` method using Java reflection and then accordingly called. The default `Elaborator::generate(...)` follows a deterministic depth-first-search (DFS) strategy based on the hierarchical structure of the fixed-part, enabling designers to reason about the flow of design decisions. It also establishes mechanisms of parametrization based on priorities of declaring and assigning values to parameters to avoid inherited or constrained parameters to be overridden by external statements [6.20]. The variable-part of an EL-assisted architecture is specified by each individual component elaboration and annotated implementation views, later assembled according to the top-level SCA composite. EL-assisted DSLs are developed by establishing a new top-level SCA composite for a specific domain and the component factory, as well as by overriding the domain-specific Elaborator at both class- and/or generate method-levels with a more customized implementation.

Main benefits of a sustainable EL-assisted architecture is avoiding architectural erosion under continuous learning and improvement during the system' lifecycle[2]. Architectural erosion happens when the original architecture is completely lost while coupling and dependencies are totally out of control, giving raise to cyclic dependencies between components. Enforcing an evolving architectural blueprint over the

[2]https://dzone.com/articles/love-your-architecture

lifetime of a system will simplify the system maintenance which is mostly and directly coupled to the system' architectural integrity. Other benefits of a sustainable EL-assisted architecture can be enumerated as follows:

1. Changes will be much more local and it will be easier to reuse parts of the system;
2. It will be easier to pass a system from a development team to a maintenance team;
3. It will become much easier to harden a system against security vulnerabilities mainly due to new coming kind of attacks;
4. It will be simpler and more straightforward to add new features, fixing bugs or implementing changes;
5. It will be easier the implementation of automated architecture checks by: packaging concepts and naming strategy that reflects the architectural model; controlling the packages' size and avoiding cyclic dependencies between packages, as well as internally between components in a package.

6.3.1 EL Workflow

EL approaches a 2-stage design workflow, starting with the compilation of a top-level SCA composite (i.e., the only compilable component) followed by an elaboration

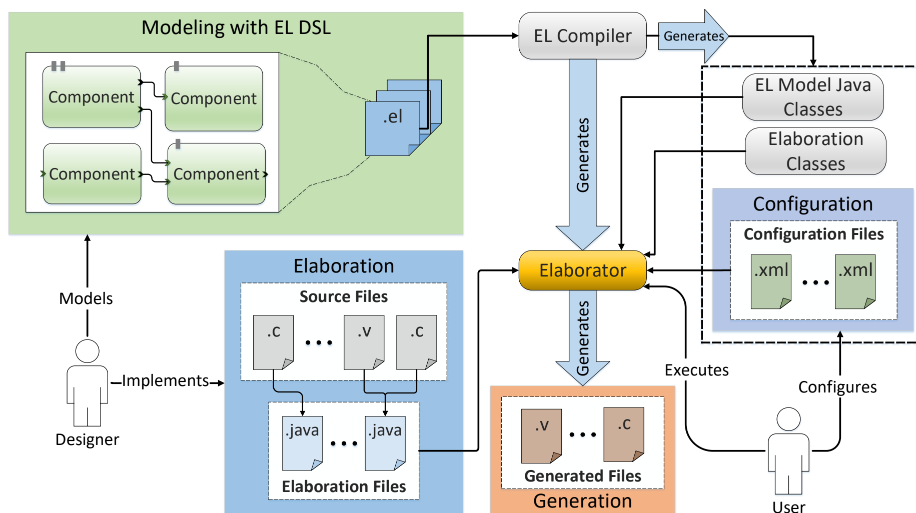


Figure 6.4: EL 2-stage design workflow

stage for the final generated application or system source files (see Figure 6.4). The following tasks will be carried out in the former stage: (1) SCA model representation will be converted into .el files, (2) .el files will be syntactically and semantically validated and only then converted into java classes, and (3) an elaboration program (i.e., the Elaborator) and Extensible Markup Language (XML) configuration files for each EL model will be generated. Problem domain model entities will be designed by properly tailoring EL components to specialized programming abstractions of the problem domain (e.g., for virtualization-assisted systems concepts such as inter-partition communication, Health monitor, security module, VM, VM manager, guest operating systems, virtual CPU, time manager, virtual file system, VM scheduler, and so on). Concrete vEL DSL programs describing SCA composites will be created through the entailment of abstract hypervisor components with configuration, elaboration and annotated behavioral/implementation artefacts (see Listing 6.1 and Listing 6.2 for a component and its artefacts' code snippets).

Listing 6.1: EL virtual board model

```
import "languages.el"
/*{...}*/
component VirtualBoard (C) {
   subcomponents:
      VRegFile vregfile
      VCP15 vcp15
      VTimer vtimer
      VGIC vgic
      VPeripheral <> vperipherals
   references:
      RegFileContextSwitch
         vregfile_init
      ContextSwitch vtimer_cs
      ContextSwitch vcp15_cs
      ContextSwitch vgic_cs
      /*(...)*/
   services:
      ContextSwitch contextswitch
      /*(...)*/
}
```

Listing 6.2: Annotated artefact

```
#ifndef __VBOARD_H__
#define __VBOARD_H__

/*{...}*/

typedef struct Vcpu{
   @@VREGFILE_STRUCT@@
   @@VCP15_STRUCT@@
   @@VGIC_STRUCT@@
}VCPU;

typedef struct Vboard{
   VCPU vcpu;
   @@VTIMER_STRUCT@@
   @@VPERIPH_STRUCT@@
}VBOARD;

/*{...}*/

#endif
```

For semantic purposes, the file languages.el defines a language entity called C, specifying the implementation language and an attribute for the annotation symbol delimiting patching locations. Semantically, the implementation files of a compos-

ite and all associated components should be written in the same general-purpose programming language.

Listing 6.3: Generated virtual board configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<component type="VirtualBoard">
   <elaboration default="SpecificVirtualBoardElaboratorZynq">
      SpecificVirtualBoardElaboratorZynq</elaboration>
   <properties></properties>
</component>
```

The elaboration stage is an interplay among several entities and their artefacts such as components models and their artefacts, as well as a structured folder layout. A component' configuration file specifies only one among possible elaboration files for that component (see Listing 6.3) while each elaboration file, Listing 6.4, explicitly specifies all implementation files to be patched as the behavior of a given component can be spread into several annotated source files.

Listing 6.4: Virtual board elaborator class

```
public class SpecificVirtualBoardElZynq extends
    AbstractVirtualBoardEl{
   public void generate(){
      openAnnotatedSource("src/vboard.c", "/src/platform/zynq/");
      /*{...}*/
      RegFileContextSwitchElaborator vregfile = (RegFileCSwitchEl)
         getElaborator((Component) target.get_vregfile_init());
      for(String inc : vregfile.getRegFileCSwitchElHlist())
         replaceAnotation("INCLUDE", "#include \""+inc+"\"");
      /*{...}*/
      replaceAnnotation("VCP15_STRUCT", struct_name + " " +
         identifier_name +";");
      replaceAnnotation("VCP15_INIT", velab.
         getCSwitchElContext_init() + "(&(p_vboard->vcpu."+
         identifier_name+"));");
      replaceAnnotation("VCP15_CSAVE",  velab.
         getCSwitchElContext_save() + "(&(p_current_vboard->vcpu."+
         identifier_name+"));");
      /*{...}*/
   }
}
```

Table 6.1: Available EL's keywords

Keyword	*Description*
`annotation`	Defines the character that limits the annotations.
`as`	Renames a promoted reference or service.
`bind`	Binds a reference to a service.
`bool`	Component's property data type.
`compile`	Tells to compiler which is the top level component.
`component`	Defines a component.
`final`	Defines a component has a concrete elaboration.
`import`	Imports the content of a specified file.
`int`	Component's property data type.
`interface`	Defines a set of functions used by a service or pointed by a reference.
`is`	Inherits the specified component.
`float`	Component's property data type.
`language`	Defines a language.
`promote`	Promotes a reference or service from a subcomponent to a component.
`properties`	Defines the properties set of a component.
`reference`	Defines the reference used in a promote or in a bind operation.
`references`	Defines the reference set of a component.
`restrict`	Restricts the values that a property can take to a user's defined set.
`service`	Defines the service used in a promote or in a bind operation.
`services`	Defines the service set of a component.
`string`	Component's property data type.
`subcomponent`	Defines the subcomponents set of a component.
`to`	Connects a reference to a service in a bind operation.

As shown in Listing 6.4, a 2-way patch manipulation is performed over annotated implementation files using elaboration APIs such `openAnnotatedSource()` and `replaceAnnotation()`, among others, for: (1) user input and string replacements and (2) interface functions replacements according to the binding of reference to service as specified by the EL interface models. To support the compilation of all Java files presented in the EL folder and also running the elaborator program, some build system scripts are generated.

Table 6.1 presents the available keywords provided by the EL DSL. EL enforces several rules that the programmer must respect in order to develop a valid model. They are divided in five rules' categories such as generic, assignment, promoting, binding, and importing. If one of the EL rules is not met, the compiler will throw an error. Some rules, according to the respective category, are presented below:

- **Generic Rules**

  - Cycles of subcomponents or inherited components are not allowed;

  - A component cannot be instantiated inside itself (as a subcomponent).

- **Assignment Rules**

- A property cannot be in both sides of an assignment;

- A component cannot assign to a property more than once.

- **Promote Rules**

  - A reference that is already bound cannot be promoted;

  - Only references and services from subcomponents can be promoted.

- **Bind Rules**

  - All the defined services must be bound to a reference;

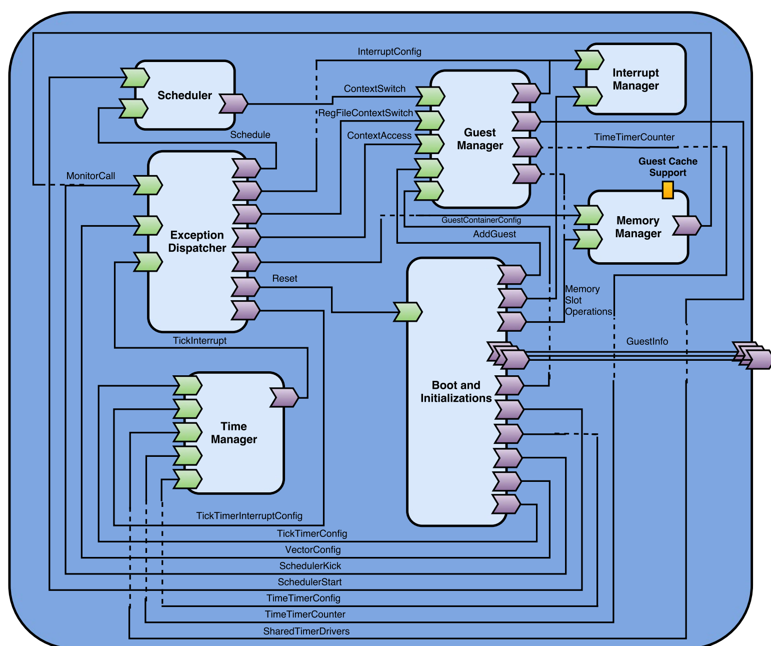
  - A component cannot bind two references to the same service.

6.4 vEL: a VMM-assisted DSL

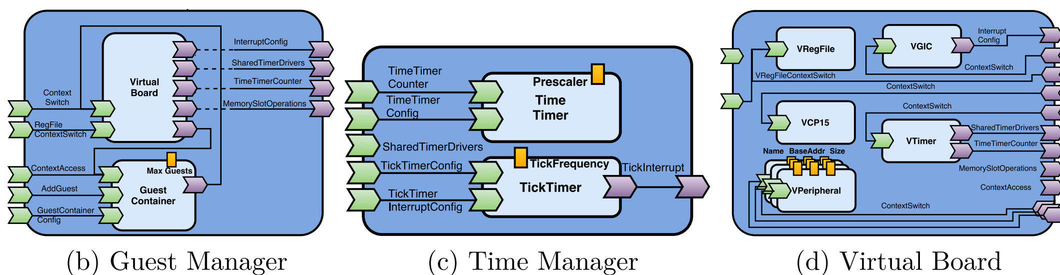
To model a hypervisor as an SCA system, the virtualization domain was distilled into bounded components, interaction maps among components, and feature diagrams expressing each component commonalities and variabilities. Feature diagrams and interaction maps serve elaboration and integration purposes, respectively. Following SCA building-block approach to create virtualized systems, the vEL-DSL assisted by EL models was created to leverage parametrization of a monolithic hypervisor framework (Figure 6.5) to specific concrete implementation of TZVisor hypervisor family annotated and hard-coded into components' behavioral artefacts.

The original TZVisor code was first refactored to fix the TZVisor framework and to accordingly annotate all implementation artefacts at code-level, while populating the TZVisor component factory. The modeling of TZVisor in the vEL DSL followed a top-down approach, starting with the implementation of the top-level Virtualization Stack component. Each one of the composite components (i.e., composed by several other components), has a dedicated .el in the project. Atomic components, that are self-contained, are defined within the source file of the top-level component that instantiates them. Two other files exist for language and interface definition. Table 6.2 shows the list of the implemented EL files, along with the components, interfaces or language defined within them.

The *languages.el* defines a type, that signals in what language a given component's behavior is implemented. Although some components in the model have a mixed



(a) TZVisor



(b) Guest Manager (c) Time Manager (d) Virtual Board

Figure 6.5: TZVisor framework as an SCA composite

C/Assembly implementation, in the EL implementation all of them are defined as implemented just in C. This is because the EL language forces all components in the model to have the same implementation language, which, as this use case shows, is a considerable limitation. The *tzv_interfaces.el* defines all the interfaces through which all the components interact with each other. The *virtualizationStack.el* is the top-level component which generically models the connection between the hypervisor and its respective guest OSes, and also allows configuration of guest properties. The *tzvisor.el* its the model of TZVisor and contains the main components inherent to the hypervisor: guest manager, time manager, exception dispatcher, among others (see Figure 6.5a). The *bootandinitializations.el* implements the model of the boot components and respective subcomponents. This includes the configuration of several architectural features such as MMU, cache and branch prediction. The *guestmanager.el* implements the model of the guest manager component, and allows fine-grained configuration over the maximum number of supported guests, as well

Table 6.2: Implemented EL files

File	***Contents***
languages.el	Definition of C language type.
tzv_interfaces.el	Definition of all interfaces types in the model.
virtualizationStack.el	Definition of Virtualization Stack and Guest components.
tzvisor.el	Definition of TZVisor, Scheduler, Exception Disptacher, Interrupt Manager and Memory Manager components.
bootandinitializations.el	Definition of Boot and Initialization components.
main.el	Definition of Main, Hardware Initialization and Software Initialization components.
guestmanager.el	Definition of Guest Manager and Guest Container components.
virtualboard.el	Definition of Virtual Board, VRegFile, VCP15, VGIC, VTimer and VPeripheral components.
timemanager.el	Definition of Time Manager, Time Timer and Tick Timer components.

as the definition of the virtual board structure (Figure 6.5b). The *virtualboard.el* implements the virtual board model, which in this specific case includes several ARMv7 architectural components such as register file, CP15, GIC, among others (Figure 6.5d). The *timemanager.el* implements the model of the time manager component, and allows configuration of the hypervisor tick time and other timing-related properties (Figure 6.5c).

The top-level component in the *virtualizationStack.el* file is depicted in Listing 6.5. It basically instantiates two other subcomponents: an array of Guest components and the TZVisor component itself. These components are connected by a `GuestInfo` interface. The TZVisor has a `GuestInfo` reference, and each of the subcomponents of the array implements a service following the same interface. The connections are made using the bind statement.

Listing 6.5: EL Virtualization Stack Model

```
component VirtualizationStack(C){
   subcomponents:
      Guest <> guests
      TZVisor hypervisor
   bind hypervisor.guest_info to guests.info
}
```

The Guest component is also defined in the same file (*virtualizationStack.el*), as shown in Listing 6.6. The component has three properties, which are accessible via the `GuestInfo` interface. This properties allow the configuration of guest features such as the path for the guest binary image in the system, as well as the memory segment the guest image should be loaded. There is also the declaration of the service

of type `GuestInfo`, which is used to bind the guests to the TZVisor component, as explained earlier.

Listing 6.6: EL Guest model

```
component Guest(C){
   properties:
      string name : "Invalid"
      string binary : "Invalid"
      int slot restrict [2->16] : 2
   services:
      GuestInfo info
}
```

Listing 6.7 presents the guest elaborator class, which is responsible for explicitly specifying where all implementation files must be patched to generate the code according to configuration properties of the Guest model.

Listing 6.7: Guest elaborator class

```
public class SpecificGuestElZynq extends AbstractGuestEl {
   /*{...}*/
   public void generate(){
      /*{...}*/
      openAnnotatedSharedSource("Zynq/EL_guests.S", "/src/arch/
         armv7/");
      String index = String.valueOf(target.get_slot());
      int n = target.get_slot() - 2;
      String guestbin = target.get_binary();
      String incbin = ".section \".guest"+index+"_bin\", \n"
                           + ".global guest"+index+"_bin\n" +
         "guest"+index+"_bin:\n" +
         ".incbin \"guests/"+guestbin+"\";\n\n";
      replaceAnotation("GUEST_BIN", incbin);
      openAnnotatedSharedSource("Zynq/linkerscript.ld", "/src/arch/
         armv7/");
      /*{...}*/
      replaceAnotation("GUEST_MEM", guestmem);
      /*{...}*/
   }
}
```

6.5 Discussion

With the development of EL and vEL, I demonstrated how DSL technology can help automating the generation of a customizable virtualization-assisted TCB, ensuring correctness by design while boosting the development time. EL represents the generic elaboration DSL, while vEL is seen as an entailment of EL DSL to vertically assist in the hypervisor design.

During the refactoring of the monolithic TZVisor C code to vEL one, it was visible to me that EL' newbies could easily face severe coding crosscutting and tangling which consequently can lead to architectural erosion while strongly binding and interfacing vEL components. Hence, I have been now refactoring the monolithic TZVisor C code following microvisor approach (i.e., a merging of hypervisor coding style assisted by a microkernel architecture), making the inter-partition component one of the main building blocks of the new architectural style. This way, the dependence side effect will be easily removed and managed which first improves a new vEL-assisted TZvisor reference architecture while leveraging long term maintenance during the lifecycle of any deployment of TZvisor-based systems. Furthermore, in virtualization for IoT domain more just than functionalities' consolidation and performance concerns have been raised but also security, safety and real-time as well. For the former, a loosely-coupling architectural style will be essential to better accommodate an evolving secure hypervisor architecture. I am still focused on the refactoring journey, trying to also improve modularity at behavioral artefact code through migration to C++. In doing so, I have been applying secure coding standard like MISRA C++ by eliminating some C++ dialects such as static and dynamic polymorphisms (e.g. templates and virtual methods), as both can severely jeopardize performance, as well as determinism. During C++ refactoring I have been tackling cyclic dependencies among components by applying other software craftsmen techniques like adding interfacing to directly eliminate dependencies among concrete classes, thus, allowing dependencies only on abstractions like interfaces. For instance, I have been using creational patterns like factory design pattern to avoid the creation of any concrete classes inside any other concrete one or calling a kind of `getInstance(...)` method.

I felt that while EL can still be a horizontal DSL based on the set of abstractions such as services, promotes, references, properties and binding, any vertical EL-assisted DSLs (e.g., vEL) should be mapped or lowered, as well as relaxed to the problem domain semantics to better expressivity and ease of use. Although it is possible to bind the whole domain components through EL' composite components

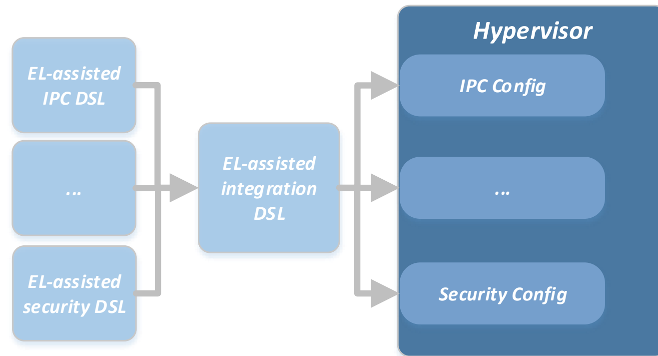


Figure 6.6: Ontology-driven integration DSL

as done so far with timer manager, guest manager and scheduler, just to name some TZVisor composite components, I also felt that it will be better to employ multiple EL-assisted DSLs, each handling a different dimension complexity in the IoT virtualization domain. For these reasons, I have been working with other in-house projects to vertically shape EL-assisted DSLs through their own individual domain ontologies, while making them fully interoperable. In so doing, an upper ontology has been devised and developed along with a relation ontology (RO) to express how each vertical domain and application ontologies can be instantiated, as well as the way component and each vertical domain can easily be integrated. Figure 6.6 illustrates how one can go beyond the limited scope of each ontology-driven DSL by modeling a problem domain with several vertical EL-assisted DSLs which will be later integrated through an integration DSL according to the devised and developed RO.

In so doing, the DSD process will be formally enhanced with ontology as the latter is naturally a formal, explicit specification of a shared conceptualization in a computation-independent manner.

6.6 Summary

The Internet of Things is a new reality that is completely enriching our everyday life. IoT 1.0 was completely focused on technology and on demonstrating the power of connecting billions of devices. The problem is the technological-centric approach reached the limit of saturation, and the shift for the next wave of transformation will require technology-design dynamic to play a key role for the success of IoT.

In this Chapter, I presented a DSL which mainly decouples the building blocks

of virtualization-assisted TCB, leveraging easy customization for different platforms and application requirements. I started by presenting a brief introduction to domain-specific languages. Then, I described how a service-oriented programming model can help automating the generation of a customizable virtualization environment, guaranteeing correctness by design while boosting the development time. Hence, I described the generic elaboration DSL and its design workflow, and then I explained the vEL DSL as an entailment of EL DSL to assist hypervisor design. Finally, I discussed the advantages and disadvantages of the developed DSL technology, and how I think main identified limitations are being or will be addressed and overcome.

References

[6.1] S. Nelson, “The Internet of Things Needs Design, Not Just Technology,” in *Harvard Business Review, Webinar*, July 2016.

[6.2] Gartner, “Gartner’s 2015 Hype Cycle for Emerging Technologies Identifies the Computing Innovations That Organizations Should Monitor.” Press Release, August 2015.

[6.3] S. Cook, G. Jones, S. Kent, and A. C. Wills, *Domain-Specific Development with Visual Studio DSL Tools*. Pearson Education, 2007.

[6.4] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, “Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages,” *ACM Trans. Embed. Comput. Syst.*, vol. 13, pp. 134:1–134:25, Apr. 2014.

[6.5] M. Payer, B. Bluntschli, and T. R. Gross, “LLDSAL: A Low-level Domain-specific Aspect Language for Dynamic Code-generation and Program Modification,” in *Proceedings of the Seventh Workshop on Domain-Specific Aspect Languages*, DSAL ’12, (New York, NY, USA), pp. 15–20, ACM, 2012.

[6.6] S. Makarov, A. D. Brown, and A. Goel, “An event-based language for dynamic binary translation frameworks,” in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 499–500, Aug 2014.

[6.7] S. Zander, G. Heppner, G. Neugschwandtner, R. Awad, M. Essinger, and N. Ahmed, “A Model-Driven Engineering Approach for ROS using Ontological Semantics,” *CoRR*, vol. abs/1601.03998, 2016.

[6.8] Y. Hua, S. Zander, M. Bordignon, and B. Hein, "From AutomationML to ROS: A model-driven approach for software engineering of industrial robotics using ontological reasoning," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, Sept 2016.

[6.9] X. Li, W. Wang, Z. Shu, N. Zhu, H. He, X. Li, and T. Liao, "A system-of-systems architecture-driven modeling method for combat system effectiveness simulation," in *2016 IEEE International Symposium on Systems Engineering (ISSE)*, pp. 1–7, Oct 2016.

[6.10] B. Trask, A. Roman, D. Paniscotti, and V. Bhanot, "Using model-driven engineering to complement software product line engineering in developing software defined radio components and applications," in *10th International Software Product Line Conference (SPLC'06)*, pp. 9 pp.–202, 2006.

[6.11] L. J. Mohapi, S. Winberg, and M. Inggs, "A domain-specific language to facilitate software defined radio parallel executable patterns deployment on heterogeneous architectures," in *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8, Dec 2014.

[6.12] C. Liao, P.-H. Lin, D. J. Quinlan, Y. Zhao, and X. Shen, "Enhancing Domain Specific Language Implementations Through Ontology," in *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '15, pp. 3:1–3:9, ACM, 2015.

[6.13] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[6.14] D. Rubel, J. Wren, and E. Clayberg, *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley Professional, 2011.

[6.15] J.-P. Tolvanen and S. Kelly, "MetaEdit+: Defining and Using Integrated Domain-specific Modeling Languages," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pp. 819–820, ACM, 2009.

[6.16] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[6.17] J. Davis, "GME: The Generic Modeling Environment," in *Companion of*

the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '03, (New York, NY, USA), pp. 82–83, ACM, 2003.

[6.18] J. Sztipanovits, T. Bapty, S. Neema, L. Howard, and E. Jackson, *OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems*, pp. 235–248. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014.

[6.19] J. Marino and M. Rowley, *Understanding SCA (Service Component Architecture)*. Pearson Education, 2009.

[6.20] O. Shacham, S. Galal, S. Sankaranarayanan, M. Wachs, J. Brunhaver, A. Vassiliev, M. Horowitz, A. Danowitz, W. Qadeer, and S. Richardson, "Avoiding game over: Bringing design to the next level," in *DAC Design Automation Conference 2012*, pp. 623–629, June 2012.

"Finally, in conclusion, let me say just this."
- Peter Sellers

7

Conclusion and Future Work

Virtualization technology, although becoming a game-changer in the embedded space, on its own, cannot fully and simultaneously guarantee the strict requirements that next-generation of intelligent devices are demanding. In this sense, this thesis proposed a scalable secure TrustZone-assisted virtualization solution which addresses security without risking real-time and safety.

In this Chapter, I describe the main conclusions drawn from the development of this thesis. I enumerate the novel contributions achieved with the development of this work, and explain the major identified limitations. Finally, I describe the main possible directions for future work.

This Chapter is organized as follows: Section 7.1 presents a summary of the main conclusions. Section 7.2 enumerates the fundamental novel contributions achieved with the development of this thesis. Then, Section 7.3 identifies the major limitations of the developed work. Finally, the Chapter closes pointing directions for future research (Section 7.4).

7.1 Summary and Conclusions

Virtualization technology starts becoming more and more widespread in the embedded space, driven by the possibility of consolidation and safe execution of multiple environments in the same hardware platform. In the beginning of this thesis, I argued, however, that despite becoming a key technology for embedded applications, virtualization, on its own, cannot fully and simultaneously guarantee the strict requirements that next-generation of intelligent devices are demanding: performance, real-time, safety and security, while simultaneously containing size, weight, power and cost. My argument was that virtualization need to be extended with security-oriented technologies, which promotes hardware as the initial root of trust. While billions of smart, connected devices are proliferating in our key infrastructures, this high level of interconnectivity have raised several security concerns, and proven security is gaining attention as a vulnerability that can also affect safety. The question that I asked, and that has served as motivation for the work, was: *How to engineer a scalable virtualization- and TrustZone-assisted TCB which scales up security from low- to high-end processors without risking safety and real-time properties required by different IoT endpoint devices?*

In this thesis I presented a novel security and safety architecture for virtualized systems by evaluating TrustZone technology as a key enabler for scalable and secure virtualization, developing a fully-featured virtualization environment providing hardware isolation, investigating which "hard entities" can extend virtualization to guarantee the security requirements, and simplifying system configurability and integration through a design ecosystem supported by a domain-specific language.

Firstly, to answers to a plethora of doubts and questions regarding the applicability of TrustZone technology for virtualization, I proposed the development of LTZVisor. With LTZVisor, I demonstrated how the security-oriented hardware enhancements introduced by TrustZone can be adequately exploited to assist virtualization, especially in the case of dual virtual machine solutions. I describe all the details behind the implementation, highlighting its benefits and discussing limitations. I conducted an extensive set of experiments which demonstrated that this technology can effectively satisfy the strict requirements for virtualizing a real-time environment, while offering a low performance cost on running an unmodified guest GPOS.

Secondly, based on the main identified limitations of LTZVisor and motivated by the fact researchers still arguing TrustZone is an ill-guided virtualization mechanism, I

proposed the development of TZVisor. With TZVisor, I demonstrated how to implement a fully-featured TrustZone-assisted hypervisor that supports the execution of an arbitrary number of guest OSes. I explained how is it possible to multiplex more than one guest OS inside the non-secure world side, by adequately handling shared hardware resources. Presented use cases in the aerospace and industrial contexts demonstrated the versatility of the hypervisor in fitting different application domains. The conducted experiments proved the viability of running multiple unmodified guest OSes on the non-secure world side with an acceptable performance cost.

Thirdly, despite TZVisor relying on TrustZone security extension for implementing virtualization, the solution does not address security. It relies on a real-time and safety synergy, but there are no security guarantees beyond the hardware-enforced isolation. To achieve a complete and fully security, safety and real-time synergy, I proposed the development of Trusted TZVisor. With T-TZVisor, I demonstrated how TrustZone technology, altogether with other hardware trust anchors, can be adequately exploited to implement a secure virtualization solution which addresses security, without risking real-time and safety. Security starts by assuring a root of trust as the basis for a secure boot process, and continues by establishing a chain of trust which validates, at boot time, all levels of secure software running on the device. Runtime security is guaranteed by implementing the main building blocks of a trusted execution environment as a lower-priority thread of the secure world RTOS. The conducted evaluation process demonstrated how security is achieved while the system's real-time properties remain nearly intact.

Finally, in the process of engineering a scalable virtualization solution from low- to high-end processors, it was hard to manage all complexity and design options for tuning the final system accordingly to the application and target platform needs. To achieve such high-level configurability, I proposed the development of a domain-specific language. With the development of EL and vEL, I demonstrated how DSL technology can help automating the generation of a customizable virtualization-assisted TCB, ensuring correctness by design while boosting the development time. EL represents the generic elaboration DSL, while vEL is seen as an entailment of EL DSL to vertically assist in the hypervisor design.

7.2 Contributions

This thesis advances the state-of-the-art in several directions. The following list describes the most relevant contributions:

- ***LTZVisor as a tool to understand, evaluate, and encourage research towards TrustZone-assisted virtualization***. While existent state-of-the-art TrustZone-assisted solutions still lack in providing detailed information about their implementation and deployment on physical platforms, as well as in performing extensive experiments and presenting convincing results, LTZVisor give answers to a plethora of doubts and questions regarding the applicability of this technology for virtualization, specially for the real-time domain. I plan to make LTZVisor available for the open-source community in the near future, encouraging research, whilst providing the foundation to drive the next generation of TrustZone-assisted virtualization solutions;

- ***TZVisor as a fully-featured TrustZone-assisted hypervisor which allows the execution of multiple guest OSes***. While existent TrustZone-assisted virtualization solutions mainly rely on a dual-OS configuration, TZVisor clearly demonstrates how is it possible to multiplex more than one guest OS inside the non-secure world side. The main reason for researchers considering TrustZone as an ill-guided virtualization mechanism was completely refuted, and to the best of my knowledge TZVisor is the unique TrustZone-assisted hypervisor which supports a complete hardware isolation among the multiple supported guest OSes;

- ***T-TZVisor as a scalable secure TrustZone-assisted virtualization solution which addresses security, without risking real-time and safety***. While existent TEEs mainly focus on protecting digital information in the information technology context, T-TZVisor clearly demonstrates how TrustZone technology, supported by other hardware trust anchors, can be adequately exploited to implement a secure virtualization solution which implements the basic building blocks of a trusted execution environment as a lower-priority thread of the secure world side. To the best of my knowledge, T-TZVisor is the unique secure TrustZone-assisted hypervisor which presents such scalability. T-TZVisor just relies on TrustZone technology to both address virtualization and security;

- ***A domain-specific language for easing design, integration and configuration of virtualization-assisted solutions***. While the major existent virtualization-assisted TCBs are very specific and just provide customization at low-level, EL and vEL demonstrated how DSL technology can help automating the generation and easy customization of the virtualized system according to the application and target platform needs. To the best of my knowledge there is not existent initiative that exploits a service-oriented programming model to provide such high-level design and customization of a virtualized environment.

7.3 Limitations

Despite the several advances and novel contributions for the state-of-the-art, this work still presents some limitations. The following list identifies the major limitations that I hope they can be overcome in the near future:

- All developed versions of the hypervisor (LTZVisor, TZVisor and T-TZVisor), despite being deployed on a multicore Zynq platform, just support a single-core configuration. While running LTZVisor for a single-core configuration can be acceptable, when scaling for a complex system composed of T-TZVisor, several general-purpose guest OSes and a real-time environment, the required complexity and high integration can lead to serious problems of starvation of the non-secure world side;

- Timekeeping issues in a virtualized environment is a well-known and open problem among the virtualization community. Despite TZVisor successfully implemented an effective time management support for Linux and RODOS (with the real notion of the passage of the time), it was possible only due to the tickless nature of both OSes. Current implementation does not provide guarantees of real notion of time for tick-driven OSes such as FreeRTOS;

- The implemented device virtualization approach does not implement a shared device access mechanism. Despite TZVisor successfully implemented device virtualization following a paravirtualization, where devices assigned to the non-secure guest OSes are handled under supervision of the hypervisor, there is no way to share them among the different guests. In a complex system, where multiple guest environments need to coexist, locking devices to just one

VM can be seen as somewhat limitative;

- The main building blocks of a TEE were implemented following the ARM TrustZone API specification. Although the implemented approach followed a specification that can provide interoperability and standardization at some extent, the TrustZone API support is not completely implemented and is somewhat outdated. For these reasons, and also from a commercialization perspective, it will be more advantageous to implement both GlobalPlatform TEE Client and GlobalPlatform TEE Internal specifications, because they will guarantee a higher level of interoperability and standardization for the developed system;
- Although system security, as it is currently implemented, is addressed from the instant the system is powered and continues during the runtime, security is much more than the addition of protection mechanisms into the devices. In this sense, the developed system still fails, as it did not follow a complete secure development process, where all software components should be designed taking into consideration the Principles of High Assurance Software. Although such an approach starts being already implemented, in the presented and described artefacts it was not taken into consideration.

7.4 Research Roadmap

I consider that several novel contributions achieved with the development of this thesis was relevant, but, in my humble opinion, I think more important was the several doors that this work opened for future research. I believe this thesis opened the possibility of researching in forth main important directions:

- ***The exploration of multicore and heterogeneous platforms -*** The shift to multicore platforms will help to solve the problems of starvation while providing a better power-performance tradeoff. Nevertheless, among existent multicore approaches (asymmetric multiprocessing, symmetric multiprocessing or hybrid), there are several challenges around the inherent problem of concurrent access to shared (kernel) resources. I have already implemented support on LTZVisor for an AMP configuration, but when scaling the number of guests across the number of cores, several questions regarding spatial isolation arises. I am not aware of any work that address this problem. Regarding

heterogeneous platforms, it will also be very interesting to evaluate the effects for dedicating guests with affinity to specific cores (for example, a RTOS to a Cortex-R core) and evaluate architectural benefits which advent with such design decision;

- ***The shift to resource-constrained devices* -** ARM recently announced the addition of TrustZone technology in the new generation of Cortex-M processors. I believe the effort for shifting the proposed solution to resource-constrained devices can even be split into a virtualization- and security-oriented problem. Regarding virtualization it will very interesting evaluate a real-time TrustZone-assisted hypervisor, implemented for a Cortex-M23 or Cortex-M33 platform, against a real-time hardware-assisted hypervisor, implemented for a Cortex-R52. While the new generation of Cortex-R family processors implement hardware support for real-time virtualization, I believe the implementation of a TrustZone-assisted hypervisor for the Cortex-M family will present a more cost-efficient solution. Regarding security, while GlobalPlatform specification was always design with high-end devices in mind, it will be very interesting to evaluate if it fits well to resource-constrained devices, and, if not, what kind of challenges does it involves. I am pretty sure several questions will arise;

- ***The offloading to hardware* -** While current solution just rely on software components that exploit the COTS TrustZone technology, I believe the offloading of some hypervisor components to hardware will bring several benefits. As RTOS presents several advantages in terms of performance and determinism when specific components such as the scheduler and task manager are offloaded to hardware, the same concept can be applied to the virtualization domain. Real-time virtualization is very demanding, and if aided by specific offloaded hardware components can make the task much easier from a software perspective. Furthermore, regarding security, the implementation of additional hardware accelerators and security monitor modules can provide an higher level of security while providing the flexibility to easily update when a mechanism proved to be inefficient for a certain class of attacks. Liquid silicon can even be used to implement several mechanisms to prevent and protect against side-channel attacks;

- ***The secure development process* -** Security is much more than the addition of protection mechanisms into the devices. The unprecedented technological trend for such level of interconnectivity have raised several security con-

cerns. I believe the only way to effectively meet the desired security level is to put process development in place since the early beginning. As previously explained, I have already started the journey for refactoring the hypervisor code to follow an object-oriented approach, while using coding standards, as well as static analysis tools for improving software quality, and finding and eliminating defects and possible breaches into the code. The vertically EL-assisted DSL (vEL) will also be shaped by its own individual domain ontologies and relation ontology, making all virtualized TCB components fully interoperable.