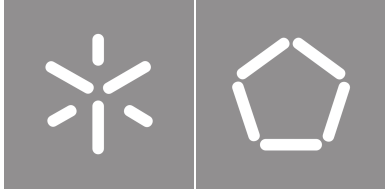
Universidade do Minho
Escola de Engenharia

David Martins Cerdeira

Towards Trustworthy TrustZone-Assisted TEEs

Dezembro de 2024

Towards Trustworthy TrustZone-Assisted TEEs
David Cerdeira
UMinho | 2024



Universidade do Minho
Escola de Engenharia

David Martins Cerdeira

Towards Trustworthy TrustZone-Assisted TEEs

Tese de Doutoramento
Programa Doutoral em Engenharia Eletrónica e Computadores

Trabalho efetuado sob a orientação de:
Investigador Principal Doutor Sandro Pinto
Professor Doutor João Monteiro
Professor Doutor Nuno Santos

Dezembro de 2024

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.0.2) .

Acknowledgements

First and foremost, I want to express my deepest gratitude to my advisors, Sandro Pinto and Nuno Santos, for their unwavering support and guidance throughout this challenging journey. Your dedication and encouragement have been invaluable, and I am truly grateful for your mentorship. Sandro, a special thank you for your long hours of dedication and support. Your superhuman willpower and relentlessness are an inspiration.

I also want to extend my appreciation to all my colleagues at ESRG. Your camaraderie and support have been instrumental in making this journey enjoyable and fulfilling. Whether it was through direct collaboration or simply sharing a few moments in-between breaks, each of you has contributed to my growth and development during my Ph.D.

A special acknowledgment goes to José Martins. Your expertise and extensive technical knowledge have been a constant source of inspiration for me. I have learned a great deal from our interactions, and I am grateful for your friendship.

To my dear parents, and grandmother, I owe a debt of gratitude that words cannot fully express. Your unwavering love, encouragement, and sacrifices have been the foundation of my journey. This achievement is as much yours as it is mine, and I am forever grateful for your endless support.

Lastly, I want to thank my girlfriend, Sónia Miranda, our cat Ada, as well as my friends, for their unwavering support and understanding. Sónia, your patience and encouragement have been my rock during the long hours of research and study. I am truly blessed to have you, and I am grateful for you every day.

Thank you all for being part of this incredible journey.

This work was supported by national funds through Centro ALGORITMI / Universidade do Minho, Instituto Superior Técnico / Universidade de Lisboa, and FCT under project UIDB/50021/2020 and UIDB/00319/2020, by FCT within the RD Units Project Scope UIDB/00319/2020, European Union's Horizon Europe research and innovation program under grant agreement No 101070537, IAPMEI via the SmartRetail project (ref. C6632206063-00466847), CROSSCON project (Cross-platform Open Security Stack for Connected Devices), and by FCT grant SFRH/BD/146231/2019.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

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

_____, _____

(Place) (Date)

_____

(David Martins Cerdeira)

”

“If you aren’t in over your head, how do you know how tall you are?”

— T.S. Eliot

Resumo

Segurança para TEEs Baseados em TrustZone

Os computadores pessoais e dispositivos móveis tornaram-se ubíquos, por isso, garantir a segurança de dados sensíveis tornou-se fundamental. No entanto, com o aumento da complexidade destes sistemas, vulnerabilidades surgem e quebram os seus mecanismos de segurança. Os *Trusted Execution Environments* (TEEs) fornecem um ambiente de execução isolado, e são por isso a tecnologia vigente para proteger os sistemas actuais. Assim sendo, surgiram várias tecnologias TEE nas arquitecturas de processador mais comuns, onde cada arquitectura oferece uma solução de TEE que cumpre com requisitos de segurança específicos ao seu caso de uso.

A tecnologia TrustZone da Arm é amplamente adoptada para implementar TEEs, fornecendo mecanismos necessários à minimização da *Trusted Computing Base* (TCB) e à protecção de dados. É assumido que o *software* destes sistemas é mais seguro do que os Sistemas Operativos (SOs) comuns; no entanto, a divulgação sistemática de vulnerabilidades, afectando bilhões de dispositivos no mundo inteiro, tem levantado dúvidas sobre quais as verdadeiras garantias de segurança que estes sistemas em oferecem.

Esta tese tem como objectivo melhorar a segurança dos sistemas TEE baseados em TrustZone, através: i) da análise e classificação de vulnerabilidades em sistemas baseados em TrustZone; e ii) desenvolvimento de mecanismos e soluções inovadoras que sejam capazes de mitigar eficazmente os principais problemas destes sistemas.

Das contribuições resultantes, destacamos: i) primeiro, uma análise abrangente de vulnerabilidades de segurança em sistemas TrustZone, na qual identificamos três principais grupos de problemas, e onde propomos uma taxonomia detalhada para a sua classificação; ii) segundo, uma abordagem para diminuir os privilégios do mundo seguro da TrustZone, mitigando assim o impacto de vulnerabilidades e melhorando a segurança do sistema; e iii) finalmente, uma *framework* para desenvolvimento de *software-defined* TEEs que suporta a execução de SOs confiáveis em máquinas virtuais, o que fornece uma outra solução para mitigar vulnerabilidades de sistemas TrustZone, e permite resolver questões de interoperabilidade e compatibilidade entre TEEs devido às suas diferenças em termos de funcionalidade e modelos de programação. No geral, as nossas contribuições abordam questões fundamentais sobre a segurança de sistemas TEE baseados em TrustZone, abrindo o caminho para melhorar a segurança destes sistemas.

Palavras-chave: Trusted Execution Environments, TrustZone, Arm

Abstract

Towards Trustworthy TrustZone-Assisted TEEs

As personal computers and mobile devices have become mainstream, ensuring the security of sensitive data has become crucial. However, as systems grow more complex, vulnerabilities often undermine traditional security mechanisms. Trusted Execution Environments (TEEs), which offer an isolated execution environment for applications, have emerged as a central technology for protecting modern computing systems. As a result, we have seen the development of various TEE technologies across mainstream Instruction Set Architectures (ISAs), each tailored to different use cases and offering unique protection models and security features.

Arm TrustZone, in particular, is a widely adopted technology for implementing TEEs, especially on mobile and low-end devices, offering hardware-based security mechanisms to reduce the Trusted Computing Base (TCB) and protect sensitive data and operations. Traditionally, the software in TrustZone-assisted TEEs has been considered more secure than standard operating systems (OSes); however, the frequent discovery of critical software vulnerabilities, affecting billions of devices worldwide, has challenged this assumption, raising doubts about the true effectiveness of these systems in providing robust security.

This dissertation aims to enhance the security of TrustZone-assisted TEE systems, particularly on Commercial Off-The-Shelf (COTS) platforms, by i) analyzing and classifying reported vulnerabilities affecting TrustZone systems, and ii) devising novel mechanisms and solutions that are able to effectively mitigate the major identified issues with TrustZone TEEs.

From the resulting contributions, we highlight: i) first, a comprehensive analysis of TrustZone TEE security vulnerabilities, identifying core architectural, hardware, and implementation issues in TrustZone systems, as well as a taxonomy used to classify them in depth; ii) second, a novel approach and proof-of-concept system that restricts the privileges of TrustZone's secure world, thus mitigating the impact of vulnerabilities and enhancing system security; and iii) a software-defined TEE framework that can host trusted OSes in normal world Virtual Machines (VMs), providing an alternative solution to mitigate vulnerabilities, and overcoming TEE interoperability and compatibility challenges stemming from the heterogeneity of TEE technologies. Overall, our contributions tackle critical security issues in TrustZone systems, paving the way for enhanced security in TEE-enabled systems on COTS platforms.

Keywords: Arm TrustZone, Trusted Execution Environments, Vulnerabilidades de Segurança

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ACU Auxiliary Control Unit
AMBA Advanced Microcontroller Bus Architecture
API Application Programming Interface
ASID Address Space Identifier
ASLR Address Space Layout Randomization
AXI Advanced eXtensible Interface

BTB Branch Target Buffer

CA Client Application
CCA Confidential Computing Architecture
CHERI Capability Hardware Enhanced RISC Instructions
COTS Commercial Off The Shelf
CoVE Confidential VM Extension
CPU Central Processing Unit
CSA Computational Storage Architecure
CVE Common Vulnerability Enumeration

DDR Double Data Rate
DL Deep Learning
DMA Direct Memory Access
DNN Deep Neural Network
DRAM Dynamic Random Access Memory
DRM Digital Rights Management
DVFS Dynamic Voltage and Frequency Scaling

FFA Firmware Framework Architecture
FIQ Fast Interrupt Request
FPGA Field-Programmable Gate Array
fTPM Firmware TPM

GDPR General Data Protection Regulation
GIC Generic Interrupt Controller
GPOS General Purpose Operating System
GPT Granule Page Tables
GPU Graphics Processing Unit

IDAU Implementation Defined Attribution Unit
IO Input/Output
IOMMU Input/Output Memory Management Unit
IoT Internet of Things
IP Intellectual Property
IRQ Interrupt Request
ISA Instruction Set Architecture

MAC Mandatory Access Control
MAC Message Authentication Code
MCS Mixed Criticality System
MCU Microcontroller Unit
ME Management Engine
MMIO Memory Mapped Input Output
MMU Memory Management Unit
MPU Memory Protection Unit
MTT Memory Tracking Tables

NS Non-Secure

OEM Original Equipment Manufacturer
OS Operating System

PA Physical Address
PMP Physical Memory Protection
PPC Platform Partition Controller

PSP Platform Security Processor
PUF Physical Unclonable Function

QNN Quantized Neural Networks

RAM Random Access Memory
RDC Resource Domain Controller
REE Rich Execution Environment
RME Realm Management Extension
ROM Read-Only Memory
RoT Root-of-Trust
RTOS Real-Time Operating System

SAU Security Attribution Unit
SDK Software Development Kit
sdTEE Software-Defined Trusted Execution Environment
SEAM Secure-Arbitration Mode
SEP Secure Enclave Processor
SEV Secure Encrypted Virtualization
SEV-ES Secure Encrypted Virtualization-Encrypted State
SEV-SNP Secure Encrypted Virtualization-Secure Nested Paging
SG Secure Gates
SGX Software Guard Extensions
SLOC Source Lines of Code
SMC Secure Monitor Call
SMM System Management Mode
SMMU System Memory Management Unit
SoC System-on-Chip
SPH Static Partitioning Hypervisors
SPU Secure Processing Unit
SRAM Static Random Access Memory

TA Trusted Application
TCB Trusted Computing Base
TD Trusted Domain
TDX Trusted Domain Extensions
TEE Trusted Execution Environment

TFA Trusted Firmware-A
TLB Translation Lookaside Buffer
TPM Trusted Platform Module
TSM Trusted Security Manager
TZASC TrustZone Address Space Controller
TZMA TrustZone Memory Adapter
TZPC TrustZone Peripheral Controller

UI User Interface

VA Virtual Address
VM Virtual Machine
VMM Virtual Machine Monitor

XPU X Protection Unit
xRDC Extended Resource Domain Controller

Part I

Introduction to TrustZone-Assisted TEEs

1 Introduction

Since the introduction of the personal computer, individuals have progressively entrusted more aspects of their lives (e.g., personal data, banking operations), to computing systems. These systems have become the cornerstone of our lives. They execute software responsible for handling our data, protecting it, and ensuring its correct use, preventing unauthorized data access that might compromise it. However, software is inherently subject to implementation flaws leading to unexpected behaviors, known as "bugs." When attackers can exploit these bugs to thwart the established security measures, they become security vulnerabilities. Thus, it's necessary to mitigate security vulnerability impact on security-critical code.

One approach to mitigate vulnerabilities is dividing the system into trusted and untrusted components. While untrusted components provide non-security critical functionality, the set of the trusted components within the system, i.e., components assumed to work correctly, establish the expected security properties. These trusted components are often referred to as the systems' Trusted Computing Base (TCB) [1]. The TCB typically includes the hardware, firmware, and software components, e.g., the Central Processing Unit (CPU), bootloader, and the Operating System (OS), and tends to expand with the system's complexity. This increase in TCB complexity also increases the system's susceptibility to security vulnerabilities, enabling attackers to compromise devices and access unauthorized data. Given the scale of current systems, which contain millions of lines of code, achieving flawless implementation is virtually unattainable. For example, the Linux kernel v6.9 alone features 25.7 million lines of code[1], with close to 4100 vulnerabilities having been found until now [2]. Another example, the Chromium web browser, an open source web browser on which Google Chrome is based, currently features 40.7 million [3], with close to 1400 vulnerabilities having been found for the Chrome browser [4][2]. Nonetheless, advancements in OS architecture, software development practices, and security-oriented hardware innovations offer means to counteract the risks posed by the potential presence of vulnerabilities. This work explores one of these countermeasures: hardware-assisted Trusted Execution Environments (TEEs).

TEEs offer hardware-based mechanisms for isolated execution and remove the OS, and other privileged components, from the system's TCB, significantly reducing it. TEEs protect the integrity and confidentiality

[1] Measured using David A. Wheeler's 'SLOCCount'

[2] Chromium security bugs are reported as Google Chrome vulnerabilities [5]

of Trusted Applications (TAs) by providing an isolated environment for secure data processing, independent of the OS's integrity. For this reason, TEEs have emerged as a pivotal technology for enhancing the security of sensitive tasks [6–8]. Additionally, they enable the deployment of otherwise commercially unviable services on consumer devices, such as payment systems and DRM-protected content streaming on mobile devices [9, 10]. TEEs may also support isolating general-purpose workloads by offering application or, more recently, Virtual Machine (VM) isolation mechanisms, popular in cloud computing scenarios, i.e., confidential computing [11]. In the cloud, TEE support is growing, with support for TEE mechanisms across all major cloud providers, e.g., Amazon [12], Google [13] and Microsoft [14], and hardware components, e.g., Graphics Processing Units (GPUs) [15]. Due to the critical security properties they provide, it is anticipated that TEEs will see extensive adoption in the Internet of Things (IoT) domain [7, 16, 17].

TEE systems fundamentally depend on hardware support to ensure secure operation. At a minimum, hardware must provide primitives for: i) secure or measured boot, preventing or detecting unauthorized code execution; ii) isolated execution, a separate security domain which untrusted software cannot access; and iii) trusted Input/Output (IO), where the TEE can establish a secure path to interact with security-critical IO devices. Other primitives include remote attestation, monotonic counters, sealed storage, etc. Arm TrustZone is a prime example of TEE enabling technology [7]. Introduced in Arm application processors (Cortex-A) in 2004 [18], TrustZone technology has been adapted for the newer generation of Arm microcontrollers (Cortex-M) to enhance security features in low-end devices [19]. These processors are integral to a wide array of IoT systems, meaning that TrustZone has the potential to bring improved security to billions of Microcontroller Unit (MCU) powered systems. Beyond TrustZone, several other technologies provide TEEs, including Intel SGX [20], AMD SEV [21], Arm Confidential Computing Architecture (CCA) [22], Intel Trusted Domain Extensions (TDX) [23], and RISC-V CoVE [24]. Other approaches such as Apple's Secure Enclave Processor (SEP) [25], Secure Processing Unit (SPU) [26], Intel ME [27], Titan M [28], and Trusted Platform Modules (TPMs) [29] rely on performing critical operations on a co-processor or dedicated chip. Different technologies serve different security requirements and feature different programming models.

Arm's architecture is the most prevalent Instruction Set Architecture (ISA) in mobile devices, and it has also gained significant traction in embedded systems, wearables, and recently in server and PC applications [30]. Unlike traditional semiconductor companies, Arm does not manufacture the chips it designs; instead, it licenses its Intellectual Property (IP) to partners who fabricate and sell these chips. This business model allows for wide adoption of Arm's technology, making it a cornerstone of modern digital electronics. Arm's influence extends through collaborations with numerous technology companies, leading to its architecture being at the heart of a vast array of products. Because Arm is the go-to hardware technology for implementing mobile systems, TrustZone [31] is the *de facto* TEE technology for billions of devices worldwide.

The software of TrustZone-assisted TEEs is assumed to be more secure than standard OSes due to the hardware-based separation enforced by TrustZone technology and a TCB several orders of magnitude smaller. Due to the abundance of OS vulnerabilities [2] and malware on mobile devices [32–35], TrustZone

has become widely adopted as part of the standard mobile device security feature set. For instance, Android platforms incorporate TrustZone-assisted TEEs to secure application-specific operations involving electronic payments [36] or Digital Rights Management (DRM) [37]. Additionally, according to the Android compatibility definition document [38], TEE technology is required to implement some of the system's critical features, e.g., user authentication [39]. Unfortunately, over the years, hundreds of vulnerabilities have subverted TrustZone systems' security guarantees.

1.1 Contributions

The systematic release of vulnerabilities found in TrustZone TEE implementations has been raising doubts about the effectiveness of TrustZone-assisted TEEs in providing robust security. For example, Gal Beniamini has demonstrated, multiple times, how to hack and attack the TrustZone Qualcomm TEE (QSEE). Prominent attacks include the subversion of Android's full-disk encryption mechanism [40] and obtaining arbitrary code execution within a TA [41], which he leveraged to obtain full execution control of the trusted OS [42]. TEE vulnerabilities affect billions of TrustZone-enabled devices worldwide, thus it is critical to devise effective vulnerability mitigation strategies. To address this, our work focuses on investigating how to improve the security of TrustZone TEE systems. Our research centers on the premise that the security of TrustZone-assisted TEE systems can be improved, specifically in Commercial Off The Shelf (COTS) platforms, and our objective is to identify and address the most significant security issues in TrustZone-assisted systems. The following research questions encapsulate the primary scientific challenges of this work:

- RQ1 — What are the causes of security vulnerabilities in TrustZone-assisted TEE systems?
- RQ2 — How to mitigate or minimize TrustZone-assisted vulnerabilities in COTS platforms?

Answering RQ1 requires understanding TrustZone-assisted TEE security issues in detail and performing a comprehensive analysis of vulnerability root causes and existing attack vectors. Answering RQ2 requires the exploration of the insights obtained from answering RQ1, leveraging them as the starting point to investigate how to mitigate the identified issues in COTS platforms. We made three core contributions and additional secondary contributions to the state-of-the-art TrustZone security research to answer both of these questions. Figure 1 gives an overview of how our TrustZone security-related contributions relate to each other, which we discuss next.

1.1.1 Main Contributions

The following three contributions represent the core of our research into improving the security of TrustZone-assisted TEE systems, covering i) the most comprehensive systematization of knowledge regarding TrustZone vulnerabilities to date, ii) a solution to address one of the fundamental TrustZone architectural

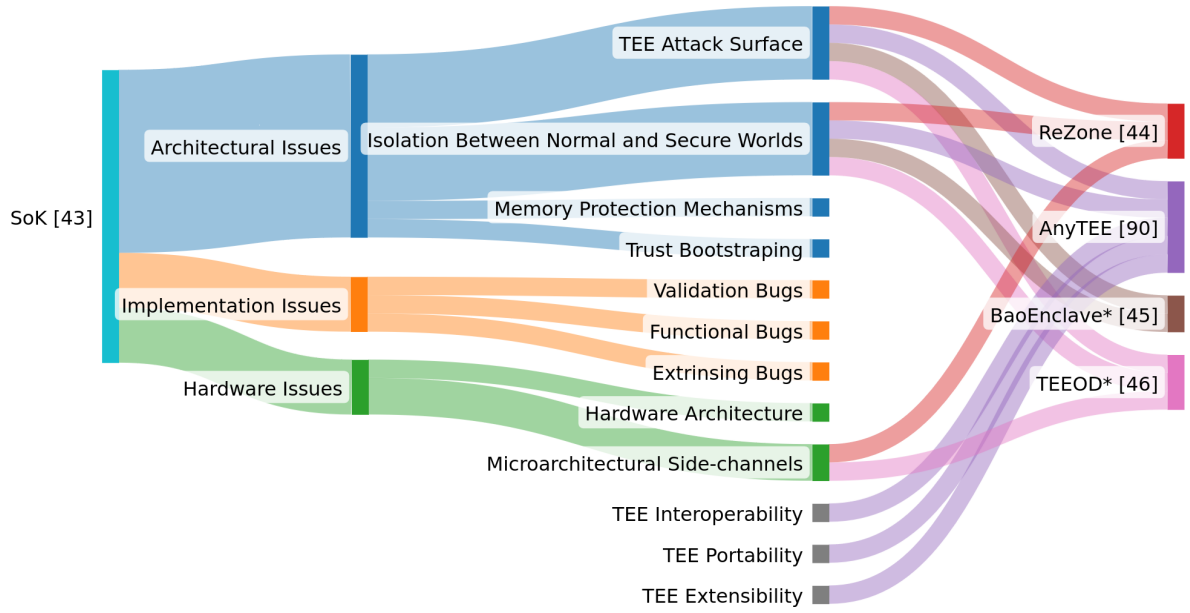


Figure 1: Overview of the relation between dissertation contributions enhancing TrustZone security. * secondary contributions of this thesis.

issues, the excessive privilege of secure world software, and iii) a software-defined TEE framework to emulate and customize TEE systems.

TrustZone TEEs Security Vulnerabilities Systematization of Knowledge. We started our work by performing the most comprehensive and systematic study, to date, of publicly disclosed vulnerabilities affecting commercial TrustZone-assisted TEEs. Despite multiple security reports, scientific papers, and blog posts about vulnerabilities affecting such systems, the information was typically scattered and poorly understood, hampering the overall understanding of the prevailing vulnerabilities and overall security properties of TrustZone TEE systems. We analyzed 207 vulnerability reports related to 5 trusted OSes. We concluded that three main categories of issues exist in TrustZone TEE systems: i) architectural, related to the deployed security mechanisms and TrustZone limitations; ii) hardware, related to how hardware behavior impacts security; iii) and implementation, related to software bugs, such as flawed logic and input validation. Issues are then subdivided into granular categories, as represented in Figure 1. This analysis was published in IEEE Security and Privacy 2020 conference [43] in collaboration with Pedro Fonseca. One of the most fundamental architectural issues identified in our study, and which facilitated the occurrence of dozens of vulnerabilities, was the excess of privileges of the secure world software, which may allow, for example, a compromised TA to automatically take over the Android OS by scanning the physical address space for the Linux kernel and patch it to introduce a backdoor.

Secure World Deprivileging. The fact that the secure world software has full access to the platform's resources motivated our following work, where we studied how to restrict secure world privileges. Our

second contribution proposes a technique that deprivileges the secure world using existing hardware mechanisms in COTS platforms, addressing issues with isolation between normal and secure worlds. Our approach, named ReZone, can successfully restrict secure world memory accesses, which we've demonstrated through a proof-of-concept implementation that reduces trusted OS privileges by only allowing access to specific memory regions. Additionally, it enables multiple trusted OSes to co-exist securely in the secure world, allowing the decomposition of trusted OS stacks into multiple isolated environments, for example, to isolate system functionality TAs from third party TAs. This decomposition addresses the issue of large attack surfaces by preventing a critical vulnerability in one trusted OS from affecting other trusted OSes. Although Armv8.4 introduces S-EL2 i.e., a secure hypervisor, which can restrict trusted OS accesses and host multiple isolated trusted OSes, the secure hypervisor has unlimited privileges to access platform resources, including memory pertaining to monitor and normal world. As a result, an attacker may exploit bugs in the secure hypervisor to obtain access to the whole platform. Thus, although our prototype targets deprivileging the trusted OS in Armv8-A prior to Armv8.4-A, our approach is still relevant in platforms feature >Armv8.4-A CPUs to enforce restrictions on the secure hypervisor. As a side-effect of our approach, which requires sanitizing the microarchitectural state (Translation Lookaside Buffer (TLB) and CPU caches), and preventing concurrent execution with the trusted OS, we also address some of TrustZone's microarchitectural isolation issues. We analyzed how effective ReZone is at mitigating 80 critical security vulnerabilities and estimated that it can thwart 86.84% of potential privilege escalation attacks arising from exploiting these vulnerabilities. The analyzed critical security vulnerabilities were selected from the set of analyzed vulnerabilities in our first work. This work was published in USENIX Security Conference 2022 [44] in collaboration with José Martins.

Software-Defined TEEs. Although ReZone can securely reduce trusted OS privileges, it offers a performance/security trade-off that, while not detrimental to the user experience, may not be acceptable in some scenarios. To address this performance impact, we've decided to run trusted OSes within VMs in the normal world, where a hypervisor emulates TrustZone platform behavior. The insight that virtualization techniques can be leveraged to emulate custom platform behavior led to our third and final core contribution. The extent of this contribution is twofold: i) on one side, we propose executing trusted OSes in normal world VMs, and ii) on the other side we address the fact that system designers face challenges in TEE interoperability and compatibility due to the diverse features and programming models of TEE technologies. For example, while TrustZone mirrors CPU privilege levels, Intel Software Guard Extensions (SGX) preserves the same privilege levels while providing additional protections that prevent arbitrary access to the TEE by system-level software. In practice, these differences translate into separate software development niches for TrustZone and SGX, preventing software reusability between these communities and forcing developers to be proficient in both TEE-enabling technologies. In this work, we propose software-defined TEEs, aiming for interoperability, extensibility, and portability, to allow the coexistence of multiple TEE models and the implementation of custom TEEs tailored to specific needs while ensuring compatibility with legacy stacks. This work counted with the collaboration of José Martins.

1.1.2 Other Contributions

While working on this dissertation, we made other contributions that advanced the existing TrustZone TEE systems body of knowledge. These contributions are discussed next.

Virtualization-based enclaves in Arm systems. TrustZone, despite its widespread use, has faced several attacks that fully compromise the system. Intel SGX does not suffer from this issue because SGX enclaves are unprivileged applications, e.g., if an SGX enclave is compromised no further system privileges can be gained. However, there's no support for this TEE model on Arm processors. We propose baoEnclave, a virtualization-based technology for Armv8-A processors, to bridge this gap. It dynamically creates and secures critical application data by creating enclaves at run time. By providing the mechanisms necessary to instantiate isolated applications, i.e., TAs, in the normal, we contribute to reducing a TrustZone TEE's attack surface. My contribution to this work was leading the system's design, and collaborating in building and evaluating the prototype. This work was published in the World Forum IoT conference 2022 [45] in collaboration with Samuel Pereira, João Sousa, Sandro Pinto, and José Martins.

FPGA-based general-purpose TEEs. Existing TEEs, such as Intel SGX and Arm TrustZone, have been shown vulnerable multiple times, mainly because they shared resources (e.g., memory, devices, the CPU) with untrusted software. We propose a new approach to building TEEs called Trusted Execution Environments On-Demand (TEEOD). TEEOD leverages an System-on-Chip (SoC)'s Field-Programmable Gate Array (FPGA) reconfigurable logic to provide enclaves for security-critical applications. Each enclave includes a dedicated soft processor, private memory, an interface to access security mechanisms, and a communication mechanism to enable Rich Execution Environment (REE) interactions. Although a different approach to BaoEnclave, by enabling security-critical code to execute outside of a TrustZone TEE, this work also contributes to reducing a TrustZone TEE's attack surface. Additionally, by running security-critical code in a physically separate environment, it also addresses some classes of side-channel attacks. My contribution to this work was collaborating on the system's design. This work is under review in the Journal of Computer & Security [46] in collaboration with Sérgio Pereira, Cristiano Rodrigues, and Sandro Pinto.

TrustZone awareness in peripheral devices. In TrustZone systems, peripherals are usually either secure or non-secure, i.e., they can't be used by both secure and non-secure worlds. Our contribution to solving this is the concept of self-secured devices. This concept allows a single device to multiplex its physical interface into two logical interfaces, one secure and the other non-secure. This allows both worlds to use a device without compromising security or performance. We've created a proof-of-concept implementation of this idea leveraging LTZVisor [47], a simple and open-source hypervisor that supports TrustZone. Our findings show that adding this feature to devices adds little extra hardware while enabling secure and non-secure worlds to share the device securely. My contribution to this work was collaborating on the system's design. This work was published in the Journal of Systems Architecture [48] in collaboration with Sandro Pinto, Pedro Machado, Daniel Oliveira, and Tiago Gomes.

Secure near-data processing. Computational Storage Architecure (CSA) are increasingly adopted in the cloud for near-data processing, where the underlying storage devices/servers are equipped with processing units that enable computation offloading near the data. While CSA is a promising high-performance architecture for the cloud, data analytics also presents significant data security and policy compliance (e.g., GDPR [49]) challenges in untrusted cloud environments. Our contribution to solve this is IronSafe. IronSafe is a policy-compliant system designed for CSA in cloud environments, addressing data security and compliance rules like General Data Protection Regulation (GDPR). It leverages Intel SGX and Arm TrustZone, to maintain CSA's performance benefits while ensuring data security in untrusted clouds. A policy compliance monitor attests to query compliance with governing rules, interpreting a declarative language for policy specification. My contribution to this work was collaborating on the design of the TrustZone-based storage device software stack. This work was published in the International Conference on Management of Data (SIGMOD) 2022 [50] in collaboration with Harshavardhan Unnibhavi, Antonio Barbalace, Nuno Santos, and Pramod Bhatotia.

Novel static partitioning system architecture. While Static Partitioning Hypervisorss (SPHs) offer consolidation and compliance with safety standards, they lack flexibility for advanced features without compromising certification and real-time performance. Our work enhances SPHs by decoupling partitioning and virtualization, allowing for the customization of isolated Virtual Machine Monitors (VMMs) for each partition's requirements. This design supports cross-ISA partitioning and seamless integration of platform-level protection mechanisms. My contribution to this work was collaborating on the system's design. This work is currently under review at the USENIX Annual Technical Conference 2024 [51] in collaboration with José Martins, Daniel Oliveira, and Sandro Pinto.

1.2 Document Structure

This document is divided into two parts. In the first part, Chapter 2 provides relevant background. We first define what is a TEE in section 2.1. Then, we provide a system view of the main TrustZone hardware components in section 2.2 and the typical software architecture in section 2.3. Lastly, we overview other common TEE technologies in section 2.4. Chapter 3 explores research on TrustZone and TrustZone-assisted TEE systems across its layers: i) TEE system (e.g., trusted OS), in section 3.1, ii) REE frameworks, in section 3.2, iii) TA, in section 3.3, and iv) other research, in section 3.4. Chapter 4 draws conclusions for this work's contributions and discusses future research directions.

In the second part of this document, we provide a list of all publications, together with the transcripts of this dissertation's main contributions. Chapter 5 consists of our IEEE Security and Privacy 2020 conference paper [43] on the analysis of TrustZone vulnerabilities. Chapter 6 consists of our USENIX Security 2022 conference paper [44], on the use of COTS mechanisms to isolate TrustZone TEEs. Chapter 7 consists of a paper detailing AnyTEE, our software-defined-TEE framework.

2 Background

In this chapter, we provide background information on Arm Trustzone, focusing on Armv8-A TrustZone, in particular, before the release of the Armv8.4-A. First, in section 2.1, we specify what a TEE is. Next, in section 2.2, we provide a system view of TrustZone, including the additions at the hardware level. In section 2.3, we analyze the typical TrustZone software architecture. Then, in section 2.4, we present a brief history of TrustZone architecture evolution over the years. Finally, in section 2.5, we conclude by presenting an overview of TEE technologies available in other common ISAs.

2.1 What are TEEs?

A TEE is an isolated domain within a system. TEEs are designed to protect sensitive operations against unauthorized access or modification, and are used across a wide spectrum of devices from mobile to server platforms, providing security to applications demanding data and execution protection. Prominent examples include mobile banking [36], DRM [9], and key storage [52]. A TEE operates in tandem with an REE. The REE often features a general-purpose OS that leverages most of a platform's hardware capabilities to implement rich functionalities. Figure 2 depicts a high-level view of a system featuring a TEE, including dedicated hardware resources and protected regions in shared memory and storage.

Various TEE technologies exist, differing in the threat models they are designed to resist and the supported programming models. Some create a trusted domain within the CPU, thus relying on sharing resources between REE and TEE, as we will briefly discuss in section 2.5. Others avoid sharing resources with the REE, most notably, i) separate co-processors in the SoC, such as Apple SEP, Qualcomm SPU or AMD Platform Security Processor (PSP), and ii) external security chips, e.g., Intel Management Engine (ME) and TPM. While offering enhanced security, the separate co-processor TEE technologies are not as flexible as TEE technologies that share the CPU and other resources, and may increase costs while offering inferior performance [53]. Key characteristics of a TEE include [54]:

Isolation: TEEs are isolated from the main OS and other software running on the device. This isolation prevents unauthorized access and protects sensitive data and processes.

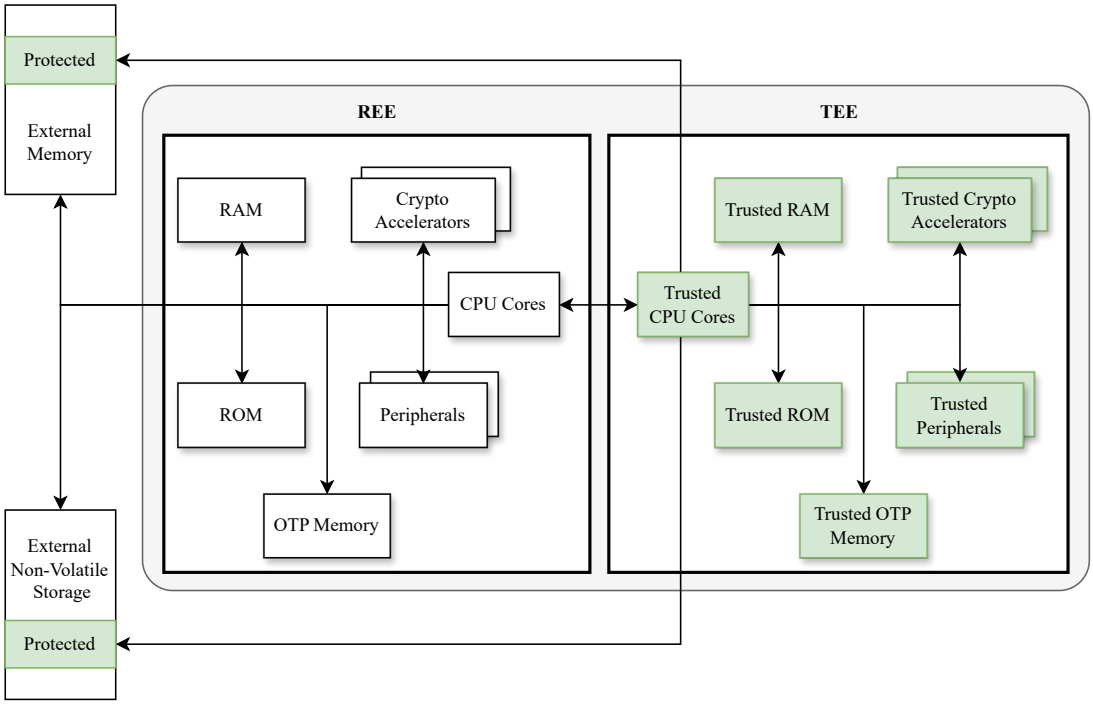


Figure 2: High-level view of a TEE within a system, including trusted and untrusted components and resources, highlighted in green and white, respectively [53].

Verifiable Launch: TEEs often involve a secure boot process, ensuring that the environment starts in a trusted state. Secure boot thus protects the integrity of the TEE and the software that runs within it. Additionally, verifiable launch may also encompass remote attestation mechanisms that prove that the system instantiated an isolated environment correctly.

Trusted IO: TEEs may need to establish a secure path for accessing devices, ensuring confidentiality and integrity, and the protection of TEE data on the device through a trusted device support.

Secure Storage: TEE applications frequently require maintaining persistent states across distinct invocations and system power cycles. This is often achieved through encryption in a process termed "sealing," while the subsequent decryption operation responsible for restoring the state is "unsealing."

2.2 A System View of an SoC

Figure 3 illustrates the architectural and microarchitectural components of a TrustZone-enabled SoC. The main feature of TrustZone is the ability to establish two security domains known as the "secure world" and the "normal world" [7, 55].

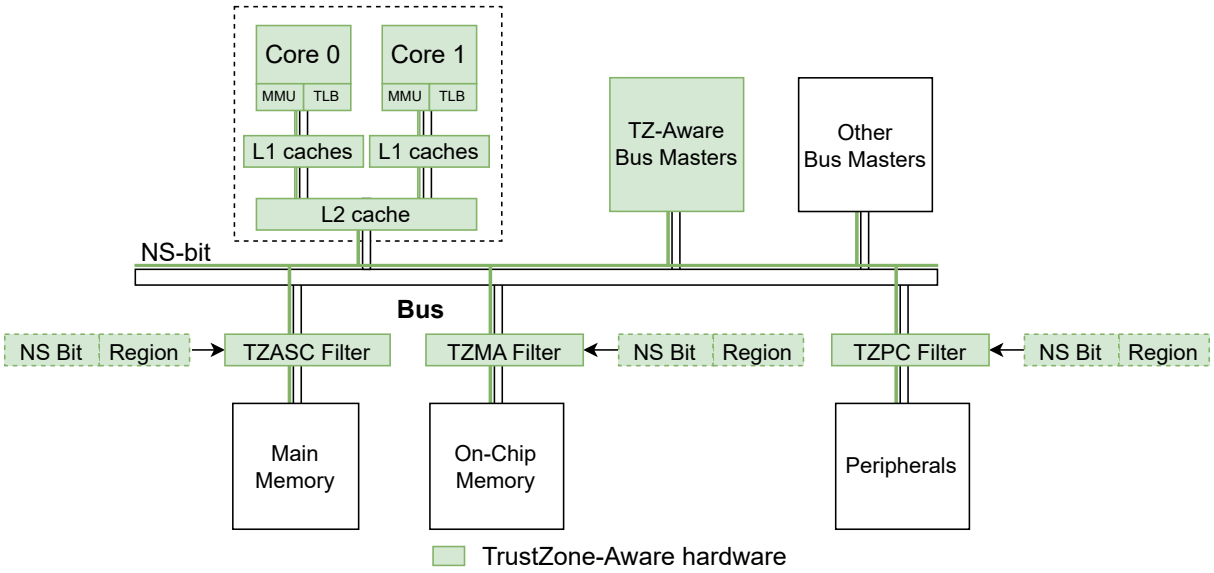


Figure 3: TrustZone-aware architectural and microarchitectural hardware components in an SoC.

2.2.1 CPU Cores

CPU cores are the primary computational units of an SoC, with multi-core CPU designs providing parallel processing to improve system performance [56]. The processor features two security states, i.e., secure and non-secure, that map to the secure and the normal world domains, respectively. The security state is determined by each CPU core's Non-Secure (NS) bit. When the NS bit is set to 1, the CPU core operates in the normal world, whereas when the NS bit is set to 0, it operates in the secure world. In the non-secure state, each CPU core features the following privilege levels: user — N-EL0, OS — N-EL1, and hypervisor — N-EL2. In the secure state, each CPU core features the following privilege levels: user — S-EL0, OS — S-EL1, and secure monitor — EL3. In Armv8-A versions prior to Armv8.4-A, no exception level exists for a secure hypervisor (S-EL2 in >Armv8.4). Please refer to section 2.3 for details on the software component hosted in each exception level.

2.2.2 Microarchitectural Components

The microarchitecture of a system encompasses the major functional units, as well as their interconnection and control [57]. Here we focus on some of the most relevant TrustZone-aware microarchitectural components: L1 and L2 caches, Memory Management Unit (MMU), and TLB. Other microarchitectural components may also be TrustZone-aware, such as the Branch Target Buffer (BTB). A deep exploration of TrustZone implementation across microarchitectural components is out of the scope of this document. Microarchitectural TrustZone awareness enables efficient operation of both secure and normal worlds. Without it, the microarchitectural state would have to be sanitized on each context switch to prevent normal world from accessing secure world resources. However, TrustZone does not protect against microarchitectural side-channels [58, 59].

Caches. L1 caches, i.e., caches at the core level, are the first level of caching in the processor's memory hierarchy [60]. They are divided into two separate caches: one for instructions (L1i) and one for data (L1d). The L1i stores the most recently used or soon-to-be-used instructions, speeding up instruction fetching. Meanwhile, the L1d holds recently accessed or soon-to-be-accessed data, allowing quick data retrieval and storage operations. These caches are relatively small, usually ranging from a few kilobytes to tens of kilobytes, and they are very close to the CPU cores. This proximity to the CPU cores ensures minimal latency in data access.

L2 caches, also known as shared caches (when shared between multiple cores), are the second level of caching in the hierarchy of CPU memory systems, usually sitting between the ultra-fast L1 caches and main memory [60]. Unlike the L1 caches, which serve a single core, L2 caches are shared among cores within a cluster [61]. Additionally, L2 caches can hold both data and instructions simultaneously and are larger than L1 caches, usually ranging from a few hundred kilobytes to a few megabytes. This increased capacity, combined with being faster than main memory, significantly reduces the latency in data retrieval when the requested data or instructions are not found in an L1 cache.

In TrustZone, cache lines are shared between the normal and secure worlds. TrustZone-aware caches add the NS bit as part of the cache line address to avoid unauthorized access. Accesses from the normal world (NS bit = 1) to secure world cache lines (NS bit = 0), i.e., to the same memory address, do not register as hits, thereby protecting secure world data at the cache level. Additionally, when paired with the secure world's ability to perform normal world accesses, this mechanism enables efficient shared memory access, obviating the need for cache flushing. Although sharing caches provides performance benefits, it also leads to the normal and secure worlds competing for the same cache lines. For example, in extreme scenarios normal world may evict all secure world cache lines. This fact has been explored to mount side-channel attacks against trusted software [62].

MMU and TLB. The MMU and TLB provide virtual memory support [56]. Virtual memory enables efficient use of memory and helps isolate processes when managed by the OS, and isolate VMs when managed by the hypervisor. The MMU is a hardware component responsible for translating the Virtual Addresss (VAs) used by software into Physical Addresss (PAs), the actual addresses used to access system resources. The translation of VAs into PAs is possible through the use of page tables. Page tables are in-memory data structures, that, in addition to establishing virtual to-physical memory translation, allow configuration of various memory access attributes, including access permissions. The TLB is a specialized cache for accelerating MMU operation. It stores recent translations, significantly speeding up the address translation process. When performing an access using a VA, the MMU first checks the TLB for a corresponding PA translation. If present, the translation process is much less time-consuming, as it saves the MMU from making comparatively expensive memory accesses to perform a page table walk.

TrustZone awareness for MMU and TLB is twofold. First, there are separate control registers for MMU and TLB management for each privilege level and security state, except for EL0. This means that address space management through virtual memory is independent for the normal and secure world, e.g., N-EL1

and S-EL1, and for each privilege level, e.g., S-EL1 and S-EL3. Second, TrustZone awareness enables efficient secure and non-secure translations. For example, cached TLB entries feature the NS bit of the page table, enabling quick matching of memory accesses.

2.2.3 Interrupts

Arm SoCs feature an interrupt controller, the Generic Interrupt Controller (GIC) [63, 64], that supports external and internal interrupts. Interrupts indicate the need for a program to stop execution and execute the appropriate interrupt handler. When interrupts occur, for example, when an IO device has finished an operation, a signal is asserted on the GIC. The GIC then decides whether to interrupt the CPU, depending on whether other interrupts are being handled, whether other interrupts are pending, or whether higher-priority interrupts occurred simultaneously. If the GIC determines that the CPU should handle the interrupt, it issues a signal that prevents the CPU from continuing its current execution (if the CPU is executing with unmasked interrupts). The CPU will instead execute handler code for the interrupt.

There are several GIC specifications and corresponding implementations. Each specification offers different features, which each GIC implementation can configure. The following is applicable for GICs that implement the GICv3 or v4 [64] specifications. Arm interrupts can be one of two types depending on context and configuration [65], Interrupt Request (IRQ) and Fast Interrupt Request (FIQ), with FIQs having higher priority than IRQs. TrustZone support for the GIC enables each interrupt to be secure or non-secure. This is done by configuring interrupts according to one of three groups: group 0 — secure interrupt, signaled as FIQ; secure group 1 — secure interrupt, signaled as IRQ or FIQ; non-secure group 1 — non-secure interrupt, signaled as IRQ or FIQ. Assigning interrupts to one of these groups is configured by trusted software, by writing to specific GIC registers only accessible to the secure world. The secure monitor typically uses group 0 interrupts, the trusted OS uses secure group 1, and the normal world OS and hypervisor use non-secure group 1 interrupts.

2.2.4 Bus and Bus Masters

The bus is the primary physical communication channel within the SoC, enabling data and control signal exchange between various on-chip components [56], including the ability to access system resources, such as memory and peripherals. Arm uses the Advanced Microcontroller Bus Architecture (AMBA) on-chip interconnect specification for the connection of functional blocks within its SoC designs [66]. Several versions of the AMBA specification exist, with newer versions, e.g., AMBA5 integrating the most advanced protocols for high-performance systems. Among other protocols, AMBA specifies Advanced eXtensible Interface (AXI), a protocol targeting high-performance operation, and the main protocol used in Arm-based SoCs for connecting bus masters with memory and peripherals. A bus master is any device that initiates transactions on the bus, including reading from or writing to system resources. Within the SoC, multiple elements, including CPU cores, Direct Memory Access (DMA) engines, GPUs, etc, may act as bus masters.

TrustZone extends the AXI bus by introducing an additional data transfer attribute, the "NS bit," which accompanies the memory and peripheral accesses, and distinguishes between address spaces for the normal and secure worlds. Specifically, the AXI bus provides access permissions signals, AWPROT and ARPROT, that are used to detect illegal accesses [66]. These signals contain 3 bits, with the second bit signaling the value of the NS bit.

Software running in the normal world is restricted to "non-secure accesses" (NS bit = 1) because the core sets the NS bit to 1 in any transaction initiated by the normal world. Software executing in the secure world typically makes "secure accesses" (NS bit = 0), but it can also perform "non-secure accesses". This is controlled through the NS bit flag in page table entries. Bus masters, other than the CPU, may also be TrustZone-aware. TrustZone-aware bus masters provide the appropriate NS bit information depending on whether they are accessing secure or non-secure resources. One use case that leverages this is a secure data path [67]. In this scenario, data is placed in the normal world encrypted. Trusted hardware accesses the encrypted data in the normal world with non-secure accesses, and decryptes it into a secure world memory region. However, it is uncommon for all bus masters to offer this flexibility. Some bus masters may be fixed to a given security state or may require additional mechanisms to perform both secure and non-secure accesses. In this case, these bus masters are not considered TrustZone-aware [65].

2.2.5 Memory and Memory Mapped IO

The memory subsystem in an SoC encompasses various types of memory. On-chip memory resources are optimized for low-latency access by CPU cores, while interfaces with external memory components, like Double Data Rate (DDR) Random Access Memory (RAM), provide access to large working memory capacity [61]. Additionally, SoCs integrate a diverse array of peripherals to enable interaction with the external environment and implement hardware features. These peripherals encompass a wide range of functions, for example, IO interfaces and communication controllers (e.g., UART, SPI, I2C). The integration of peripherals on the SoC reduces external component count and simplifies system design while enhancing the overall functionality of the SoC.

TrustZone controllers define the normal and secure world physical address spaces, i.e., which resources, memory, and peripherals, are assigned to each world. To do this, TrustZone controllers enforce access control policies for specified memory regions, ensuring that non-secure accesses (NS bit = 1) to secure resources (NS bit = 0) are not permitted. Without TrustZone controllers, secure world isolation is impossible, as evidenced by devices like the Raspberry Pi 3/4 [68], which despite having TrustZone support at the core level, lack platform-level TrustZone controllers. Arm offers three types of TrustZone controller IP, the TrustZone Address Space Controller (TZASC) [69, 70], the TrustZone Peripheral Controller (TZPC) [71], and the TrustZone Memory Adapter (TZMA) [72].

TZASC allows secure world software to control and manage the normal and secure address spaces for main memory, typically DDR RAM. It defines which memory regions are considered normal or secure, and establishes the access permissions for each region. Secure and non-secure regions and access

permissions are typically established during boot, but can also be established during execution. To date, Arm has released two TZASC versions: TZC-380 [69] and TZC-400 [70]. They share similar high-level features, i.e., the ability to configure access permissions for memory regions. The TZC-400 provides additional features, such as per-bus master access control of secure and normal memory regions. Similarly to the TZASC, the TZPC acts as a filter, but grants access to peripherals (e.g., UART, SPI, etc). Additionally, TZPC is used by trusted software to establish the secure memory region size for on-chip memory when paired with TZMA. Whereas TZASC establishes the TrustZone address spaces for main memory, TZMA's purpose is to establish the TrustZone address spaces in on-chip memory, typically Static Random Access Memory (SRAM).

TZASC, TZPC, and TZMA are not part of the TrustZone specification, they are provided as IPs by Arm. Vendors often implement their own TrustZone controllers and may consolidate functionality into a single unit, implementing the security features of TrustZone in a way that suits their specific needs. For example, some of NXP's iMX platforms feature an Extended Resource Domain Controller (xRDC) [73]. The xRDC is a system-wide filtering mechanism that enables the definition of multiple security domains within the system. These domains include a set of memory and Memory Mapped Input Output (MMIO) regions, as well as bus masters. The xRDC completely integrates TrustZone controller functionality, being leveraged by secure world software to establish the TrustZone address spaces. Another example is Qualcomm's X Protection Unit (XPU) [74] which provides a similar mechanism and integrates TrustZone controller functionality.

2.3 Software Architecture

In TrustZone, each physical processor core can execute in either the secure or non-secure state. The secure state hosts TAs, the trusted OS, and the secure monitor, while the non-secure state hosts the hypervisor, OS, and applications, some of which may be Client Applications (CAs). Figure 4 depicts the typical software architecture of a TrustZone-assisted TEE system.

Secure Monitor. The secure monitor operates at EL3, the highest privilege mode. There's no need to distinguish between normal and secure world security states for EL3 because there is no normal world counterpart to this privilege level. Only the secure monitor can control the security state of the processor, thus, it is responsible for performing a context switch between security states. Coupled with the new privilege level, TrustZone provides a new instruction that allows the software to directly invoke the secure monitor, the Secure Monitor Call (SMC). In addition, the secure monitor also implements a firmware layer that performs platform-specific operations, which include, for example, power management capabilities.

Trusted OS. In the secure world, a trusted OS runs in EL1, i.e., S-EL1, and provides runtime support for sustaining the lifecycle of TAs, which run in user mode, EL0, i.e., S-EL0. The trusted OS implements drivers for accessing trusted peripherals, handles cross-world requests, and implements features, such as cryptographic algorithms, trusted IO, and secure storage.

Trusted Application. While the trusted OS provides the core mechanisms, TAs implement the TEE

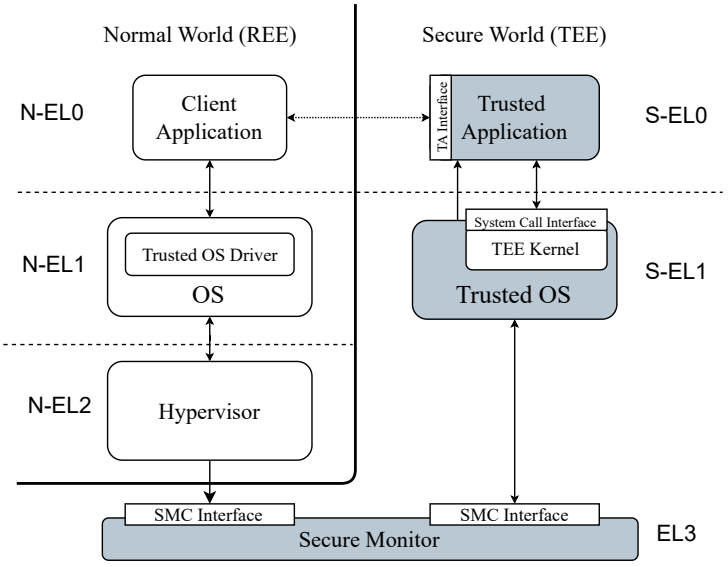


Figure 4: Typical Armv8-A software architecture of a TrustZone-assisted TEE system (pre Armv8.4).

security services. A device manufacturer can implement its own TA for any purpose. TAs often provide system security features, such as authentication, through fingerprint or otherwise, or provide third-party services such as media decryption. TAs execute in S-EL0, with access to system resources managed by the trusted OS.

Hypervisor. The hypervisor resides in the normal world, running in N-EL2. The hypervisor creates and manages VMs, similarly to how OSes manage applications, setting boundaries on resource access and often mediating hardware interactions on behalf of the OSes. It may host multiple VMs, or serve other purposes such as integrity monitoring [75]. The hypervisor can mediate its guest's access to the secure world by interposing on SMCs.

OS. The OS, sometimes referred to as rich OS, resides in the normal world, and offers the diverse range of features expected from a full-featured OS, including multi-tasking, file systems, network support, and graphical interfaces. Thus, the OS is suitable for non-system-critical functions, allocating system resources for diverse computing functions. The OS also provides interfaces that enable CAs to communicate with the trusted OS.

Client Applications. The CAs are unprivileged applications that may interface with the user. CAs are ultimately responsible for requesting services from the trusted OS and TAs to provide security-critical functionality.

2.3.1 Typical Platform Boot

A secure boot process is paramount to the TrustZone system's security. Any unauthorized modifications to the trusted software must be detected during boot, as otherwise, attackers would be able to execute

arbitrary code with the highest privileges. An example of this is the first release of the Nintendo Switch, which launched with vulnerabilities in the secure Read-Only Memory (ROM) code [76].

The bootloader bootstraps the TEE system into a secure state. Bootloader software is often divided into several stages, summarized and illustrated in Figure 5, to accommodate platform restrictions such as boot ROM size. The boot process typically initiates with trusted code executing from an immutable on-SoC ROM. Each subsequent module's authenticity and integrity are verified before execution. The verification process involves digital signatures from the vendor, with the binary image signed using the vendor's private key. The corresponding public key, or its hash, is stored in a one-time programmable memory like eFuses, ensuring the binary's integrity and origin. The bootloader stored in the SoC's ROM has the goal of authenticating and loading the second bootloader, the trusted board boot, from storage ①. Upon successful authentication, the trusted board boot proceeds to load and authenticate additional components, including the secure monitor ②, trusted OS ③, and the normal world bootloader ④. The secure monitor, once initiated, configures platform security settings, such as defining secure and non-secure memory regions. Following this, the trusted OS is launched. After, execution control is given to the normal world software. The trusted OS also verifies the authenticity of TAs during the TA loading process ⑤, which may occur during the system's run time.

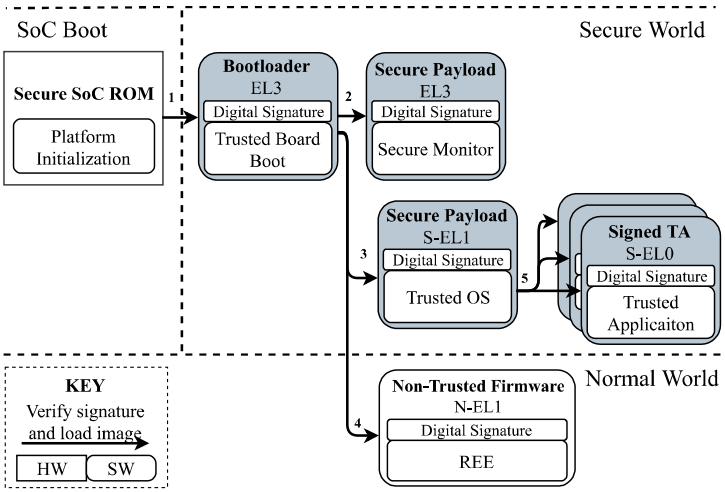


Figure 5: Generic example of a TrustZone secure boot process.

2.3.2 Cross World Requests

The communication between the two worlds is made possible by the TEE driver. The driver implements an interface that allows CAs running in the normal world to connect to specific TAs running in the secure world and allows the trusted OS in the secure world to request services from the normal world.

Requests to the Secure World. In the normal world, access to the TrustZone interface, i.e., SMC, is privileged. The instruction is not available to user space applications, and can only be performed by the kernel. After an SMC is issued by the kernel, i.e., the TEE driver, the Secure Monitor performs a context

switch to the trusted OS. The trusted OS identifies which TA the request is targeting, loads the TA if not already loaded, and executes it. After execution is completed, the results are communicated to the normal world, by reversing the invocation steps.

Requests to the Normal World. The trusted OS occasionally needs services from the normal world, such as loading TAs from the normal world file system or utilizing network sockets. To facilitate this, it interacts with the TEE driver. The TEE driver may leverage a dedicated application, e.g., TEE supplicant in the case of OP-TEE, designed to handle these requests on behalf of the trusted OS.

Trusted OS Scheduling. Two approaches are used to schedule secure world execution [77]. One is direct invocation, where the core that requests the secure world, yields execution control. Depending on the type of request, the trusted OS will execute until a normal world interrupt occurs, at which point it will resume normal world execution, or it will run the operation until completion, ignoring normal world interrupts. The second scheduling approach is interrupt-based. In this approach, secure world requests are stored in message queues that are consumed periodically by the trusted OS. The implemented scheduling approach differs between trusted OSes. For example, while OP-TEE [78] uses direct invocation, Trusty [79] integrates an interrupt-based scheduler [77].

2.4 TrustZone Evolution

Although, from a high-level perspective, TrustZone continues to serve the purpose of protecting TAs, it has suffered minor changes over the years.

TrustZone was introduced in Armv6-A, and its architecture remained unchanged in Armv7-A [31]. Then Arm introduced Armv8.0-A, the version that we described in previous sections, with modifications to the privilege levels. With Armv8.4-A [80], TrustZone incorporated a secure hypervisor, enabling the execution of multiple isolated VMs in the secure world. The advent of Arm's CCA [22], available as the optional Realm Management Extension (RME), in Armv9.2-A, marked Arm's entry into confidential computing, offering confidential VMs for arbitrary code execution. With the RME, Arm addressed the fact that in previous versions secure monitor was not isolated from other trusted software, e.g., trusted OS and trusted hypervisor, by introducing a dedicated world for the secure monitor. TrustZone has also been implemented in microcontrollers, with Armv8-M introducing optional TrustZone support for Cortex-M [81] processors, bringing hardware-based security features to MCUs.

2.4.1 Armv6-A and Armv7-A

TrustZone in Armv6-A and Armv7-A differs from TrustZone in Armv8-A primarily in the privilege levels. In Armv6-A and Armv7-A the trusted OS shares the same address space and privilege level with the secure monitor, whereas TrustZone on Armv8-A introduces an EL3 privilege level, distinct for its highest system privileges and separate virtual address space. Although this new privilege level promotes separation

between secure monitor (EL3) and trusted OS (S-EL1), ultimately, a trusted OS can still access all system resources, including secure monitor code and data. Figure 6 illustrates the software components and privilege levels for Armv6-A and Armv7-A, where the hypervisor (PL2), OS (PL1), and user modes (PL0) are present in the normal world, and only PL1 and PL0 are present in the secure world.

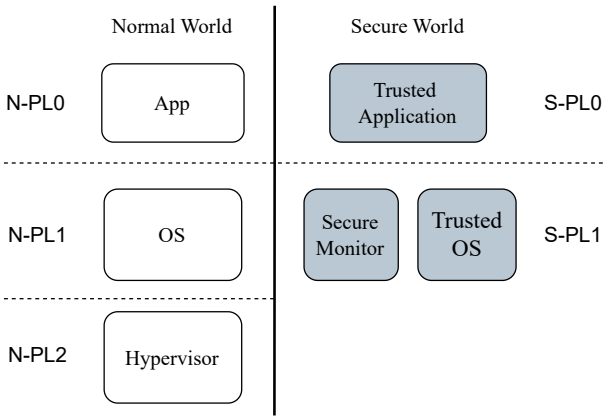


Figure 6: Typical software architecture of an Armv6-A and Armv7-A TrustZone-assisted TEE system.

2.4.2 Secure World Virtualization

Armv8.4-A TrustZone includes the secure hypervisor mode (S-EL2) and thus, the ability to host VMs in the secure world, each isolated from one another and isolated from the normal world. Notably, the secure hypervisor can restrict the address space of trusted OSes, which was not possible in previous versions. However, EL3 is not isolated from S-EL2, with S-EL2 being able to access EL3 memory. The reference secure world hypervisor is Hafnium [82] which does not support dynamic instantiation of VMs, creating them only during boot. Figure 7 illustrates the reference software architecture for Armv8.4-A, with the secure hypervisor running in the secure world.

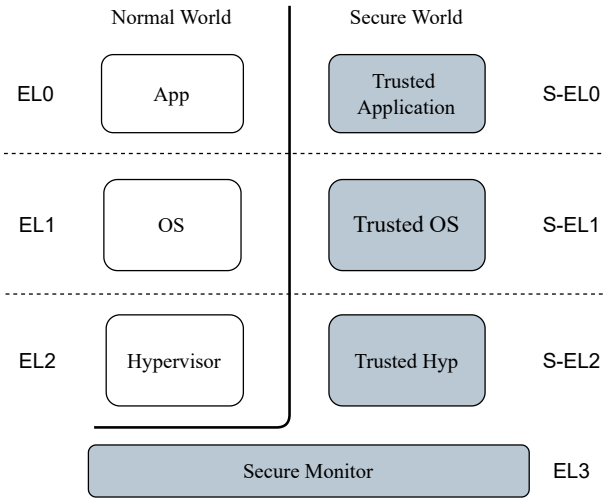


Figure 7: Typical software architecture of an Armv8.4-A TrustZone-assisted TEE system.

The introduction of S-EL2 is paired with Firmware Framework Architecture (FFA). With FFA, TrustZone secure software transitions from a single secure partition model to supporting multiple secure partitions, i.e., secure VMs in Armv8.4-A. FFA standardizes interfaces across software components, e.g., communication between the OS and trusted OSes.

2.4.3 Arm's Confidential Computing Architecture

Arm CCA [22] introduces RME, an optional extension in the Armv9.2-A architecture, represented in Figure 8. Realms are isolated environments for the secure processing of sensitive code and data and feature the necessary properties to implement confidential computing. The RME diverges from previous TrustZone models by establishing two additional worlds: the root world, hosting the secure monitor, and the realm world, hosting the realm monitor and realms. Isolation between worlds is as follows: the root world is inaccessible to others but can access resources from any world; realm and secure worlds cannot access each other's resources but can access normal world resources; and the normal world is universally accessible and constrained to its own resource domain. The root world, i.e., the secure monitor, plays a key role in system security by being responsible for defining resources accessible to each world. The RME replaces TrustZone controllers for a Granule Page Tables (GPT), a page table-like mechanism enforced by the MMU.

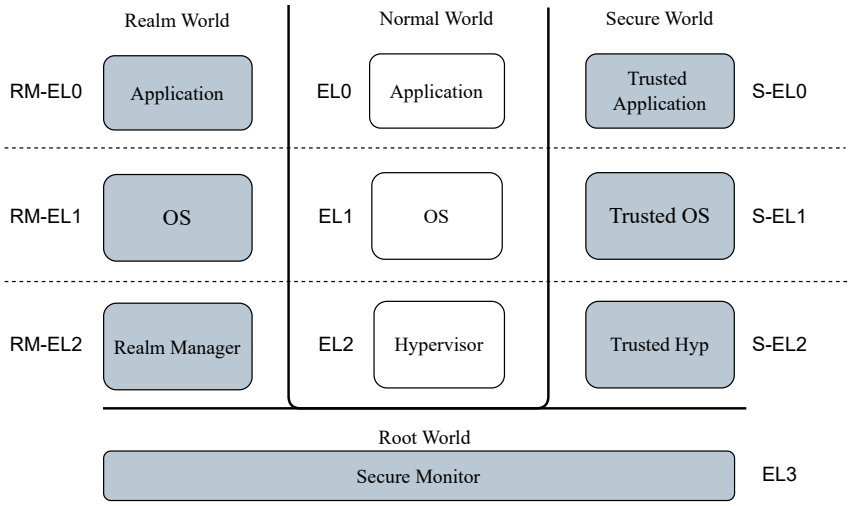


Figure 8: Typical software archtecture for ARMv9.2-A Realm-enabled TrustZone-assisted TEE system.

2.4.4 TrustZone-M

Arm TrustZone for Armv8-M [81], also known as TrustZone-M, is integrated into some Cortex-M microcontrollers. Unlike its TrustZone-A counterpart, TrustZone-M utilizes a memory map approach; the secure state of the processor is determined by whether the code runs from normal or secure memory. Other notable differences include the removal of the secure monitor and the replacement of SMCs for Secure Gates (SG)

to enable context switching between worlds. Figure 9 illustrates the resulting software architecture, with normal and secure versions of the thread and handler modes. Thread and handler modes are loosely equivalent to EL0 and EL1 in Armv8-A. The typical TrustZone controllers are replaced by the Security Attribution Unit (SAU) and Implementation Defined Attribution Unit (IDAU). The SAU is provided as part of the architecture, while the IDAU is external to the core, and its presence and implementation vary between vendors.

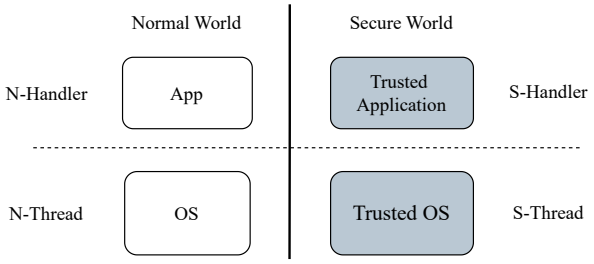


Figure 9: Typical software architecture of a Armv8-M TrustZone-assisted TEE system.

2.5 Other Common Trusted Execution Environments

TEEs have achieved widespread adoption across well-established computing architectures, with prominent CPU manufacturers having integrated mechanisms into their respective platforms to provide TEEs.

2.5.1 Intel

Intel [83] microprocessors are present in a majority of personal computers and cloud infrastructure. Intel integrates security features into its product portfolio, including TEEs. Presently, Intel offers two primary TEE solutions, namely SGX and TDX.

Software Guard Extensions (SGX)

Intel SGX [84], released in 2015, is a set of security instructions integrated into some Intel CPUs, enabling the creation of isolated memory regions, known as “enclaves,” within applications. Figure 10 illustrates the SGX software model, with the enclave being isolated as a part of the application and isolated from the rest of the system. SGX protects enclaves from malicious or compromised OSes and hypervisors, as well as hardware-level threats such as bus-snooping, or cold-boot attacks. Isolation is achieved by using CPU-imposed access controls and a built-in memory encryption engine. The memory encryption engine guarantees the integrity and confidentiality of enclaves’ memory through on-the-fly memory encryption and decryption. Encryption and decryption are performed with a key inaccessible to any software. Additionally, Intel SGX establishes remote attestation as a foundational security measure, enabling external entities to validate the secure instantiation of an enclave.

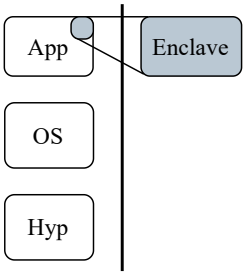


Figure 10: Typical software architecture of an SGX TEE system.

Trust Domain Extensions (TDX)

Intel TDX [85], released in 2021, introduces hardware-based isolation features for VMs within designated Trusted Domains (TDs). Similar to SGX, TDX's primary objective is to protect TDs against high-privileged software, e.g., the hypervisor, while also providing defense against hardware-level attacks like bus-snooping and cold-boot. TDX introduces a new execution mode, denoted as Secure-Arbitration Mode (SEAM). SEAM mode hosts the execution of the TDX Module and TD VMs, ensuring their isolation. The TDX Module functions as monitor software, primarily responsible for defining access control policies, while resource management remains the responsibility of the untrusted hypervisor. Figure 11 illustrates the TDX software model, with the TDX Module executing with the highest privileges and supporting the execution of confidential VMs, i.e., VMs executing in TDs. In contrast to SGX, where a unified key is utilized for all enclaves, TDX adopts a per-TD key model, assigning a distinct encryption key to each TD. Additionally, TDX leverages SGX's remote attestation mechanism, enabling the validation of TDX protection to remote third parties.

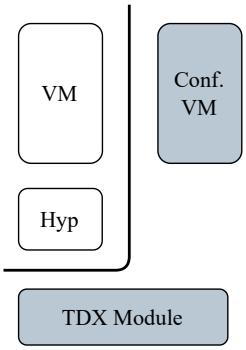


Figure 11: Typical software architecture of a TDX TEE system.

2.5.2 AMD

AMD [86] processors are used in a wide spectrum of computing devices, including personal computers, gaming consoles, and data center servers. AMD's TEE technology protects the execution of VMs from an untrusted hypervisor, while still allowing the hypervisor to manage the system's resources.

AMD Secure Encrypted Virtualization (SEV)

AMD Secure Encrypted Virtualization (SEV) [87], released in 2016, ensures the confidentiality of VM data and code, isolating VMs from potentially compromised hypervisors and physical attacks. Each VM memory is encrypted using a unique key to isolate guests from the hypervisor. AMD PSP, briefly discussed below, manages the necessary cryptographic keys. Figure 12 illustrates the SEV software model. A confidential VMs executes atop an untrusted hypervisor, with the PSP running in a separate processor. Since its first appearance, AMD has extended SEV with improved security features. The first is AMD Secure Encrypted Virtualization-Encrypted State (SEV-ES), an extension of SEV that encrypts all CPU register contents when a VM stops running. This prevents leaking information in CPU registers to the hypervisor, and can even detect malicious modifications to a CPU register state. More recently, AMD developed Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP), another extension of SEV that adds memory integrity protection to help prevent malicious hypervisor-based attacks like data replay, and memory re-mapping.

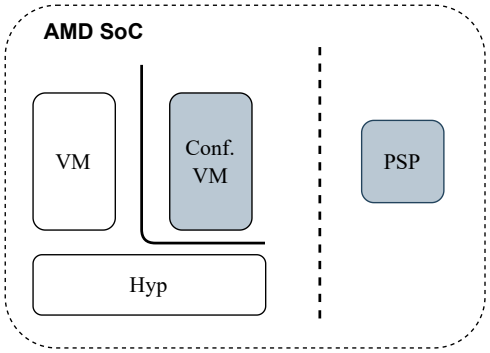


Figure 12: Typical software architecture of an AMD-SEV TEE system.

Platform Security Processor (PSP)

The AMD PSP, integrated into AMD processors since 2013, is an isolated secure processor, present in the SoC, responsible for security-sensitive features such as the generation and management of encryption keys used in SEV. Additionally, it contributes to the secure boot process by authenticating firmware signatures during system initialization. Furthermore, the PSP provides functionality analogous to a Firmware TPM (fTPM), encompassing hardware-based security features such as secure key generation, storage, device authentication, and encryption services.

2.5.3 RISC-V

RISC-V [88] is an open-source, royalty-free ISA for designing computer processors gaining significant traction in recent years. Unlike proprietary ISAs, e.g., Arm and x86, RISC-V is freely available for anyone to use, modify, and implement, allowing companies and developers to innovate and create custom processors without licensing fees. It offers flexibility and customization, making it suitable for various applications, from microcontrollers to data center servers.

Physical Memory Protection (PMP)

The RISC-V architecture features the Physical Memory Protection (PMP) [89], a mechanism designed to perform CPU access control over system resources. Firmware running in machine mode (M-mode), the highest privilege level, can leverage the PMP to establish multiple isolated execution domains, where each domain is restricted in the memory or peripherals it can access, protecting against untrusted software stacks, or creating mutually distrusted domains in the same platform. Figure 12 illustrates the software model, where the firmware manages multiple domains. PMP operates through control registers, which enable the specification of access permissions. Although the number of entries is implementation-specific, the number of PMP entries is limited compared to virtual memory-based solutions.

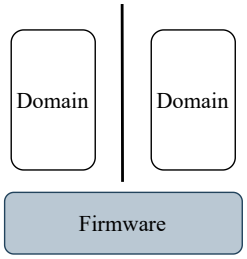


Figure 13: Typical software architecture of a RISC-V PMP TEE system.

Confidential Virtual Machine Extensions (CoVE)

The RISC-V Confidential VM Extension (CoVE) [24] represents RISC-V's response to the confidential computing use case. Analogous to AMD SEV, Intel TDX, and Arm CCA, it allows for the execution of VMs shielded from an untrusted hypervisor, offering protection against hardware attacks when coupled with memory encryption, aligning with other confidential computing solutions. Figure 14 illustrates the software model where the firmware and Trusted Security Manager (TSM) support the execution of confidential VMs. A fundamental component of the CoVE architecture is the introduction of the TSM, operating in hypervisor (HS) mode. The TSM is similar to the TDX Module in that its main responsibility is establishing access control policies on trusted VMs. Notably, when coupled with the Memory Tracking Tables (MTT), CoVE provides fine-grained access control mechanisms similar to Arm's GPT. The MTT, resembling memory page tables, enhances access control by surpassing the limitations of PMP in terms of the number of regions, thereby enabling highly granular and adaptable access control policies.

2.6 Summary

In this chapter, we have provided a comprehensive understanding of Arm TrustZone. We have delved into the hardware support for TrustZone and analyzed a typical software architecture. Additionally, we have described the evolution of TrustZone over time and provided insights into the advancements in TEE technologies on current and upcoming platforms. This sets the stage for the exploration of the state of the

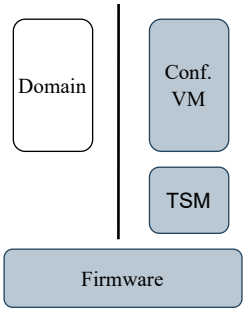


Figure 14: Typical software architecture of a RISC-V CoVE TEE system.

art in the next chapter and provides the necessary background information for the main contributions of this dissertation.

3

State of the Art

In this chapter, we explore research related to TrustZone TEEs. We group the identified works into four categories. The first three categories correspond to the layer of the TEE that the works target. These include TEE systems, REE frameworks, and TAs. The fourth category encompasses additional research efforts that do not fit directly into the previously mentioned classifications. Figure 15, illustrates these layers, and how each of the three categories map to them. We depict privilege levels vertically and in ascending order, with the highest privilege at the bottom. In section 3.1, we provide an overview of existing TEE systems, from commercial to open-source and research-driven implementations. We examine, in section 3.2, how TEEs are used to improve REE systems' security properties. This includes, for example, their role in providing access control or validating the integrity of software components. Then, in section 3.3, we provide an overview of works that implement TAs for various purposes. Lastly, in section 3.4, we address research improving functionality and performance aspects of the TEE system, existing TrustZone system security analysis, and literature reviews. The works featured in this chapter do not represent an exhaustive list but rather highlight areas where TrustZone research has focused on and evolved. We contextualize the major contributions of this PhD Dissertation with the state of the art, in section 3.5.

3.1 TEE Systems

In this section, we highlight research in TEE architecture and implementation. Two of our three main contributions focus on this layer, ReZone [44] and AnyTEE [90]. Our focus in this layer is justified by the high impact that security-critical vulnerabilities in the TEE system have on the whole platform. Existing research has explored various TEE implementations and architectures, showcasing improvements in TEE design. This section focuses on open-source and research-driven TEE systems, TrustZone-based Mixed Criticality Systems (MCSs), formally verified TEEs, and the use of TrustZone as virtualization mechanism.

Commercial TEE Implementation. TrustZone TEE software is often deployed by the SoC vendor, as is the case on mobile phones featuring Qualcomm [91], Huawei [92] and Samsung [93] chips. Alternatively, vendors may use a third party TEE such as SierraTEE [94] and Kinibi [95].

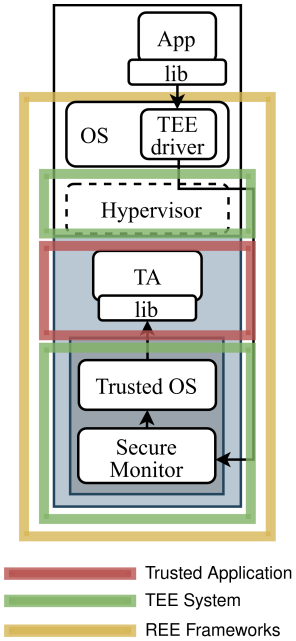


Figure 15: Categories of TrustZone-related research, and how they map to TrustZone-assisted TEE layers.

Open Source and Research TEE Implementations. Over the years, several open-source TrustZone TEE components have emerged. Open-source TEE software naturally benefits from transparency and collaboration among developers to enhance the software's security, providing, in turn, accessible TEE implementations that researchers can leverage in their works, and industry can leverage in their products. OP-TEE [78] and Trusty [79] are notable open-source trusted OS solutions, with OP-TEE supported by the Linaro foundation and Trusty integrated into the Android open-source project. Trusted Firmware-A (TFA) [96] offers a reference implementation for secure world firmware with wide-ranging compatibility. As for academic systems, Yang et al. [97] introduced Trust-E, a trusted embedded operating system architecture utilizing TrustZone to establish a TEE. Paolino et al. [98] demonstrated isolated VMs with TEE access through T-KVM. Open-TEE [99] implemented a virtual, hardware-independent TEE developed in software, adhering to GlobalPlatform specifications, enabling developers to create and debug TAs using standard software development tools. Feske et al. [100] presented Genode, an OS framework that incorporates TrustZone within its TEE infrastructure.

Secure World Application Runtimes. Researchers have provided solutions to enhance security and memory management for TAs and to execute general-purpose applications in the secure world. Komodo [101], TrApps [102], TrustShadow [103], and TLR [8] proposed running non-vendor TAs in isolation from the operating system. Komodo is a formally verified TEE which allowed arbitrary TA execution in Armv7-A platforms. TrApps and TrustShadow proposed approach sandboxes and executes existing Linux applications in the secure world. TLR [8] uses a managed code strategy with runtime memory checks and garbage collection to minimize memory errors. Ginseng [104] enhances data security by compile-time register allocation and runtime encryption, supporting existing apps with minimal modifications and avoiding application logic execution in the TEE. CaSE [105], SecTEE [106], and SoftMe [107] encrypt TA

states during writes to main memory, confining operations within the SoC to address TrustZone's absence of memory encryption. CaSE employed cache-locking to use cache as RAM, while SecTEE and SoftMe offered memory encryption using on-chip RAM. RusTEE [108] leveraged the Rust programming language for TA development, benefitting from the language's memory safety to enhance security. WaTZ [109] proposed a runtime for Wasm code execution within TrustZone, featuring a remote attestation mechanism tailored for Wasm applications, filling the gap in TrustZone's attestation capabilities. Rubinov et al.[102] automated the partitioning of Android applications and divided their execution between secure and normal world execution environments.

Secure World Trusted OS Isolation. The use of software-based virtualization techniques to isolate trusted OSes securely from each other has been proposed by several works: TEEv [110], PrOS [111] and Cicero et al. [112]. These solutions implemented a dual-hypervisor scheme that allows the execution of multiple domains in isolation. Leveraging secure virtualization hardware features, instead of software techniques, Twinvisor [113] and virtCCA [114] provided support for instantiating confidential VMs in the secure world.

Normal World Trusted Application Runtimes. Academia has been actively exploring deprivileging TAs by running them in the normal world. TrustICE [115] and Sanctuary [116] proposed the use of TrustZone controllers to ensure TA isolation in the normal world. TrustICE dynamically protects inactive TAs by modifying the TZASC settings at runtime. Sanctuary uses the TZASC-400 paired with non-COTS hardware modifications to allow restricted access to a TA executing in the normal world. OSP and PrivateZone [117, 118] also suggested running TAs in the normal world. However, instead of TrustZone controllers, they leverage virtualization techniques for effective isolation from the rich OS. Hua et al. [119] introduced TZ-Container, a solution leveraging TrustZone to control the OS memory mappings to securely host TAs running on top of an untrusted OS.

Normal World OS Isolation. Another avenue of research is leveraging TrustZone to create multiple OS environments in the normal world. HA-VMSI [120] applies TrustZone-assisted VM isolation for protecting guest VMs during runtime, even in scenarios where the hypervisor is compromised. vTZ [121] establishes per-VM TEE VM using a hypervisor tightly controlled by the secure world, effectively isolating multiple trusted and untrusted OSes in the normal world.

Virtualization and Mixed Criticality Systems. There's a growing concern with enabling secure and efficient simultaneous operation of Real-Time Operating System (RTOS) and General Purpose Operating System (GPOS). This development is crucial for improving the security of IoT edge devices and fulfilling the stringent real-time demands of industrial IoT applications [122]. Cereia [123] pioneered the implementation of an asymmetric virtualization layer via TrustZone, enabling simultaneous RTOS and GPOS execution and showcasing the feasibility and security advantages of multitasking on single-processor platforms, setting the stage for further research into mixed execution environments. Sangorrin et al. [124] also provide an architecture that facilitates the concurrent running of RTOS and GPOS on a single-core processor.

Subsequent studies have yielded lightweight TrustZone-assisted hypervisors like LTZVisor [47], exploring TrustZone as a virtualization mechanism, and μRTZVisor [125], offering a microkernel architecture. Parallel efforts include VOSYSmonitor [126], a multi-core software layer that leverages TrustZone for securely partitioning RTOS and GPOS on ARMv8-A platforms, and TZDKS [127], optimizing mixed-time-sensitive systems with TrustZone technology. Pinto et al. [128] adapt their hypervisor for asymmetric multiprocessing setups, facilitating concurrent RTOS and GPOS operation. RT-TEE [129] and FreeTEE [130] introduce a real-time TEE, addressing the need for real-time processing in security-sensitive contexts.

Formally Verified Systems. Formal verification techniques are necessary to ensure the highest security guarantees. Ma et al. [131] propose a methodology to design TEEs that supports security properties verification. Komodo [101] offers formal models for the Arm CPU hardware and the Komodo monitor software. Additionally, MIPE [132] suggests employing the B Method for automated proof generation in a system that provides a secure memory protection mechanism leveraging TrustZone.

3.2 Frameworks for the REE

In this section, we provide an overview of research that enriches the REE beyond standard TAs. This includes integrating the REE with TEE-provided mechanisms to improve security.

Peripheral Access Control. TrustZone has been used to perform access control of normal world resources. Brasser et al. [133] developed Restricted Spaces, a solution that allows hosts to regulate device use within specified physical areas and enable, for example, peripheral device removal. Kim et al. [134] introduced a secure automotive software platform utilizing TrustZone to limit rich software accesses, ensuring vehicle peripherals interact only with authorized software. PROTC [135] offered a similar mechanism to a drone use case, where a trusted computing module applies access controls to peripherals. Liu et al. [136] propose software abstractions for providing trusted sensors to mobile applications. SeCloak [137] proposed a method for securely toggling device peripherals on and off, protecting against unauthorized component access or tampering. Other works propose control systems not related to peripheral access. Shim et al. [138] introduced SOTPM, a software-based, one-time programmable memory solution. iFlask [139] implemented a flask-based Mandatory Access Control (MAC) utilizing TrustZone, where the access control policy is stored within the TEE and accessed for access control decisions.

Integration with Network Protocols. TrustZone has been used to enhance the security of data communication. MQT-TZ [140] introduced a secure MQTT broker leveraging Arm TrustZone to provide end-to-end security for messaging protocols, addressing the vulnerability of MQTT brokers processing data in clear text. Ahlawat et al. [141] proposed a method for securing VoIP calls on compromised mobile devices.

Network Gateways. Several works have used TrustZone to improve the security of network gateways.

Gupta et al. [142] implemented an edge-computing-based Industrial Gateway to bridge information and operational technologies. Schwarz et al. [143] unveil TrustedGateway, an architecture that segregates essential networking functions from the gateway OS and allows modifying policy modifications only by verified remote administrators.

3.3 Trusted Applications

One of TrustZone's main goals is securely hosting TAs. TAs provide wide-ranging functionality, from system-related security primitives, such as user authentication and key management, to application-related features, such as DRM or electronic payments. In this section, we survey TrustZone research that leverages TAs to solve various security problems across domains.

Root Of Trust. A Root-of-Trust (RoT) is a critical component within the TCB, typically embedded in hardware or firmware. A RoT can be leveraged, for example, to provide secure boot processes, cryptographic key generation and storage, and software measurement [144]. Several works have explored using TrustZone to emulate a RoT. DFCloud [145] introduced software TPMs within the secure world for encryption key management and establishing a key-sharing protocol among verified users to mitigate vulnerabilities in cloud storage services like Dropbox. Zhao et al.[146] improved this concept by integrating on-chip SRAM Physical Unclonable Functions (PUFs) for secure key storage and generating random values. Raj et al.[147] presented fTPM, a TAs implementation of TPM2.0, allowing instantiation of multiple virtual TPMs. Gross et al. [148] subsequently refined fTPM into a hybrid hardware/software architecture to mitigate software security flaws while preserving its adaptability. Xia et al. [149] utilize TrustZone to create a secure communication channel between mobile devices and cloud servers, incorporating a PUFs for key management. In the context of FPGAs, Jyothi et al. [150] introduce FPGA Trust Zone, a methodology designed to detect and isolate anomalies like hardware trojans within the FPGA fabric.

Key management functions. Cryptographic key storage systems are designed to securely store keys while enabling their use without revealing them. Android features one of these systems, the Android KeyStore service [52], which may employ a Keymaster TA [151] for the generation, storage, and management of secret keys. These services often allow for the storage of secrets within the normal world file system, with encryption and decryption processes executed by the Keymaster TA operating in the TEE. For cloud scenarios, TZ-KMS [152] introduces a distributed key management service to the challenges of key management in multi-cloud environments, enhancing secure data utilization across various cloud platforms.

Data Protection. Researchers have leveraged TrustZone TEEs to improve the confidentiality and integrity of data at rest. DroidVault [153] introduced an isolated data protection manager within the TEE for secure file management, thus enabling users to securely store files on their devices. In contrast, Secure Block Device [154] implemented a block abstraction rather than a filesystem, employing Merkle-Tree-based integrity checks and encryption for security. Liao et al. [155] proposed a system for protecting sensitive

data, making use of a decoy key to conceal the existence of secrets, even when attackers have access to the storage medium. In cloud environments, Brenner et al.[156] have improved ZooKeeper [157] by integrating a transparent encryption layer within the TEE. This modification enables the deployment of ZooKeeper on untrusted cloud platforms, ensuring the confidential coordination of distributed applications by segregating applications into trusted and untrusted components. Harshavardhan et al. [50] implement Ironsafe, a policy-compliant system designed for CSA in cloud environments using TrustZone to establish a secure environment in Arm-based storage devices.

User Authentication. TrustZone-assisted TEEs have been used for secure user authentication. Android requires essential authentication services, such as fingerprint recognition [39] and Gatekeeper [158], to utilize TEEs for the secure management of cryptographic keys and authentication processes. TrustOTP [159] introduced a one-time-password system leveraging TrustZone for improved security. Similarly, On-board Credentials (ObC) [160] enhance credential and authentication mechanisms through TrustZone integration. VeriUI [161] addresses credential security with attested login mechanisms to improve user credential security. Additional approaches encompass device-based authentication [162], two-factor authentication [163, 164], and access control measures [165]. DAA-TZ [166] implemented a system that offers device and user anonymity to remote services, while Feng et al. [167] provided a framework for biometric-based continuous user verification. TrustFA [168] uses TrustZone for secure facial authentication, incorporating photo capture and accelerometer data. TrustZone-assisted TEEs have been leveraged not only to authenticate users but also applications. Some solutions [169, 170] verify the authenticity and integrity of mobile applications, ensuring secure and trustworthy application execution. TZ-MRAS [171] presents a mobile remote attestation scheme designed for attestation applications in the normal world.

Payments. Research has into mobile payment security and privacy leveraging TrustZone aims to protect transaction integrity and user data. Hussin et al. [172] present a mobile receipt system designed to secure transactions for both customers and merchants. Pirker et al. [173] explore the implementation of a privacy-friendly payment mechanism. Zheng et al. [174] introduce TrustPAY, a mobile payment framework to protect payment transactions leveraging trusted User Interface (UI) and trusted IO mechanisms.

Databases. Researchers have proposed secure database management systems leveraging TrustZone. Akowuah et al.[175] and Ribeiro et al.[176] demonstrate secure database management to protect database storage and SQL query confidentiality and integrity. Benedito et al. [177] introduce Kevlar-TZ, which provides a secure REST interface and network connectivity within the TrustZone enclave, capitalizing on the secure storage features of TrustZone-enabled systems.

Logging. Research has proposed solutions to provide logging mechanisms leveraging TrustZone to secure data logs against unauthorized access and manipulation. Lee et al.[178] and Mirzamohammadi et al.[179] have developed secure logging frameworks on TrustZone-enabled platforms. Lee et al. introduce POSTER, a method that provides forward and backward secrecy in log generation via Message Authentication Codes (MACs) produced by TAs, facilitating efficient secure logging through inter-process communication without

altering the standard logging system. Mirzamohammadi et al. propose Ditio, a system for auditing sensor activities by securely logging sensor data for subsequent verification against compliance policies.

Blockchain Wallets. Researchers have developed various TrustZone-assisted cryptocurrency wallets. Zhang et al.[180] and Gentilal et al. [181] develop a wallet utilizing TA data sealing for enhanced security. Dai et al.[182] propose a Hyperledger Fabric wallet (TSLWHF) that leverages a TA for verification, ensuring the integrity of transaction verification results through encryption. SBLWT [183] presents a TrustZone-based wallet to protect simplified payment verification processes.

Blockchain Smart Contracts. The use of TrustZone has also been explored in the context of smart contracts. Jian et al. create TSC-VEE, a virtual execution environment for running Solidity smart contracts within TrustZone [184]. Similarly, TZ4Fabric [185] modifies Hyperledger Fabric to execute smart contracts securely within the TEE, without requiring the entire Hyperledger Fabric node to operate inside the TEE.

General Data Processing. The integration of TrustZone and data processing applications has been proposed to enhance security on edge devices and cloud platforms. StreamBox-TZ [186] provided a stream analytics engine, delivering data security and verifiable analytics outcomes with low overhead. Brito et al.[187] introduced Darkroom, a secure image processing service utilizing TrustZone technology for cloud platforms. Ren et al.[188] proposed a data deletion strategy that restricts data access through encryption key usage limitations.

Neural Network Inference. The utilization of TrustZone for confidential execution of proprietary models on untrusted devices has been investigated in several studies. VanNostrand et al. [189] and DarkneTZ [190] developed a framework for partitioning models within a TEE to secure Deep Neural Networks (DNNs). Trusted-DNN [191] introduced an adaptive isolation approach for DNN models in TrustZone, incorporating a dynamic model partitioning technique for versatility across different models and devices. SecDeep [192] presented a low-power Deep Learning (DL) inference framework that utilizes TEEs to protect the confidentiality and integrity of data and models in edge computing scenarios. Islam et al. [193] proposed a system for confidential DL inference using TrustZone to preserve model and data integrity on memory-constrained edge devices. Additionally, Babar et al. [194] introduced a real-time scheduling framework targeting the execution of resource-intensive DNN workloads within TrustZone. Costa et al. propose SecureQNN[195], a framework that leverages TrustZone-M to protect the privacy of Quantized Neural Networkss (QNNs) in MCUs.

Neural Network Training. GradSec [196] mitigates the challenge of needing substantial portions of machine learning models inside the TEE during training. It achieves this by securing only the sensitive layers of a model, which can be selected either statically or dynamically. SecureFL [197] introduced a secure federated learning framework that leverages TEEs to ensure end-to-end security, incorporating partitioning and aggregation methods to enhance efficiency.

Normal World Introspection and Control. Research has expanded the scope of TrustZone technology in securing and managing normal world functions, including integrity and security monitoring at

the application, kernel, and hypervisor levels. TZ-MRAS [171] introduces a mobile remote attestation scheme leveraging integrity monitoring. T2Droid [198] implemented a dynamic analysis tool for Android. TZMon [199] provided a client-side mechanism for protecting mobile games, securing confidentiality and integrity through various security protocols. Shadow-box v2 [200] proposed an integrity monitoring framework that supports monitoring the integrity of executable files. At the OS-level, SPROBES [201] created a mechanism for OS introspection, facilitating monitoring of specific operating system instructions. Additional research [132, 200, 202–205] explored methods to verify normal world kernel integrity. Liu et al. [206] developed an architecture with an active measurement mechanism for proactive attack defense. At the hypervisor level, Sechkova et al. [207] detailed mechanisms for remote VM deployment and attestation that enforce geolocation restrictions.

3.4 Other Research

Additional research efforts have improved functionality, efficiency, and interoperability between TEE systems, from secure user interaction methods to performance optimization solutions. Lastly, we identify works that analyze TrustZone systems and survey existing research in depth.

Trusted IO and UI. Researchers have developed solutions that balance security needs with practical constraints like code footprint and complexity to address the challenge of providing secure IO channels within TEEs. TrustUI [208] demonstrates a method where TEE device drivers utilize the normal world's drivers to minimize the TCB and secure user interfaces. SchrodinText [209] addressed the confidentiality of UI textual content by rendering text in a way that prevents OS access, using Arm's virtualization hardware and TrustZone. Guo et al. [210] introduce "driverlets," minimal drivers created by recording driver/device interactions. This approach introduced the necessary TEE functionality without significantly enlarging the TCB. GR-T [211] adopted a similar tactic with GPU commands replay to reduce GPU software within the TEE, coupled with a recording architecture for mobile and cloud service interactions. Rushmore [212] also utilized small drivers in the secure world, specifically employing an image processing unit for secure image display. Frameworks like TrustOTP [159], TrustDump [213], VeriUI [161], along with others [162, 163, 174], and Android's Protected Confirmation [214], implemented constrained user interfaces with minimal secure world drivers for secure user interaction. AdAttester [215] proposed security mechanisms for mobile advertising, guaranteeing authentic clicks and verifiable displays. TruZ-Droid [216] and its enhancements [217] integrated TEE with mobile OS to protect user inputs and secrets, employing a delegation model to reduce TCB.

Performance Analysis and Improvement. Recent research has focused on enhancing performance measurement, task scheduling, and optimization strategies to address the complexities and demands of TrustZone-assisted computing. Suzaki et al.[218] introduced TS-perf, a compiler-based method to measure TEE performance by accessing hardware timestamp counters in TEEs and REEs. Gottel et al.[219] developed iperfTZ, an open-source tool for detecting network performance bottlenecks in Arm

TrustZone applications, offering insights into network and energy performance trade-offs. Wang et al.[220] established a benchmark to highlight TEE program execution bottlenecks, and a solution that focuses on time and energy consumption through a machine learning model that adjusts CPU frequency based on application characteristics. Wang et al.[221] analyzed TEE encrypted text performance and propose ETS-TEE, an energy-efficient scheduling strategy using deep learning for dynamic TA allocation between local and edge servers based on task complexity. Li et al. [222] proposed a dependency-aware offloading strategy for TrustZone-supported edge clouds to improve coordination between local and cloud resources for better performance and efficiency.

TA Management. Only Original Equipment Manufacturer (OEM) or device vendor-approved TAs are typically allowed to execute in the TEE. Gowrisankar et al. [223] presented GateKeeper to address this. GateKeeper gives operators increased control and flexibility over the installation and updating of TAs, eliminating the need for mobile platform vendors' involvement at every stage.

Standard Interfaces. Standard TEE interfaces provide interoperability and compatibility between the different TEE components, such as between TAs and trusted OSes, regardless of their manufacturers. The GlobalPlatform organization [224] plays a critical role in defining standard Application Programming Interfaces (APIs) for TAs and communication interfaces, enabling interaction between rich OS software and TEE applications. Standardizing interfaces for TEEs is key to achieving interoperability among TAs, rich OSes, and trusted OSes from various manufacturers. SierraTEE [94] and OP-TEE [78], for example, align with these standards, enhancing the development and portability of TAs. Arm's FFA specification [225] standardizes software communication across worlds

TEE SDKs. TEE Software Development Kits (SDKs) provide developers with tools, libraries, and documentation to create, deploy, and manage TAs within TEE environments. By providing portable uniform APIs, SDKs enable developers to interact with TEE functionalities across different platforms and vendors, allowing TAs to run seamlessly on various TEE-enabled devices, and TEE technologies, without modification. The Open Enclave SDK [226] provides a hardware-agnostic library for TEE application development. TeaClave [227] and Veracruz [228] facilitate secure multi-party computation with TrustZone support, ensuring data confidentiality and trusted computation. Other initiatives aim to provide interoperability and compatibility between platforms by offering their own TEE SDKs and APIs. The Trusted Services project [229] delivers a framework for root-of-trust services on A-profile devices, supporting various isolation technologies for system integrators. Pettersen et al. [230] proposed a secure middleware system for IoT devices and cloud servers, emphasizing the need for comprehensive security measures.

TrustZone Security Analysis. A comprehensive examination of TrustZone security has been conducted across various platforms and applications by different researchers. The role of TEE in Samsung KNOX for creating secure containers is analyzed by Atamli et al. [231]. Koutroumpouchos et al. [232] and Khalid [233] categorize TrustZone attacks and vulnerabilities through Common Vulnerability Enumeration (CVE) analysis. Shakevsky et al. [234] and Cooijmans et al. [235] exposed vulnerabilities in Android's key

storage services, while Bush et al. [236] reverse-engineer the TrustedCore's components and examine the TA loader of the TrustedCore platform and reveal multiple design flaws. Liu et al. [237] assess defenses against cache-based attacks on Arm chips. Benhani [238] discusses security challenges in FPGA SoCs as they relate to TrustZone.

Literature Reviews. Various studies have examined TrustZone's hardware and software architecture and implementation challenges. Pinto et al. [7] and Li et al. [239] have perform a deep analysis of current TrustZone state of the art, highlighting existing challenges and shedding light on its implementation limitations.

3.5 This Work's Contributions

This dissertation provides three significant contributions to the field of TrustZone-assisted TEE systems. Here, we contextualize their impact on the state-of-the-art.

TrustZone TEEs Security Vulnerabilities Systematization of Knowledge. We conducted the first and most comprehensive and systematic study of publicly disclosed vulnerabilities affecting commercial TrustZone-assisted TEEs [43], which was absent, to date. This work has inspired further analysis of TrustZone vulnerabilities [232, 233, 240] and has been used as a reference to motivate further TrustZone TEE research [241], and as insight into developing new TEE [242]. As of this writing, the paper counts with 200+ citations, according to google scholar statistics.

Secure World Deprivileging. With ReZone, we address a fundamental architectural limitation of TrustZone: an overprivileged secure world [44]. ReZone restricts secure world privileges by leveraging hardware mechanisms in COTS platforms. Additionally, we partition a monolithic TEE into multiple sandboxed domains, thus supporting the execution of multiple trusted OSes in isolated environments in the secure world. TEE systems like Sanctuary [116] aim to reduce the attack surface of the trusted OS by relocating TAs onto user-level enclaves in the normal world. Other approaches confine the TEE stack inside a secure world sandbox by leveraging same-privilege isolation, e.g., TEEv [110] and PrOS [111], or hardware virtualization, e.g., Hafnium [82]. However, these approaches do not solve the fundamental problem of the excessive privileges granted to S.EL1 (or S.EL2).

Software-Defined TEEs. With AnyTEE, a system that enables the creation of software-defined TEEs, allows for the emulation and creation of multiple TEE programming models. Unlike most existing TEE systems that support only a single programming model, e.g., Komodo [101], HyperEnclave [243], MyTEE [244], or VirtCCA [114], or support multiple models only in a single architecture, e.g., Keystone [245], AnyTEE enables the emulation of various TEE models concurrently, across architectures. This work not only allows for the simultaneous execution of trusted OSes alongside other TEE programming models but also contributes to the TrustZone ecosystem landscape by extending the availability of the TrustZone programming model to platforms beyond the Arm architecture.

3.6 Summary

This chapter has provided a comprehensive overview of research focused on TrustZone and TrustZone-assisted TEE systems. We have provided an overview of developments in each of the TrustZone TEE system layers. Furthermore, we have contextualized the contributions of this dissertation within the broader state of the art, emphasizing their significance in advancing the field.

4

Conclusion And Future Work

In this chapter, we conclude with some final remarks on the major contributions of this Dissertation, in Section 4.1, as well as discuss potential future work based on our security vulnerability systematization of knowledge and the current TEE landscape, in Section 4.2.

4.1 Conclusions

As computing devices become essential parts of our lives, the protection of sensitive data and operations becomes increasingly important. However, software vulnerabilities pose significant risks. TEEs have emerged as a key technology to address these risks. Arm TrustZone, featured in billions of mobile and embedded devices worldwide, is one such technology. However, TrustZone-assisted TEE systems have been vulnerable to attacks that compromise TrustZone's security mechanisms. Prominent attacks include the subversion of Android's full-disk encryption mechanism [40] and gaining high-privilege arbitrary code execution [42]. This dissertation focuses on enhancing the security of these systems to address potential vulnerabilities in TrustZone-assisted TEE systems. We have made three core contributions to the field, which enable us to draw the following conclusions.

First, TrustZone is not a silver bullet for system security. By systematizing the information from over 200 vulnerabilities reported between 2013 and 2018, we identified several issues which we grouped into three major categories according to our proposed taxonomy: architectural, implementation, and hardware. Architectural issues are related to the deployed security mechanisms, such as the use of stack smashing protection or access permission controls, and TrustZone limitations, such as the excessive privilege level of secure world software, which has access to all system resources, including normal world and secure monitor code and data. Hardware issues are concerned with how hardware behavior impacts security, for example, allowing access to the Dynamic Voltage and Frequency Scaling (DVFS) configuration controls, which can enable the normal world to perform software-based fault injection in the secure world [246]. Lastly, implementation issues involve common software bugs, such as flawed logic and input validation that result in buffer overflows or incorrect permission checks.

Second, TrustZone's primary architectural limitation, i.e., the excessive privileges of secure world

software, can be mitigated on COTS platforms. We address this weakness with ReZone, a system that curtails the privileges of secure world software, such as the trusted OS, by repurposing existing isolation mechanisms in COTS platforms along with an Auxiliary Control Unit (ACU). This restriction of privileges also enables the co-existence of multiple trusted OSes in the secure world, a feature we use to split a single trusted OS into multiple ones, thus isolating system TAs from third-party TAs. ReZone offers a balance between performance and security that may not be suitable in some scenarios, as it requires microarchitectural state management and suspending multi-core execution. To tackle this performance impact, we explored a new design space by running trusted OSes within VMs in the normal world, where a hypervisor emulates TrustZone platform behavior. This insight, i.e., that virtualization techniques can emulate (albeit not fully) the behavior of custom platforms, has driven the third and final contribution of this thesis.

Third, TEE interoperability, portability, and extensibility can be propoted by leveraging virtualization techniques. Based on this approach, we developed AnyTEE; in this project, we introduced the concept of software-defined TEEs, an emulation and customization paradigm for TEEs that utilizes extensible primitives and building blocks. The contributions of AnyTEE are twofold: i) we execute trusted OSes in normal world VMs, addressing the same issues as ReZone but with lower performance penalties, and ii) we tackle the challenges that system designers encounter in TEE interoperability and compatibility, due to the diverse features and programming models of TEE technologies. In practice, this enables application developers to utilize TAs implemented for different TEEs and platforms on a single platform and enables the customization of TEE models to better meet the needs of specific applications.

4.2 Future Work

From the comprehensive proposed taxonomy grouping all identified issues affecting TrustZone-assisted TEEs, our primary focus has been on the architectural flaws within TrustZone TEE systems, particularly concerning the attack surface and inadequate isolation between the normal and secure worlds. Although our work primarily addresses these architectural concerns, other aspects of TrustZone TEE system architecture could also be the target of future research. Next, we provide directions of further research to create trustworthy Trustzone-assisted TEE systems.

Architectural Issues. We foresee the growing adoption of hardware-supported capabilities, and its integration with hardware TEE support. Hardware-support for capabilities has recently gained momentum, as evidenced by Arm's experimental Morello processor [247], which supports Capability Hardware Enhanced RISC Instructions (CHERI). Merging TEE technology and capabilities may help solve some of TrustZone-assisted TEEs architectural issues by providing flexible and fine-grained access control mechanisms. This merge not only requires hardware modifications to existing architectures, but also trusted OS or secure hypervisor, capability support. There is already progress in supporting hardware capabilities and TEEs [248], however they only focus on MCU platforms.

Implementation Issues. Implementation issues occur primarily due to the use of low-level languages, which lack features such as memory and thread safety. Thus, future work to address implementation issues may include, for example, using modern languages with built-in safety features, such as Rust. There already is support for TAs written in the Rust programming language in OP-TEE, which demonstrates a trend to incorporate safe languages into TEE systems. However, we believe that their use in developing high-privilege components, e.g., trusted OS, may further improve the security of TrustZone-assisted TEE systems.

Hardware Issues. Hardware-related issues remain a critical area for investigation. We believe that the full extent of the impact of side-channel attacks or rowhammer attacks on TrustZone security has yet to be fully understood. Regarding side channels, an in-depth analysis of how microarchitecture could potentially leak secure world information across its components could yield useful insights. Similarly, regarding Rowhammer attacks, because adjacent rows in a memory bank do not correspond to contiguous memory, we believe there is a possibility that a normal world attacker could target secure world rows beyond those at the secure/normal world borders.

Part II

Publications

List of Publications

During my thesis, our core contributions results in 2 publications in conference proceedings. Additionally, our secondary contributions resulted in 2 publications, with other 2 currently under review.

Publications in this Thesis

- [43] D. Cerdeira et al. “SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems”. In: Proc. of S&P. 2020.
- [44] D. Cerdeira et al. “ReZone: Disarming TrustZone with TEE Privilege Reduction”. In: Proc. of USENIX Security. 2022.
- [90] D. Cerdeira et al. “AnyTEE: An Open and Interoperable Software Defined TEE Framework”. To be submited.

Other Contributions

- [46] S. Pereira et al. “Towards a Trusted Execution Environment Via Reconfigurable FPGA”. In: Under Review in Computer & Security. 2021.
- [48] S. Pinto et al. “Self-secured devices: High performance and secure I/O access in TrustZone-based systems”. In: Journal of Systems Architecture. 2021.
- [50] H. Unnibhavi et al. “Secure and Policy-Compliant Query Processing on Heterogeneous Computational Storage Architectures”. In: Proc. of SIGMOD. 2022.
- [51] J. Martins et al. “Genesys: In the Beginning There Were Static Partitioning Hypervisors... Let There Be Light!”. In: Under review in USENIX ATC. 2024.

5

SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems

Publication Data

D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. In: IEEE Symposium on Security and Privacy. 2020.

SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems

David Cerdeira†, Nuno Santos‡, Pedro Fonseca*, Sandro Pinto†

†Centro Algoritmi, Universidade do Minho

‡INESC-ID / Instituto Superior Técnico, Universidade de Lisboa

* Department of Computer Science, Purdue University

Abstract

Hundreds of millions of mobile devices worldwide rely on TEEs built with Arm TrustZone for the protection of security-critical applications (e.g., DRM) and OS components (e.g., Android keystore). TEEs are often assumed to be highly secure; however, over the past years, TEEs have been successfully attacked multiple times, with highly damaging impact across various platforms. Unfortunately, these attacks have been possible by the presence of security flaws in TEE systems. In this paper, we aim to understand which types of vulnerabilities and limitations affect existing TrustZone-assisted TEE systems, what are the main challenges to build them correctly, and what contributions can be borrowed from the research community to overcome them. To this end, we present a security analysis of popular TrustZone-assisted TEE systems (targeting Cortex-A processors) developed by Qualcomm, Trustonic, Huawei, Nvidia, and Linaro. By studying publicly documented exploits and vulnerabilities as well as by reverse engineering the TEE firmware, we identified several critical vulnerabilities across existing systems which makes it legitimate to raise reasonable concerns about the security of commercial TEE implementations.

5.1 Introduction

Trusted Execution Environments (TEE) are a key security mechanism to protect the integrity and confidentiality of applications. By leveraging dedicated hardware, TEEs enable the execution of security-sensitive applications inside protected domains isolated from the platform's operating system (OS). Arm TrustZone [31] has become the de facto hardware technology to implement TEEs in mobile environments and has been employed in industrial control systems [249], servers [121], and low-end devices [250]. In the future, where trillions of TrustZone-enabled IoT devices are expected worldwide [251], TEEs can provide secure environments for data processing at the edge.

TrustZone-assisted TEEs are generally assumed to be more secure than modern OSes due to the hardware-based separation enforced by TrustZone technology and their smaller Trusted Computing Base (TCB), which is several orders of magnitude smaller than standard OSes'. For this reason, TEEs have become widely adopted for securing mobile devices against malware [32–35, 252]. For instance, Android platforms incorporate TrustZone-assisted TEEs to secure application-specific operations involving, e.g., user

authentication [39], online banking [36], or DRM [37]. Unfortunately, some of these systems have been exploited over the past years, which casts doubt on the real security guarantees that existing commercial TEEs can effectively provide.

In this paper, we perform a systematic study of publicly disclosed vulnerabilities in commercial TrustZone-assisted TEEs for Arm Cortex-A devices. Despite the existence of multiple security reports affecting such systems, this information tends to be scattered and, in certain cases, unverified, which makes it difficult to obtain a comprehensive understanding of the prevailing vulnerabilities and overall security properties of these systems. To fill this gap, we analyzed 207 TEE bug reports spanning a nearly 5 years, from 2013 until mid-2018, focusing on widely deployed TEE systems developed for Arm-based devices by five major vendors: Qualcomm, Trustonic, Huawei, Nvidia, and Linaro. We examined and categorized numerous vulnerabilities, in particular, some of those that have been leveraged to carry out successful attacks. From our analysis, along with the manual inspection of TEE firmware, we have gained multiple insights about the extent and causes of existing vulnerabilities, and about potential solutions to mitigate them.

One first observation is that TEE systems have a long history of *critical implementation bugs*. Numerous bugs have been (and continue to be) found inside TEE applications – named TAs – and inside the trusted kernel responsible for managing the TEE runtime. Many bugs involve classic input validation errors, such as buffer overflows. As shown by multiple attacks, these bugs can be leveraged to hijack Android's Linux kernel or to entirely compromise the TEE kernel of devices featuring TEEs by Qualcomm [42, 253], Trustonic [254, 255], or Huawei [256].

Second, exploiting vulnerable TAs is facilitated by the *numerous architectural deficiencies* of TrustZone-assisted TEE systems. For instance, the memory protection mechanisms commonly found in modern OSes, e.g., Address Space Layout Randomization (ASLR) or page guards, are almost absent or ill-implemented in most analyzed systems. TEE systems also tend to expose a large attack surface, including dangerous TEE kernel system calls that can be invoked by TAs. For example, on Qualcomm's TEE, any TA can map in memory regions of the host OS. As a result, by hijacking a vulnerable TA, e.g., leveraging a buffer overflow, an attacker can easily control Android [253].

Third, *important hardware properties are overlooked* in most TrustZone systems at the architectural and microarchitectural levels, which can compromise the security of the TEE. Some vulnerabilities are caused by unexpected behavior of trusted hardware components due to microarchitectural side-channels (e.g., in caches) [62, 257–260]. Others are caused by components that can be leveraged to exfiltrate sensitive data from TEE-restricted memory, for instance via reconfigurable hardware (FPGAs) embedded into the modern SoCs [261, 262].

Although many of these problems remain difficult to solve for software systems in general, we observe that the defense mechanisms currently implemented in the studied TEEs lag considerably behind the state-of-the-art defenses incorporated into commodity mainstream OSes and proposed by the research community. We argue that, by adopting up-to-date defenses, commercial TrustZone-assisted TEEs could be made significantly more secure and capable of countering many prevailing vulnerabilities. We present

a collection of relevant defense techniques according to their suitability to address specific kinds of issues: architectural, implementation or hardware issues.

In summary, this paper makes the following contributions: (1) presents the first systematic study of known vulnerabilities in widely used TrustZone-assisted TEE systems (Section 5.3); (2) analyzes the main architectural flaws of TEE systems in perspective with modern OSes (Section 5.4); (3) introduces a taxonomy for classifying implementation bugs that are more likely to be used for exploiting TEE systems (Section 5.5); (4) raises awareness of hardware elements that can be leveraged for attacking TEEs (Section 5.6); (5) analyzes the main defenses techniques proposed by the research community (Section 5.7); and (6) puts TrustZone-assisted TEEs in perspective with alternative TEE enabling technologies (Section 5.8).

5.2 Background and Motivation

This section provides context on TrustZone-assisted TEEs. Also, it motivates our study by showing the impact of TEE vulnerabilities on the security of widely-used mobile devices.

5.2.1 Trusted Execution Environment and Arm TrustZone

A TEE provides an isolated environment for secure processing of sensitive data, without the need to rely on the integrity of the OS. TEEs aim at guaranteeing the secure execution of programs, known as TAs or *trustlets*. TEE systems rely on trusted hardware, such as Arm TrustZone [7], which has been supplied on Arm application processors (Cortex-A) since 2004 [18] and it was recently re-engineered for the new generation of Arm microcontrollers (Cortex-M) [19]. Our work focuses primarily on the Cortex-A TrustZone implementation, which is widely used on mobile devices.

TrustZone is centered around the concept of protection domains named *secure world* (SW) and *normal world* (NW). Each physical processor core provides two virtual cores, one considered 'secure' (SW) and the other 'non-secure' (NW), as well as a mechanism to securely switch between them. The state of the system is identified by the NS bit of the processor, which identifies the current executing world. Hardware logic present in the TrustZone-enabled AMBA bus extends the security state of the processor to other system components, ensuring that SW resources cannot be accessed by NW components.

5.2.2 Software Architecture of TrustZone-assisted TEE

The typical software architecture of a TrustZone-assisted TEE runs the untrusted OS inside NW – named Rich Execution Environment (REE) – and the TEE software components run in the SW (see Figure 16). Inside SW, the *trusted OS* runs in supervisor mode (protection ring EL1) and provides runtime support for sustaining the lifecycle of TAs, which run in user mode (protection ring EL0). The core of the trusted OS is the trusted kernel, which provides the basic OS primitives for scheduling and managing TAs. The trusted OS additionally implements device drivers for accessing trusted peripherals, handles cross-world requests

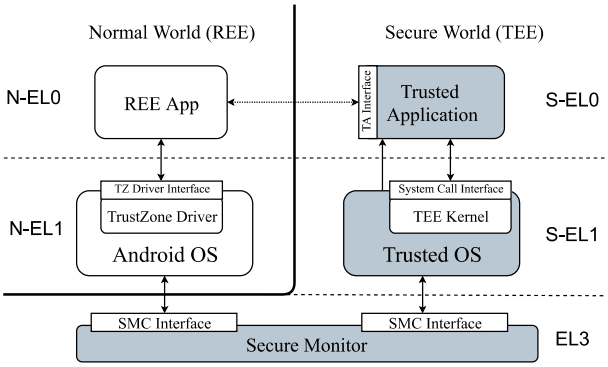


Figure 16: Software architecture of a TrustZone-assisted TEE system.

ID	Ref	Year	Description	Component	Vulnerabilities	Impact
E1	[263]	2015	Input validation weakness can be used as a zero-write primitive anywhere on memory QSEOS's virtual memory to obtain arbitrary code execution in trusted OS. Requires root privileges in Linux kernel.	SW Monitor	[264]	Full control of TZ kernel
E2	[265]	2015	Exploits bug in the TrustZone Linux driver, which allows an attacker to obtain root privileges and thus launch the E1 attack.	NW Driver	CVE-2014-4322	Full control of Linux kernel
E3	[266]	2016	Vulnerability in Android's Mediaserver process which allows an unprivileged REE application to gain access the Qualcomm's TrustZone interface driver. When used with E1 and E2, allows an unprivileged application to obtain trusted OS-level arbitrary execution.	NW Service	CVE-2014-7920, CVE-2014-7921	Full control of Android Mediaserver
E4	[263]	2016	Privilege escalation attack to obtain arbitrary execution in the context of a TA. The vulnerability occurs in the Widevine TA, and can be exploited by accessing the TrustZone interface Linux driver using E3.	SW TA	CVE-2015-6639	Full control of Widevine TA
E5	[42]	2016	Lack of input validation in Qualcomm's trusted OS system calls allows a TA to write to any address within the OS and hijack the TEE kernel. Requires privilege escalation into TA through the TA's interface.	SW Kernel	CVE-2016-2431	Full control of TZ kernel
E6	[253]	2016	An attacker with TA-level execution privileges can gain control of the Linux kernel. This attack can be built upon E4.	NW Kernel	Bad system call	Full control of Linux kernel

Table 1: Representative vulnerability exploits for QSEE, Qualcomm's TEE system, showing the diversity of affected components and security impact.

through the world switching SMC instruction and shared memory, and implements shared libraries (e.g., cryptographic) and TEE primitives, namely remote attestation, trusted I/O, and secure storage.

Beyond the trusted OS, a TEE comprises two fundamental software components. The *secure monitor* implements mechanisms for secure context switching between worlds and runs with highest privilege, in protection ring EL3. The *TEE bootloader* bootstraps the TEE system into a secure state, and it is critical to implement the trusted boot primitive. It is split into two parts which run, first, in EL3, and then in EL1. Together, trusted OS, secure monitor, and TEE bootloader constitute the software TCB of a typical TEE system. For this reason, TEE designers aim for small and bug-free implementations.

5.2.3 Attacking TEE-enabled Devices

Over the past years, critical security vulnerabilities have been identified in TEE systems of widely deployed mobile devices. Some vulnerabilities can be exploited to acquire privileged access to targeted devices and sensitive information stored therein. In this section, we explain how this can be achieved using the set of representative exploits listed in Table 1 to hijack two critical components of a TEE-enabled device: the TEE kernel and the REE kernel (i.e., Linux). Altogether, these exploits demonstrated how to escalate its privileges from a user-level NW application on a platform running Qualcomm's TEE system. Since then, a similar methodology has been successfully employed to attack devices featuring other popular TEE systems.

Compromising the TEE kernel:. Targeting Qualcomm TEE (QSEE), the TEE system developed by Qualcomm, Gal Beniamini showed how to hijack the TEE kernel from an unprivileged user-level NW application in two different ways. One way requires escalating privileges into the Linux kernel (see Figure 16) in several steps. First, use exploit $\mathrm{E3}$ to control Android's Mediaserver, which has privileged access to the TEE driver. Then elevate privileges into the Linux TrustZone driver to access the SMC interface ($\mathrm{E2}$). A third exploit ($\mathrm{E1}$) takes advantage of a bug in the TEE kernel and achieves arbitrary code execution with EL1 privileges in SW. Once in control of the TEE kernel, an attacker can launch other attacks, e.g., hijack a guest TA to extract secret keys and break Android's full disk encryption [40], or unblock the device bootloader [267]. A second way to compromise the TEE kernel only requires access to the interface of a vulnerable TA. Using $\mathrm{E4}$, an attacker can hijack the Widevine TA, a DRM service for Android OS. Then, through a vulnerability in the system call interface, the attacker can further elevate privileges into the TEE kernel ($\mathrm{E5}$).

Compromising the REE kernel:. Additionally, it is possible to compromise Linux without even the need to gain control of the TEE kernel. This can be achieved by using a vulnerable TA as a trampoline for privilege elevation into the Linux kernel. For instance, exploit $\mathrm{E6}$ allows an attacker to take over the Linux kernel by sending crafted input from a user-level NW application into the Widevine TA. A vulnerability in this TA along with QSEE's system calls that allow TAs to map in NW physical memory, enable an attacker to modify memory regions allocated to the Linux kernel and control the system.

The extent of the problem. Several other exploits have been developed for the Qualcomm TEE [255, 268–270]. Beyond mobile devices shipping Qualcomm chips, other platforms have been attacked, namely devices running Trustonic's TEE system, renamed from Mobicore to Kinibi [10, 254, 255, 271], and Huawei's proprietary TEE named Trusted Core [256, 272]. Most of these exploits adopt the divide-and-conquer strategy presented in Table 1. Considering that Trustonic's TEE is estimated to run on 1.7 billion devices (mostly Samsung's) and Huawei's mobile devices are widely adopted (200 million sold in 2018), TEE flaws can have a large impact worldwide.

5.3 Overview

This section provides an overview of our study of security vulnerabilities on commercial TrustZone-assisted TEE systems.

5.3.1 Methodology of our study

Performing a comprehensive security assessment of commercial TEE systems entails several challenges. For many such systems, the source code is not available. Their binaries also tend to be inaccessible or difficult to analyze due to the lack of documentation and the employment of code obfuscation techniques. Additional complexity is caused by the co-existence of legacy TEE software versions by the same vendor and the diversity and heterogeneity of TrustZone hardware. We cope with these challenges by adopting the following methodology.

Adversary model. We consider an attacker that pursues one or more of the following objectives: a) obtain secrets from the TEE, b) obtain secrets from the REE, c) escalate privileges to the REE kernel, or d) escalate privileges to the TEE. He can access the SMC interface exclusively from the NW in two ways: either *directly* by obtaining code execution privileges in supervisor mode (N-EL1), allowing for crafting arbitrary SMC calls, or *indirectly* from unprivileged user-level applications (N-EL0) by issuing commands toward some target TA. All NW components are assumed to be untrusted.

Analyzed TEE Systems. We analyzed TEE systems by Qualcomm, Trustonic, Huawei, Nvidia, and Linaro. Nvidia maintains a proprietary TEE used mostly for Nvidia chips. Linaro maintains OP-TEE, an open source TEE software very popular for TrustZone development. All these systems are actively maintained, are widely adopted for commercial purposes, and a fair amount of information can be obtained about them. We excluded research prototypes (e.g., Andix [249]) or commercial products not currently deployed at scale (e.g., SierraTEE [94]). We also consider relevant cross-cutting vulnerabilities, e.g., hardware side-channels. For the sake of readability, henceforth, we refer to each analyzed TEE by the company name rather than by software name (e.g., Qualcomm TEE means QSEE).

Data sources. We resorted to multiple sources that we grouped into four areas (see Table 2). We analyzed bug reports from the CVE database [273] relative to the TEE systems under study. We retrieved the CVE reports published officially by Qualcomm [274, 275], Nvidia [276] and Huawei [277] which are documented also in their respective security bulletins. We gathered additional CVE reports by searching for relevant keywords, e.g., the TEE names, “TrustZone”, etc. We also collected bug reports from Samsung Vulnerabilities and Exposures (SVE) database [278] which have not been assigned specific CVE IDs. We analyzed scientific publications (SP) in major security conferences from the past 10 years, miscellaneous reports (MR) available online (e.g., [255, 263, 279–282]), and inspected source code (SC) for TEEs’ with public source code, namely Linaro’s OP-TEE. For OP-TEE, we also analyzed its changelog to identify security fixes and interviewed the system designers.

TEE System	CVE	SVE	SP	MR	SC	Total
Qualcomm TEE	92	-	-	7	-	99
Trustonic TEE	5	17	-	4	-	26
Huawei TEE	3	-	-	1	-	4
Nvidia TEE	10	-	-	-	-	10
Linaro TEE	3	-	-	1	36	40
Other	11	-	15	2	-	28
Total	124	17	15	15	36	207

Table 2: Sources of reports: CVE (CVE databases), SVE (SVE databases), SP (scientific publications), MR (miscellaneous reports), and SC (source code).

Classification of disclosed security vulnerabilities. After collecting the vulnerability reports, we manually analyzed and categorized them. For the vulnerabilities assigned with a CVSS score [283], we adopted a classification metric based on the attribute score. Our rating system comprises four categories: *critical* (CVSS $\geq$9), *severe* (CVSS [7,9[), *medium* (CVSS [5,7[), and *low* (CVSS [0,5[). The severity of a specific vulnerability may have different security implications. A critical vulnerability is normally one that can lead to a complete compromise of confidentiality or integrity in the TEE, in the REE, or both.

Binary analysis:. To obtain accurate details about the studied TEE systems, we reverse engineered a subset of them. First, this method allowed us to quantify the size of each system's TCB. Second, it helped determine the specific software architecture of each system, for example, that Huawei uses Arm Trusted Firmware (ATF) as a base for its secure monitor software, while Qualcomm uses its own implementation. Third, it allowed us to analyze the memory protection features implemented by each TEE. For Trustonic TEE we analyzed the firmware for Samsung Galaxy S7 (Exynos) version G930FXXS1APG3, for Qualcomm TEE the Pixel XL firmware version PQ2A.190205.003, and for Huawei TEE the P8-Lite system image ALE-L21C432B603.

Threats to validity:. Since most vulnerabilities have no proof-of-concept exploits or their CVE descriptions may not provide enough detail, our identification and classification of vulnerabilities might have some imprecisions. The lack of information regarding the vulnerabilities existing in proprietary systems may have also led to inaccurate classifications. There is also the risk of over-representation of a given TEE system, particularly in the case that the number of publicly reported vulnerabilities about that system largely outnumbers those of other systems. In such cases, we require extra care while drawing general conclusions. Lastly, we analyzed only vulnerabilities that have been previously reported. As a result, unknown types of vulnerabilities might exist that could reveal additional fundamental security issues in TEE systems.

System	Critical	Severe	Medium	Low	Total
Qualcomm TEE	52	19	12	9	92
Trustonic TEE	1	-	0	4	5
Huawei TEE	-	2	-	1	3
Nvidia TEE	-	5	1	4	10
Linaro TEE	-	-	2	1	3
Other	-	1	7	3	11
TEE Total	53	27	22	22	124
FreeRTOS	-	-	5	8	13
VxWorks	2	2	5	1	10
Linux	242	254	393	758	1647

Table 3: Number of disclosed CVE per system from 2013 to 2018.

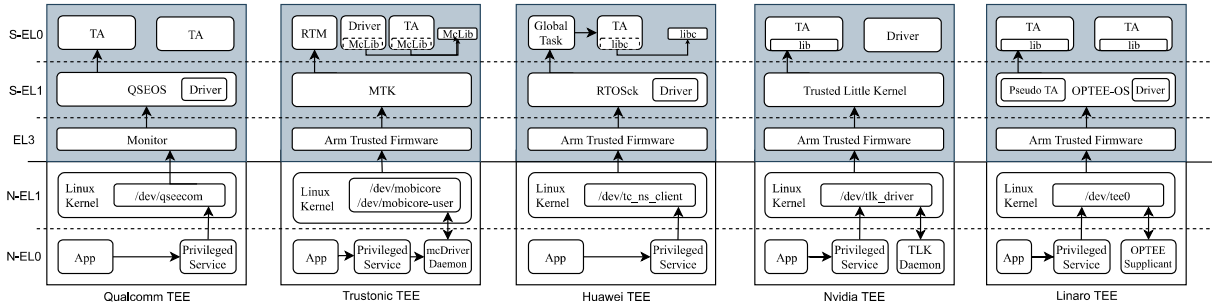


Figure 17: Detailed architecture of the studied TEE systems. A few relevant common features include: (a) the communication between a NW application and the SW is mediated by a privileged OS daemon which uses a TrustZone driver to issue SMC calls to the SW; (b) in four cases, the monitor is based on ATF [284], which consists of the reference implementation provided by Arm for the secure bootloader and monitor software.

5.3.2 Summary of Observations

We analyzed the vulnerability reports of all major commercial TEEs, namely the TEE systems by Qualcomm, Trustonic, Huawei, and Nvidia. Considering the reports obtained from CVE databases, which are classified with a severity score, we manually identified, in total, 124 TEE vulnerabilities during a time window of six years (i.e., 2013 – 2018).

Table 3 quantifies the number of disclosed vulnerabilities associated with each system according to their severity. Almost *half of the bug reports are rated as critical or severe*. In particular, 53 of the 124 reports (42%) disclosed security vulnerabilities are considered critical. Perhaps even more surprising, *every single TEE that we analyzed was found to have at least one non-low severity vulnerability* during the considered time period: Trustonic has 1 critical vulnerability, and Nvidia and Huawei's systems have, respectively, 5 and 2 classified as severe. Considering that collectively these systems are widely deployed, millions of users worldwide may have been seriously affected by these vulnerabilities.

Although it stands out that the Qualcomm TEE accounts for the largest fraction of disclosed vulnerabilities (74%), we caution that we cannot conclude from this data that it is the least secure TEE; or similarly compare individual TEEs. This is due to the disparity in methodology with regards to the

CVE reporting process and could simply be a consequence of higher reporting diligence of Qualcomm developers and users. However, these results are useful because they allow us to establish a lower bound on the vulnerabilities of such systems, reason about aggregate trends, and compare general TEE trends against the trends of other types of systems.

For instance, we observed that during the same time window the entire Linux operating system, which is several orders of magnitude larger than any of these TEEs, only had 1647 CVEs (see Table 3). When comparing the studied TEEs against Linux and real-time OSes of similar complexity (FreeRTOS and VxWorks) both classes of OSes account for a smaller relative percentage of critical and severe vulnerabilities. These observations suggest that the current development methodologies for some of the most popular TEEs are not as robust as the development methodologies of other systems, and may benefit from the adoption of such methodologies.

5.3.3 Sources of Vulnerabilities in TrustZone-assisted TEEs

Overall, we identified three main sources of security vulnerabilities in existing TEE systems: *architectural*, *implementation*, and *hardware*. Architectural issues involve deficiencies in the overall TEE system architecture, e.g., absence of memory protection using ASLR. Implementation issues correspond to flaws in the TEE system's software, e.g., buffer overflows. Hardware issues concern hardware behavior that can be abused to undermine the security of a TEE, e.g., side-channels.

To a great extent, these problems continue to persist. Apart from incremental improvements, TEE systems preserve their original architectural features and retain serious weaknesses. Even the systems which present less critical and severe vulnerabilities, such as Trustonic TEE, suffer from important architectural limitations. Vulnerability reports abound which reveal the presence of critical implementation bugs. Many of these bugs have a similar nature as the ones exploited by the attacks described in Table 1. We identified other kinds of bugs that can further be exploited, e.g., concurrency bugs. Hardware issues are prevalent in TrustZone-enabled SoCs and can potentially be leveraged for launching highly damaging attacks in the future. In the next sections, we present our findings in detail by covering each type of issues.

5.4 Architectural Issues

This section presents the main architectural security issues of existing TEE systems. We group these issues into several categories, and refer the reader to the diagram of Figure 17 which presents the specific internal details of each system.

5.4.1 TEE Attack Surface

TEE systems expose a wide attack surface that can potentially be exploited to compromise the overall security.

System	Core (bin / src)	TAs	Details
Qualcomm TEE (Google Pixel XL)	1.61MB / –	2.71MB	Binary contains the secure monitor (96.2KB) and QSEOS(1.50MB). TAs include device management: bootlocker (76 kB); Android services: keymaster (332 kB), fingerprint (600 kB); DRM and decoding: venus (924 kB), Widevine (391 kB); Common libs: cmnlib32,64(204/256 kB)
Trustonic TEE (Samsung S7)	350KB / –	5,02MB	The monitor (140KB) and trusted OS (210 KB) binaries are separate. There are 3 built in TAs, and 33 loadable TAs taking, which add to 5.02MB, implementing Android system functionality, DRM, kernel integrity management, secure element I/O, etc, either as TAs or drivers.
Huawei TEE (Huawei P8 Lite)	744KB /–	479KB	Secure monitor (47KB) based on ATF. Trusted OS binary contains kernel (305KB) and GlobalTask (329KB). TAs include libc shared library (5KB) and implement Android system services, e.g., keymaster (188KB) and gatekeeper (27KB), amongst several others services.
Nvidia TEE (Nvidia Tegra)	97KB / 123Kloc	80KB	The kernel (60KB/23kloc) is based on little kernel (lk) and the monitor we consider ATF Monitor (36.9KB/100kloc). Two test TAs are considered, trusted_app1 (45KB), implements two tests, swapping operands, and copying a string to a buffer. The second, trusted_app2 (35KB), increments the operands by one, and then overwrites them when replying to the client.
Linaro TEE (Hikey960)	365KB /210Kloc	-	The kernel (328.5KB/110kloc), incorporates pseudo-TAs: kernel modules benefiting from full S-EL1 privileges. In Linaro TEE the monitor is the ATF (36.9KB/100kloc).
Linux (4.14.rc7)	19MB / 15Mloc	-	Linux kernel on hikey960 configured with a number of kernel services and drivers built in.
seL4 (kernel)	166.5KB / 19Kloc	-	Formally verified microkernel. When configured correctly guarantees logical task separation.

Table 4: TCB sizes of TEE systems vs. reference OSes (respectively above and below the middle line): Values obtained from TEE binaries and loadable TAs in firmware / system image file system. For open source systems, software was compiled enabling optimizations. Lines of code were counted using SLOCCount [285].

I01**. SW drivers run in the TEE kernel space:** In general, a TEE system requires the existence of drivers in the SW to mediate access to security-sensitive devices, e.g., a fingerprint sensor for user authentication purposes, or the display framebuffer for secure output of DRM-protected content. Given that drivers tend to be complex and a traditional source of bugs, they should not run in the TEE kernel space (i.e., in S-EL1 mode). Trustonic and Nvidia follow this approach by adopting a microkernel architecture where drivers run in the SW user space (S-EL0). In contrast, Qualcomm, Huawei, and Linaro run TEE drivers in S-EL1 mode. Both Qualcomm and Linaro adopt a monolithic architecture where all the privileged code runs in kernel space. Huawei delegates some of the trusted OS functionality to user space, namely the job of controlling the lifecycle of TAs which is assigned to a privileged TA called GlobalTask (see Figure 17).

I02**. Wide interfaces between TEE system subcomponents:** These interfaces have become worryingly large for TEE systems. In Android OS, at least four daemons have privileged access to the TrustZone driver. The SMC call interface exposed by the TEE kernel gives NW software access to a considerable number of TAs (e.g., Trustonic TEE counts 32 different TAs). The set of commands handled by TAs also tends to be fairly large. For instance, the Widevine TA implements 70 different commands, many of them manipulate complex media data structures. The TEE kernel exposes a large number of

system calls to TAs: 69 syscalls in Qualcomm's TEE. Moreover, access permissions to the TEE system calls are frequently coarse-grained, such as in Qualcomm TEE where TAs have promiscuous access to all system calls. In certain cases, the interface provided by the secure drivers can grow very large, such as in the Trustonic TEE, where the TA that controls access to the fingerprint device driver gives access to virtually every TA deployed in the TA. Most of these issues have been instrumental for the development of the exploits listed in Table 1.

I03**. Excessively large TEE TCBs:** Part of the design philosophy of a TEE system is that it should rely on a small TCB. To verify whether this principle holds for the studied TEE systems, we analyzed their TCB sizes based on their firmware and, when available, on their source code. Given that TAs implement security-sensitive REE functions, we include in the TCB both trusted OS and TAs. Table 4 presents our results comparing them against a few reference OSes. We find that TCBs of TEE systems are substantial, e.g., reaching 1.6 MB in the Qualcomm TEE. Further, these numbers are conservative since additional TAs that are not included the firmware package can be dynamically loaded. Strikingly, some TAs have individually considerable sizes. With such sizes, confidence in the full correctness of these TAs is weakened: since TAs accept inputs from the NW via SMCs, potential vulnerabilities are exposed to easy exploitation. To put TCB sizes in perspective, Table 4 shows that although existing TEE kernels are significantly smaller than the Linux kernel (by about three orders of magnitude), most of them are growing considerably larger than a microkernel of comparable complexity (seL4).

5.4.2 Isolation between Normal and Secure Worlds

A TEE system must enforce strong isolation between NW and SW while enabling efficient communication across worlds. In some TEE systems, this isolation can be undermined by the exposure of dangerous system calls by the TEE kernel.

I04**. TAs can map physical memory in the NW:** Certain applications, e.g., for DRM-protected video rendering, require an efficient shared-memory mechanism that allows for exchanging high volumes of data across worlds with low latency. However, some TEE systems provide mechanisms that can easily be abused for privilege escalation. For example, Qualcomm TEE exposes a trusted OS system call that allows any TA to map any physical memory belonging to the NW, including to the REE OS kernel. As a result, by compromising a TA, an attacker can automatically takeover the Android OS by scanning the physical address space for the Linux kernel and patch it to introduce a backdoor (see E6 in Table 1).

In contrast, Trustonic TEE prevents TAs from mapping in and modifying physical memory. Instead, this operation is restricted to specific driver TAs. Hence, TAs willing to exchange data volumes via shared memory must issue a request to a dedicated driver TA. Samsung uses this approach to split the functionality of the TrustZone-based Integrity Measurement Architecture (TIMA): a TA driver provides the ability to map physical memory while another TA uses this service to measure the integrity of system image. A white list is used to prevent access to the TA driver by arbitrary TAs. However, the white list is hard-coded in the TA

driver and the number of allowed TAs reaches 34, which is fairly large. By compromising any of these TAs, an attacker has free way to hijack Android.

I05**. Information leaks to NW through debugging channels:** Another source of isolation breaches is caused by leakage of information from the SW to the NW via TEE debug mechanisms. Some exploits described in Table 1 have been facilitated by this feature. A privilege escalation attack [256] leverages a system call of the Huawei TEE that allows a TA application to dump its stack trace to a memory region in the NW. Using this mechanism, the attacker can learn the physical address space of the GlobalTask and use this information to craft the exploit. Debugging logs exposed to the NW are also common in the Trustonic TEE which may help disclose sensitive information about the internals of TAs.

5.4.3 Memory Protection Mechanisms

Most TEE system exploits have been facilitated by poorly designed memory protection mechanisms. Table 5 summarizes our findings with respect to the mechanisms implemented for each analyzed TEE system. We highlight the following issues.

I06**. Absent or weak ASLR implementations:** In all analyzed TEE systems, ASLR is either absent or poorly implemented. In Trustonic TEE, TAs are all loaded into the same fixed address in the virtual address space (0x1000). Each TA is provided with a common library which is also mapped to a constant address for each TA (0x7D01000). Thus, any vulnerability found in a TA can be exploited without requiring extra effort in determining the TA's loading address. Furthermore, this common library, named mcLib (see Figure 17), contains a substantial amount of code, which can provide a source of gadgets to call functions, invoke trusted OS system calls, etc.

Likewise, Huawei, Nvidia, and Linaro TEEs offer no ASLR mechanisms. The Qualcomm TEE provides a form of ASLR for all TAs but uses only a small segment of physical memory into which the TA code is loaded. All TAs are loaded into a relatively small region of continuously allocated physical memory spanning around 100MB in size. Consequently, the amount of entropy offered by the ASLR is limited by this region's size. Thus, while it would be theoretically possible to implement high entropy ASLR by using a 64-bit virtual address space, the ASLR implemented by Qualcomm TEE is limited approximately to 9 bits, which greatly reduce the number of guesses an attacker would need to try to guess a TA's base address. None of the studied TEE systems features KASLR, i.e., ASLR for the TEE kernel.

I07**. No stack cookies, guard pages, or execution protection:** In addition to ASLR, modern OSes employ additional memory protection mechanisms. Stack cookies (SC) are unique values that help detect stack smashing instances and abort the program execution. Guard pages (GP) delimit the mutable data segments in each process (namely, stack, heap, and global data) to prevent attackers from using an overflow in one segment to corrupt another by triggering a fault in case of illegal access. Execution protection (XP) prevents programs from executing within certain memory regions and can be achieved by various means. On Arm, the WXN bit in the SCTLR register can be used whereby writable memory

Mechanisms		Qualcomm	Trustonic	Huawei	Nvidia	Linaro
User	ASLR	◐	○	○	○	○
Space	SC	●	○	○	○	○
	GP	○	○	–	–	–
	XP	WXN	WXN	○	UXN/PXN	UXP/PXN
Kernel	KASLR	○	○	○	○	○
Space	SC	●	○	○	○	○
	XP	WXN	WXN	○	UXN/PXN	UXN/PXN

Table 5: Memory protection mechanisms for user and supervisor modes. Filled circle: fully implemented. Half-circle and empty circle: partially implemented or not implemented. Dash: implementation-related information not found.

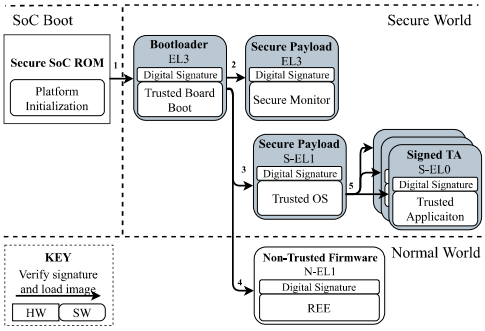


Figure 18: Secure boot process: Implements a chain of trust that starts with the execution of a trusted component – Trusted Board Boot – stored in an on-SoC ROM. Then, each loaded component verifies the authenticity and integrity of the subsequent module, or modules, and loads them if no anomalies are detected. A vendor digitally signs the SW image with its private key, while the respective public key (or its digest) is burned, or flashed into a one-time programmable memory, typically eFuses. The public key is used to verify that the binary has not been modified and it was provided by the vendor.

regions are implicitly marked as Execute Never (XN). Another option is to use memory page attribute XN, Unprivileged XN (UXN), and Privileged (PXN).

However, TEE systems only partially implement these mechanisms (see Table 5), which has facilitated exploitation [256]. Trustonic TEE, in spite of its security-driven goals, lacks stack cookies, making it relatively easy to exploit stack overflows in vulnerable TAs. It allocates both globals and stack from the TA's data segment without providing guard pages in between. Moreover, the memory layout places the stack at the end of the data segment and the globals before it; this is the perfect configuration for overflowing one region into the other. Qualcomm TEE implements randomized pointer-sized stack cookies, but it does not provide guard pages between globals, heap, and stack. Huawei TEE has no stack canaries, no data execution protection, and no write-protected .text section, possibly because Huawei TEE is based on the Micrium μ/OS, an RTOS which leaves aside most of the said memory protection mechanisms to deliver maximum performance.

Class	Subclass	# Bugs
Validation Bugs	Secure Monitor	2 (1.07%)
	Trusted Applications	62 (33.16%)
	Trusted Kernel	52 (27.81%)
	Secure Boot Loader	5 (2.67%)
Functional Bugs	Memory Protection	32 (17.11%)
	Peripheral Configuration	8 (4.28%)
	Security Mechanisms	11 (5.88%)
Extrinsic Bugs	Concurrency Bugs	11 (5.88%)
	Software Side Channels	4 (2.14%)

Table 6: Number of bug reports involving implementation issues.

5.4.4 Trust Bootstrapping

We report a number of architectural issues which might undermine the process of trust bootstrapping by client applications – local or remote – on a TrustZone-assisted TEE platform.

I08**. Lack of software-independent TEE integrity reporting:** Secure boot ensures the authenticity of the software running on a device. Figure 18 illustrates a possible secure boot process, including the booting of TAs. However, Arm TrustZone lacks the hardware mechanisms for securely reporting the software integrity measurements to a remote third party. In the absence of hardware support, remote attestation needs to be implemented in software by one of the TEE components. This weakens the security of remote attestation as it requires the correctness of all SW software of the trust chain running in EL3 mode.

I09**. Ill-supported TA revocation:** Problems have been identified with the way Android OEMs deal with TA revocation [255]. TA revocation is necessary to prevent patched TAs from being downgraded. Updates allow for vulnerabilities and other errors to be corrected, increasing the overall security of the device. To make them easier to update, TAs are usually loaded from the REE filesystem and to prove their authenticity they are digitally signed. However, the TEE must revoke old TAs to prevent attackers in the REE from intentionally loading an old, known vulnerable TA and exploiting it to gain code-execution within the TEE. The successful downgrading of the Widevine TA to a previous, known vulnerable, version in Qualcomm and Trustonic TEEs has been shown [255].

5.5 Implementation Issues

In addition to architectural weaknesses, many TEE vulnerabilities are caused by implementation bugs. To characterize the prevalence of these issues, our primary source consisted of bug reports retrieved from public CVE databases and vendor bulletin reports. Table 6 lists how we classified all the analyzed bugs into a few meaningful categories which we present below.

5.5.1 Validation Bugs

A common type of software bugs in TEE systems involves improper handling of input and/or output values which we refer to by the name *validation bugs*. Examples include buffer overflows, incorrect parameter validation, mishandled integer overflows, etc. Bugs of this nature are very prevalent and frequently used as entry points for privilege escalation. They can be found in all major components of existing TEE systems.

I10**. Validation bugs within the secure monitor:** By exploiting a bug in the secure monitor, an attacker can automatically gain full control of the device. For instance, the vulnerability abused by exploit E1 for hijacking the Qualcomm TEE kernel (see Table 1) allowed an attacker to write a zero double word anywhere in the SW memory by crafting an input into an SMC call. To reduce the chance of critical bugs, most TEE systems (excepting Qualcomm TEE) use Arm's reference monitor (ATF) implementation (see Figure 17). Unfortunately, critical validation bugs have been reported within ATF itself. Ironically, one bug was located on a C macro whose goal was to help detect arithmetic overflows (CVE-2017-9607). Shown in Listing 5.1, any AArch32 code relying on this macro to detect integer overflows is not protected. This means that multiple monitor entry points that use this macro could be vulnerable.

I11**. Validation bugs within TAs:** Besides the secure monitor, TAs are mostly exposed to attacks from the NW through the SMC interface. As it turns out, the largest fraction of vulnerability reports in TEE systems corresponds to validation bugs within TAs. For instance, critical vulnerabilities in the ESECOMM trustlet can be leveraged to compromise client applications such as Samsung Pay [254]. In Trustonic TEE, validation bugs can be exploited systematically using the respective bug fixes [271]. Some TA validation bugs (e.g., CVE-2016-5349) may allow for direct privilege escalation into the Linux kernel through boomerang attacks [286], in which a vulnerable TA does not properly validate the input memory addresses, allowing an attacker to access NW memory region and read or write memory allocated to REE apps or OS.

I12**. Validation bugs within the trusted kernel:** By hijacking a TA, an attacker may successfully elevate its privileges by exploiting a vulnerability in the TEE kernel's system call interface. For instance, an attack on the Huawei TEE [256] relied on a vulnerable system call where its inputs are entirely unchecked for bypassing a security check within the trusted kernel (see Listing 5.2). Even more worrisome, the Qualcomm TEE kernel lacks any code for validating supplied input pointers, which means that all the system calls are vulnerable [42].

I13**. Validation bugs in secure boot loader:** The boot loader may also be prone to attacks by exploiting validation bugs upon system bootstrap. An example is documented in CVE-2017-7932. This vulnerability is due to a stack-based buffer overflow in the X.509 certificate parser which allows an attacker to potentially install or load a crafted X.509 certificate during the image verification. As a result, the legitimate TEE software image can be replaced to attain arbitrary code execution.

```
/* Evaluates to 1 if (ptr + inc) overflows, 0 otherwise.
 * Both arguments must be unsigned pointer values (i.e. uintptr_t). */
#define check_uptr_overflow(ptr, inc)  \
        (((ptr) > UINTPTR_MAX - (inc)) ? 1 : 0)
```

Listing 5.1: Vulnerability in ATF macro. Located in header file include/lib/utils_def.h, this macro aims at detecting arithmetic overflows when computing the sum of a base pointer and an offset. However, if the sum of the input base pointer and offset wraps around, unpredictable behavior might occur. In AArch32 images, it fails to detect overflows when the sum of its two parameters falls into the (2^{32}, 2^{64} - 1) range.

```
signed int ___fastcall sys_call_overwrite(int a1, int a2) {
  signed int v2; // r3@2
  int v4; // [sp+0h] [bp-14h]@1
  int v5; // [sp+4h] [bp-10h]@1
  v5 = a1;
  v4 = a2;
  if ( *(_DWORD *)a1 == 0x13579BDF ) {
    // write (*(int*)(arg1 + 0x18C) + 7) >> 3 to arg2
    *(_WORD *)v4 = (unsigned int)(*(_DWORD *)(v5 + 0x18C) + 7) >> 3;
    v2 = 0;
 }
 return v2;
}
```

Listing 5.2: Reverse-engineered syscall from Huawei TEE (RTOSck) without any input check. An attacker can overwrite memory anywhere in NW or SW.

5.5.2 Functional Bugs

By *functional bugs* we refer to programming errors caused, not by flaws in validating inputs/outputs, but by inconsistencies between the implementation and the program specification intended by the programmer (e.g., incorrectly programming of a cryptographic algorithm). We identified three types of functional bugs that can lead to security breaches in TEEs.

I14**. Bugs in memory protection:** Some functional bugs may introduce security vulnerabilities in the memory protection mechanisms of a TEE system. For instance, a vulnerability reported for ATF [287] involves a configuration error of memory translation tables which allows read-only memory areas to always be executable in the context of the S-EL1. In OP-TEE, we identified 15 bug reports causing memory protection vulnerabilities. For instance, one error in the OP-TEE's secure monitor code responsible for saving and restoring FIQ registers for ARMv7 may allow the REE to escalate privileges to obtain code execution in the TEE [288].

I15**. Bugs in configuration of peripherals:** Misconfiguration of certain peripherals may also be security-critical. In Qualcomm TEE, a flaw disclosed as CVE-2016-10423, allows a TA to read data on an SPI interface previously opened by another TA due to non-exclusive access of the SPI bus. In OP-TEE, one patch [289] aimed to fix a misconfiguration of the pseudo random number generator causing an

insufficient source of entropy for the cryptographic libraries used within OP-TEE.

I16**. Bugs in security mechanisms:** Another potential source of vulnerabilities is the existence of bugs in the implementation of security protocols or cryptography primitives. In ATF, an attacker could bypass the Amlogic S905 SoC secure boot process [281] due to a deficiency in the authentication checks, where only the integrity of the boot image was checked, not the signature. In OP-TEE, for example, a Bellcore attack vulnerability in LibTomCrypt could compromise a private RSA key (CVE-2017-1000412), and a hardcoded security key for RPMB result in the key leakage (fix on 23 Jan 2017).

5.5.3 Extrinsic Bugs

Lastly, we use the term *extrinsic bugs* to refer to programming defects that are not related with validation of values or functional correctness of code, but with the proper handling of external factors that might introduce security vulnerabilities. In particular, we identify two classes of bugs that fit this category.

I17**. Concurrency bugs:** Caused by the interference of multiple concurrent programs, we consider concurrency bugs as extrinsic because their manifestation depends on factors external to the program itself (e.g., thread interleaving). Some concurrency bugs may introduce security vulnerabilities within TEE systems. For instance, in OP-TEE, one bug due to concurrent access to the file system by different TAs [290] allowed a TA to delete a directory on trusted storage while being created by another TA. Samsung reported two race condition vulnerabilities in the TIMA driver deployed in Trustonic TEE (SVE-2017–8974 and SVE-2017–8975). A specific instance of race conditions may lead to TOCTOU vulnerabilities, where some aspect of the system state changes after a condition check, such that the condition-check results are no longer valid. A TOCTOU vulnerability was reported in a DRM TA of the Nvidia TEE which may lead to privilege escalation (CVE-2017-6296).

I18**. Software side-channels:** Another instance of bug types that we consider to be extrinsic is software side-channels, which are caused by specific implementation artifacts that are foreign to the program logic but can reveal undesired information based on the program execution time. For example, a timing side-channel was found in the cryptographic library LibTomCrypt used by OP-TEE's trusted kernel (CVE-2017-1000413). This vulnerability was caused in the optimization of modular exponentiation which leaked information about the exponent. It was fixed by ensuring constant time exponentiation.

5.6 Hardware Issues

TEEs rely not only on the correctness of the software architecture and implementation, but also on the correctness of trusted hardware components. Figure 19 provides an overview of the typical hardware architecture of a TrustZone-assisted TEE system and shows how these components are connected by an AXI bus. Since hardware components are part of the TCB of a TEE, the TEE developers must correctly configure

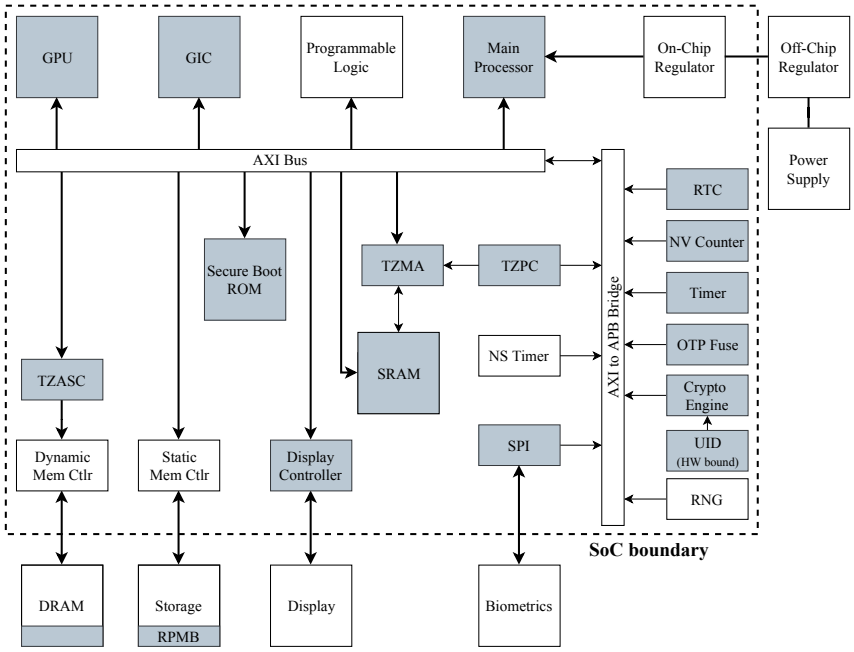


Figure 19: Hardware architecture of a TrustZone-assisted TEE system, including programmable logic present in FPGAs. The fully shaded boxes represented the trusted components exclusively allocated to the TEE software running in the SW, SPI/UART, for example, allow communication with off-SoC peripherals (e.g., for biometric authentication or smartcard interaction). Partially colored boxes represent components that can be partially, or totally, restricted to the SW, such as DRAM, and storage (e.g., to provide secure storage to TAs).

Component	Attack	Device	SoC	TEE	Outcomes
Cache	[257]	Freescale i.MX53	i.MX53 (ARMv7-A)	–	Cache rootkit can evade NW and SW detection
	[258]	Raspberry Pi2	BCM2836 (ARMv7-A)	Self Developed	AES 128-bit key recovery
	[62]	Galaxy S6	Exynos 7420 (ARMv8-A)	Trustonic TEE	AES 128-bit key recovery
	[259]	Freescale i.MX53	i.MX53 (ARMv7-A)	Self-Developed	AES 128-bit key recovery
	[260]	Samsung Tizen TV	(ARMv7-A)	SecureOS	Cross-Core Covert Channels demonstrated by
		Hikey	Kirin 620 (ARMv8-A)	Linaro TEE	transmitting images from SW to NW
Branch Predictor	[58]	LG Nexus 5X	Snapdragon 808 (ARMv8-A)	Qualcomm TEE	Extract 256-bit private keys from Keystore TA
DRAM	[291]	-	Cortex A-9 (ARMv7-A)	Trusty	Derive RSA private key

Table 7: Microarchitectural issues exploited to attack TrustZone-assisted TEEs.

and interface with these components, as well as carefully take into consideration all the implications of the microarchitecture.

5.6.1 Architectural Implications

TEE developers must be well aware of all architectural hardware components, such as FPGAs, and all architectural details, both inside and outside the SoC boundary.

I19**. Attacks through reconfigurable hardware components:** Reconfigurable platforms, i.e., FPGA SoCs, combine a conventional CPU architecture with programmable hardware logic. Although there is no evidence of massive adoption of reconfigurable platforms in the context of mobile devices, OP-TEE supports the Xilinx Zynq-7000 and Zynq UltraScale+ platforms on its mainline. Unfortunately, the addition of new hardware increases the attack surface. Configurable hardware within FPGA SoCs is typically connected to the main bus, which means that hardware must block access to memory regions that are managed by the software running in the main CPU. On TrustZone-enabled systems, the AMBA AXI interface includes an additional control bit (NS bit) for both read (ARPROT) and write (AWPROT) channels on the main system interconnect. This lets all hardware components become aware of the security state of the CPU. Nevertheless, some unusual exploits can take advantage of reconfigurable hardware logic to break the security of TrustZone-based systems [261, 262]. One attack explores malicious hardware deployed on an FPGA to break the secure boot process [261]. In a study about NS bit propagation to FPGA, six different attacks were exposed using small malicious modifications on the hardware logic [262].

I20**. Attacks through energy management mechanisms:** Software-exposed energy management mechanisms can pose significant challenges to system security, possibly in subtle ways. For instance, CLKSCREW [246] relies on a malicious (non-secure) kernel driver to push frequency and voltage regulators to operate beyond the vendor-recommended limits, until the point of inducing faulty computations. By influencing the computation of SW operations, it is possible to break the TrustZone hardware-enforced boundaries to extract secret cryptographic keys and bypass code signing operations.

5.6.2 Microarchitectural Side-Channels

In addition to architectural-level details, the security of TEEs also depends on microarchitectural details (e.g., caches). In this section, we discuss three major classes of microarchitectural aspects that can affect the security of TrustZone-assisted TEEs.

I21**. Leaking information through caches:** On TrustZone-enabled processors, cache memory is shared between the secure and normal worlds. Although the secure cache lines are not accessible by the NW, both worlds are guaranteed equal rights when competing for the use of cache lines. This cache coherence design improves system performance at the cost of cache contention between the two worlds [257]. This contention is the main source of exploitation for extraction of information from the SW by monitoring caches from the NW. R. Guanciale et al. [258] implemented a low-noise cache storage channel which can successfully extract a 128-bit key from an AES encryption service. ARMageddon [62] uses the Prime+Probe technique to infer activities on the SW and distinguish whether a provided key is valid or not. TruSpy [259] also leverages Prime+Probe to recover a full 128-bit AES encryption key in two different ways. Prime+Count was also employed for enabling cross-world covert channels on TrustZone [260].

I22**. Leaking information through branch predictor:** The branch predictor can also be leveraged to attack TrustZone. Modern processors include a branch target buffer (BTB) unit, which stores the

computed target addresses of taken branch instructions and fetches them when the corresponding branch instructions are predicted [292]. Since the BTB is shared between NW and SW, Prime+Probe can be performed to leak secure information to the NW. The process encompasses priming the BTB by executing many branches, and later let the victim process execute which will evict the attacker BTB entries. When the attacker gets control of execution, the attacker re-executes those branches to detect mispredictions. Given that the internal hardware structure of the BTB works at byte granularity instead of cache-line granularity, this particular attack vector increases considerably the spatial resolution of the probe mechanism. A 256-bit private key has been fully recovered from Qualcomm's hardware-backed keystore [58].

I23**. Leaking information using Rowhammer:** Rowhammer is a software-induced hardware fault that affects Dynamic Random Access Memory (DRAM) memories and enables an attacker to flip bits in physical memory by solely performing memory read operations [293, 294]. This type of attack has been explored to subvert TrustZone [291]. A malicious Linux kernel module is used to generate faults to a specific NW target address using Rowhammer, while a secure signature service running on a Trusty TEE instance uses the secure private RSA key to sign a specific message. If the private key is allocated in a secure memory region adjacent to the secure/non-secure memory boundary, the Rowhammer generated by high-rate memory read operations on the non-secure memory border induces faults on the secure one, corrupting the private keys and generating a faulty RSA signature. After retrieving a faulty generated signature on the Linux side, it is possible to deduce the private key. Among the discussed microarchitectural issues, this attack is harder to conduct because it generally requires a higher degree of control over the environment; plus, it is relatively easy to mitigate it.

5.7 Defenses for TrustZone-assisted TEEs

This section presents a compilation of defense techniques that can help overcome the architectural, implementation, and hardware issues prevalent in commercial TEE systems. Table 8 presents examples of some representative papers that introduced some of these defenses. These examples are shown chronologically, from 2014 to 2019. A filled bullet indicates that the respective paper implements at least one defense technique that can help address the issue indicated in the heading of the corresponding column. The caption of the table provides the reading key for interpreting which defenses (numbered as D*xx*) are relevant for each class of TEE issues.

5.7.1 Architectural Defenses

We highlight four relevant techniques that can help mitigate the architectural issues identified in existing commercial TrustZone-assisted TEEs. Each technique addresses a particular subclass of issues presented in Section 5.4.

D01**. Multi-isolated environments:** This technique can be employed to reduce the excessively large

		Architectural Issues				Implementation Issues			Hardware Issues	
		Att. Surf.	*Wor. Iso.*	*Mem. Pro.*	*Tru. Boot.*	*Val. Bugs*	*Fun. Bugs*	*Ext. Bugs*	*Arch. Imp.*	*Micro. S.D.*
2014	TLR [8]	○	○	○	●	●	○	○	○	○
2015	TrustICE [115]	●	○	○	○	○	○	○	○	○
	SeCReT [295]	○	●	○	○	○	○	○	○	○
2016	OSP [117]	●	○	○	○	○	○	○	○	○
	CaSE [105]	○	○	●	○	○	○	○	○	○
	R. Guanciale et al. [258], ARMageddon [62], Truspy [259]	○	○	○	○	○	○	○	○	●
2017	BOOMERANG [286]	○	●	○	○	○	○	○	○	○
	Komodo [101]	○	○	○	●	●	●	●	○	○
	MIPE [132]	○	○	○	○	●	●	●	○	○
	vTZ [121]	●	○	○	○	○	○	○	○	○
	CLKSCREW [246], Jacob et al. [261], Benhani et al. [262]	○	○	○	○	○	○	○	●	○
2018	TFence [296]	○	●	○	○	○	○	○	○	○
	PrivateZone [118]	●	○	○	○	○	○	○	○	○
	RustZone [297]	○	○	○	○	●	○	●	○	○
2019	TEEv [110]	●	●	○	○	○	○	○	○	○
	PrOS [111]	●	○	○	○	○	○	○	○	○
	SANCTUARY [116]	●	●	○	○	○	○	○	○	●
	Ginseng [104]	○	○	●	○	●	○	●	○	○
	K. Ryan [58]	○	○	○	○	○	○	○	○	●

Table 8: Examples of representative papers that contribute with relevant defense techniques (D*xx*) for overcoming reported TrustZone-assisted TEE issues. For architectural issues, filled circle in *attack surface*, *world isolation*, *memory protection*, or *trust bootstrapping*: the paper proposes D01, D02, D03, D04, respectively. For implementation issues, a filled circle in *validation bugs* means it proposes any of D05, D06, or D07; in *functional bugs* proposes D07; and in *extrinsic bugs*, D06 or D07. For hardware issues, *architectural implications* and *microarchitectural side-channels* have a filled circle, respectively, if the paper proposes D08 or D09.

attack surface of commercial TEE systems (see I01, I02, and I03). Multiple isolated environments (other than the standard TA sandboxes in SW) help to reduce exposure of TEE systems to attacks by (a) increasing the isolation granularity between TEE components, thus containing the extent of potential damage caused by a security breach, and/or (b) limiting the amount of code that runs in the SW, thereby reducing the chances of highly damaging SW privilege escalation attacks. Different variants have been proposed. One line of work aims at creating strongly isolated compartments within the NW itself where TAs would be allocated. To protect TAs, TrustICE [115] and SANCTUARY [116] leverage different features of the TZASC. OSP [117], PrivateZone [118], and vTZ [121] instead, explore the hardware virtualization extensions available in NW (NS-EL2) to implement isolated environments. A second line of research retains TAs within the SW but aims to strengthen the isolation between them, e.g., TEEv [110] and PrOS [111] implement a minimalist hypervisor in SW, allowing TAs to run on multiple isolated secure guest OSes. Due to the current lack of hardware virtualization support in SW, both systems use same-privilege isolation to secure the hypervisor from secure guest OSes.

D02**. Secure cross-world channels:** Isolation between worlds can be threatened by vulnerabilities in SW triggered from the NW. In particular, the reported TEE deficiencies that can undermine this isolation (see I04 and I05) may lead to the extraction of sensitive data from SW. Although these specific issues can be addressed by fixing vulnerable TEE kernel system calls, cross-world isolation can further be strengthened

by secure NW-SW channels. Proposed by different authors, these mechanisms help to overcome two existing limitations in mainstream TEEs: (1) absent or weak authentication when accessing TEE resources from NW and (2) potentially insecure shared-memory for data exchange within the channel. SeCReT [295] provides a session key (to REE applications) that can be utilized to encrypt the messages. To protect the session key from untrusted REE kernel, SeCReT interposes mode switches from/to the kernel and removes the key from memory during kernel mode execution. TFence [296] further removes this kernel dependency by creating a partially privileged process – a shielded portion of the REE application process – which can directly communicate with TEE. Both TEEv [110] and SANCTUARY [116] implement exclusive shared memory, and PrivateZone [118] enables communication without sharing memory, i.e. through data copies. Aravind et al. [286] use pointer sanitization for preventing boomerang attacks.

D03**. Encrypted memory:** Existing deficiencies in TEE memory protection (I06 and I07) can mostly be addressed with mechanisms from mainstream OSes (e.g., ASLR, stack cookies). Nevertheless, commercial TEEs can provide stronger security defenses, e.g., against cold boot attacks, by implementing encrypted memory capability. In contrast to Intel SGX, TrustZone does not provide built-in support for on-chip memory encryption. To bridge this gap, CaSE [105] allows TAs to run entirely from the cache and ensures that their state is encrypted while written back to main memory. Along the same vein, Ginseng [104] protects variables tagged by the application programmer as "sensitive", by allocating them on CPU registers and encrypting them at runtime before saving them in memory.

D04**. Trusted computing primitives:** Commercial TEEs rely on secure boot to guarantee the integrity of the TEE image. However, this mechanism, *per se*, is insufficient to enable a TA's client – local or remote – to verify the integrity and identity of both TEE and TA binaries (see I08, I09). To overcome this limitation, commercial TEEs can implement additional trusted computing primitives that help provide such guarantees, namely remote attestation and sealed storage. For instance, TLR [8] includes a sealed storage primitive that allows for protecting data cryptographically and bind it to specific hash values of the TEE/TA software stack. Komodo [101] demonstrates how to implement, for TrustZone-assisted TEEs, the security protocols of sealed storage and remote attestation as originally specified for enclaves (i.e., Intel SGX's secure environments for TAs). There is also a body of work in trusted I/O path primitives [137, 208] which aims at providing secure access to peripherals. Given that we identified a relatively small number of vulnerabilities involving access to peripherals, which can be mitigated using standard hardware features for I/O mediation (e.g., System Memory Management Unit (SMMU), bus bridges), Table 8 omits such references.

5.7.2 Implementation Defenses

With respect to defenses that can be leveraged to improve the implementation correctness of TEE components and TAs, we underline three main techniques. Some of these techniques can be applied to prevent more than one single type of bugs, i.e., validation, functional, and/or extrinsic bugs (see

Section 5.5).

D05**. Managed code runtimes:** Commercial TEE systems are mostly written in the C programming language which allows for compiling highly efficient code but do not provide memory safety. However, many validation bugs are caused by memory violation errors introduced by the programmer. In alternative TEE systems, such as in TLR [8], TAs are not compiled to native code, but rather to .Net managed code which is then interpreted by a small-sized managed code runtime (akin to a JVM). At the expense of some performance overhead, the managed runtime helps to prevent validation bugs, e.g., by implementing run-time memory checks and garbage collection.

D06**. Type-safe programming languages:** Researchers have explored the idea of using type-safe programming languages to implement specific components of TrustZone-assisted TEE software. Notably, RustZone [297] is an extension for OP-TEE where TAs are implemented in the Rust programming language. Given that Rust provides memory and thread-safety, RustZone can help prevent validation bugs and some concurrency bugs responsible for crippling TA software (see I11). The Rust programming language has also been used in Ginseng [104] for implementing a large part of the software that runs in monitor mode, i.e., the GService (see I10).

D07**. Software verification:** Implementation bugs tend to exist due to a mismatch between the expected requirements of a piece of software and its actual implementation. Software verification, which comprises techniques such as model checking, symbolic execution, and formal methods, aims at preventing this mismatch by ensuring that the implementation fully satisfies all envisioned requirements. For this reason, it has the potential to help prevent all three classes of prevalent TEE implementation bugs. However, these techniques can be challenging to apply in practice, not only because they require considerable effort and skill, but also because they are difficult to scale for complex programs. Despite these obstacles, important advances have been achieved with the formal verification of specific TEE components, e.g., a small TEE monitor named Komodo [101], which implements the specification of Intel SGX enclaves, and a memory manager called MIPE [132].

5.7.3 Hardware Defenses

Next, we cover relevant countermeasures known to date for addressing hardware issues affecting TrustZone-assisted TEEs.

D08**. Architectural countermeasures:** Hardware manufacturers tend to increasingly pack more components into the SoC chips, becoming very difficult for TEE designers to fully understand its implications to the security of a TEE system. To prevent a growing abuse of reconfigurable hardware technology (see I19), researchers have proposed: (1) the inclusion of a small hardware wrapper into all IP cores endowed with an AXI interface so as to restrict their operation during system boot [261]; (2) the implementation of a dedicated AXI interconnect for secure devices [262]; and (3) the inclusion of a non-secure only port to connect all non-sensitive memory-mapped IP cores and restrict its operation through memory protection

		Dedi-cated RAM	Cross-World Isol.	Encryp. Mem.	Protec-tion Ring	Attestation	Previously Exploited	Communica-tion w/ REE
CPU Extensions	Arm TrustZone [31]	○	MMU + HW	○	-2	sec. boot.	●	sh. mem.
	Intel SGX [20]	○	MMU + HW	●	1	remote att.	●	data copy
	Intel SMM [298]	○	MMU	○	-2	sec. boot.	●	sh. mem.
	Sanctum [299]	○	MMU + HW	○	-2	sec. boot.	○	data copy
Co-Processors	Apple SEP [25]	●	Phys. + HW	●	-3	sec. boot.	○	sh. mem.
	Qualcomm SPU [26]	●	Phys. + HW	●	-3	sec. boot.	○	sh. mem.
Chips	Intel ME [27]	●	Phys.	●	-3	sec. boot.	●	sh. mem. + HECI
	Google Titan-M [28]	●	Phys.	○	-3	sec. boot.	○	SPI/USB/I2C
	TPM [29]	○	Phys.	○	-3	sec. boot.	●	SPI/I2C/LPC
Virtualization	Windows VSM [300]	○	MMU	○	-1	sec. boot.	●	sh. mem.
	AMD SEV [87]	○	MMU	●	-1	remote att.	●	sh. mem.
RISC-V	Multizone [301]	○	PMP	○	-2	sec. boot.	○	data copy
	Keystone [245]	○	PMP	○	-2	remote att.	○	sh. mem.

Table 9: *Dedicated RAM:* used for allocation of security-sensitive state and isolation from potentially insecure main RAM. *Cross-world isolation:* implemented using memory management components (MMU / PMP) or in combination with HW-specific features (e.g., TrustZone's TZASC); dedicated off-SoC chips achieve isolation through physical separation. *Encrypted memory:* filled circle indicates that hardware-enforced memory encryption is supported. *Protection Ring:* classified in five levels [7], i.e., 1 (user), 0 (kernel), -1 (hypervisor), -2 (monitor), -3 (off-chip). *Attestation:* if the TEE runtime can perform local attestation only (i.e. secure boot), or remote attestation also. *Previously exploited:* black circle indicates publicly known exploits to TEE systems enabled by that particular technology. *Communication mechanisms with REE:* shared memory, data copying, and communication bus (e.g. USB or SPI).

mechanisms (e.g. SMMU) [262]. To prevent misuse of hardware voltage regulators (see I20), a possible approach is to place specific operation limits into the software (i.e., drivers) or into the hardware itself [246].

D09**. Microarchitectural countermeasures:** One way to prevent cache side-channels (see I21) is through careful implementation of cryptographic algorithms in software [58, 62, 258, 259] or using dedicated hardware (e.g., specific ISA instructions such as AESD and AESE in Armv8-A) [62] to prevent information leakage in cryptographic-related operations. Another path is to leverage cache maintenance techniques to prevent information leakage through caches. For TrustZone-assisted TEEs that do not use shared L2 cache, one approach is to flush the L1 cache on every SW exit [116]. If shared L2 cache is used, although cache flushing (total or selective) or cache normalization operations performed at every SW entry and exit may be sufficient to prevent cache-storage attacks [258], L1 flushing may not be able to prevent Prime+Probe attacks in multicore systems [62]. In this case (which also holds for all aforementioned cases), cache partitioning can prevent an attacker from leveraging contention with victim [62, 116, 259]. Carefully implemented cryptographic algorithms seem also to be effective at preventing breaches through the BTB (see I22). This was shown and highlighted by Keegan et al. [58], where different versions of an

algorithm were able to render side-channels ineffective. To prevent Rowhammer attacks (see I23), TEEs must avoid the use of memory at the NW-SW boundary.

5.8 Beyond TrustZone-assisted TEEs

Although our work is focused on TEEs specifically assisted by TrustZone, there are alternative TEE-enabler hardware technologies. In this section, we briefly present some related technologies and highlight their main features in Table 9.

One class of hardware technologies provides a set of CPU extensions where the processor is augmented with circuitry that implements specific TEE-enabling security functionality. TrustZone fits this category as well as technologies such as Intel SGX [20], Intel System Management Mode (SMM) [298], and Sanctum [299], for instance. Separate co-processors in the SoC, such as Apple SEP [25] or Qualcomm SPU [26], may include dedicated non-volatile storage and RAM which allows for reducing shared hardware resources and help prevent side-channel attacks [62, 302]. In dedicated security chips, the runtime environment comprises a processor, memory, and persistent storage. For instance, Intel Management Engine (ME) [27] is a firmware based on Minix OS that runs on a separate processor in Intel systems. It is designed to be an almost fully independent system, with access to many peripherals and its own secure boot functionality. Some security chips may be equipped with tamper detection, as in the case of the Titan-M [28]. Others, such as TPM [29], implement specific functions for trusted boot, remote attestation, and other primitives. Hardware support for virtualization can also be used for implementing TEEs. In Windows' Virtual Secure Mode [300] the hypervisor establishes two hierarchical privileges modes VTL0 (analogous to the normal world) and VTL1 (analogous to secure world). AMD SEV [87] provides the ability to encrypt virtual machine memory using hardware-accelerated memory encryption. Lastly, RISC-V is an instruction set architecture which, although not widely deployed yet, can also be used for implementing TEEs [245, 301].

5.9 Conclusion

This paper presents a vulnerability study of TrustZone-assisted TEEs. Despite the common belief that TEEs are secure due to their hardware-enforced isolation capability and small TCB, our study reports on numerous pieces of evidence that question this assumption. In particular, current TEE systems have serious limitations at the implementation, architecture, and hardware levels that potentially introduce exploitable vulnerabilities affecting millions of devices. Based on our analysis, we highlight multiple state-of-the-art defenses, proposed by the research community, which we believe can make commercial TEE systems substantially more secure.

Acknowledgments. We thank our shepherd David Kohlbrenner and the anonymous reviewers for their comments and suggestions. We are grateful to Joakim Bech for the insightful discussions about

OP-TEE. This work was supported by national funds through Centro ALGORITMI, Instituto Superior Técnico / Universidade de Lisboa, and FCT via projects UIDB/00319/2020 and UID/CEC/50021/2019. David Cerdeira was supported by FCT grant SFRH/BD/146231/2019.

6

ReZone: Disarming TrustZone with TEE Privilege Reduction

Publication Data

D. Cerdeira, J. Martins, N. Santos, and S. Pinto. In: USENIX Security Symposium. 2022.

ReZone: Disarming TrustZone with TEE Privilege Reduction

David Cerdeira†, José Martins†, Nuno Santos‡, Sandro Pinto†

†Centro Algoritmi, Universidade do Minho

‡INESC-ID / Instituto Superior Técnico, Universidade de Lisboa

Abstract

In TrustZone-assisted TEEs, the trusted OS has unrestricted access to both secure and normal world memory. Unfortunately, this architectural limitation has opened an aisle of exploration for attackers, which have demonstrated how to leverage a chain of exploits to hijack the trusted OS and gain full control of the system, targeting (i) the REE, (ii) all TAs, and (iii) the secure monitor. In this paper, we propose ReZone. The main novelty behind ReZone design relies on leveraging TrustZone-agnostic hardware primitives available on COTS platforms to restrict the privileges of the trusted OS. With ReZone, a monolithic TEE is restructured and partitioned into multiple sandboxed domains named *zones*, which have only access to private resources. We have fully implemented ReZone for the i.MX 8MQuad EVK and integrated it with Android OS and OP-TEE. We extensively evaluated ReZone using microbenchmarks and real-world applications. ReZone can sustain popular applications like DRM-protected video encoding with acceptable performance overheads. We have surveyed 80 CVE vulnerability reports and estimate that ReZone could mitigate 86.84% of them.

6.1 Introduction

Arm TrustZone [19] is a technology embedded into Arm processors shipped in billions of mobile phones and embedded devices. Vendors and OEM rely on TrustZone for deploying TEEs, which protect the execution of sensitive programs named TAs. Some TAs implement kernel-level services of the operating system (OS), e.g., for user authentication or file disk encryption [303]. Other TAs provide shared user-level functionality, e.g., DRM media decoders [37] or online banking services [36]. TEEs themselves consist of a trusted software stack offering API and runtime support for hosting TAs. TEEs such as Qualcomm's QSEE and OP-TEE protect the confidentiality and integrity of TAs' memory state, thereby ensuring it cannot be inspected or tampered with by a potentially compromised OS.

To protect the TAs, TEEs leverage mechanisms provided by TrustZone. In Armv7-A/Armv8-A, this technology provides two execution environments named *normal world* and *secure world*[1]. The normal world runs a rich software stack, named REE, comprised of full-blown OS and applications; it is generally

[1]As per Arm's recent documentation [65], *secure world* can be referred to as *trusted world*; we adopt the previous and more familiar terminology [19].

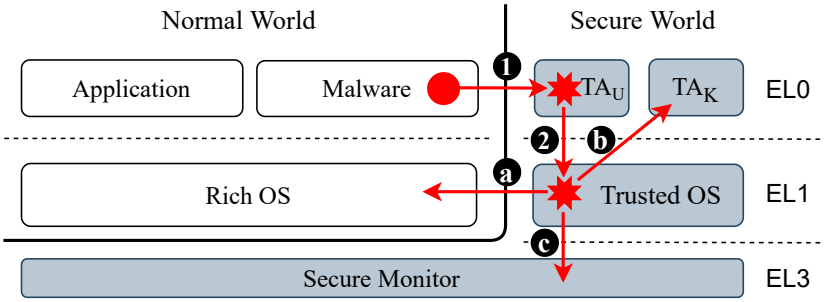


Figure 20: Privilege escalation attack in a TrustZone-assisted TEE: (1) hijack a user-level TA, e.g., exploiting vulnerability in the Widevine TA (CVE-2015-6639) [263], (2) hijack the trusted OS, e.g., exploiting vulnerability in syscall interface of Qualcomm's QSEOS (CVE-2016-2431) [42]. Once in control of the trusted OS, other attacks can be launched: (a) control the rich OS, e.g., backdoor in the Linux kernel [253], (b) control a kernel-level TA, e.g., to extract full disk encryption keys of Android's KeyMaster service [40], or (c) control the secure monitor, e.g., to unbrick the device bootloader [267].

considered untrusted. The secure world runs a smaller software stack consisting of TEE and TAs. TrustZone enforces system-wide isolation between worlds and provides controlled entry points for world switching whenever the REE invokes TAs' services.

However, in the TrustZone security architecture, the *trusted OS* – i.e., one of TEE's core components – runs with an excess of privileges, exposing the entire system to catastrophic privilege escalation attacks if this component gets compromised. Figure 20 illustrates these attacks. It shows the TEE stack, consisting of secure monitor and trusted OS. The secure monitor runs in EL3 – the most privileged exception level – and provides essential mechanisms for world switching and platform management. The trusted OS runs in S.EL1 – i.e., secure kernel mode – and implements memory management, scheduling, IPC, and common services for TAs. TAs run in unprivileged S.EL0 – i.e., secure user mode. TAs are directly exposed to malicious requests coming from normal world applications. If a TA's control flow gets hijacked as a result of exploiting a software vulnerability (1), the attacker can control that TA. If the attacker can further hijack the trusted OS by exploiting another vulnerability (2), he can gain full control of the system, including the REE in the normal world (a), all the TAs (b), and the secure monitor (c).

This level of control is possible due to a fundamental violation of the principle of least privilege where an excess of privileges is dangerously (and unnecessarily) granted to some protection modes in the secure world. In particular, the attacks just described leverage the fact that S.EL1 gives full permission to access and reconfigure both secure and non-secure memory regions. Such level of control, however, should be reserved only to the highest protection domain, i.e., EL3. Similar problems mostly carryover to Armv8.4-A and onward, with the new S.EL2 hypervisor mode for the secure world having the same privileges over the monitor and the normal world as S.EL1 had up until Armv8.4-A; if the secure hypervisor is exploited the entire system will be compromised. Armv9-A is also susceptible to some of these concerns, for instance, allowing S.EL2 to arbitrarily control the normal world state.

Unfortunately, numerous devices face serious risks of being compromised by attacks of this nature. In the past, security researchers have demonstrated how these attacks can be mounted by targeting the

trusted OS through a chain of vulnerability exploits as described in the caption of Figure 20. Notably, recent studies [43] have shown that commercial TEE systems have plenty and serious vulnerabilities, most of them affecting user-level TAs and trusted OS. This suggests that the current attack surface for TEE systems is seriously enlarged.

Despite the abundant research on TrustZone security [19, 43], there is a lack of practical system defenses if the trusted OS is hijacked. TEE systems like Sanctuary [116] aim to reduce the attack surface of the trusted OS by relocating TAs onto user-level enclaves in the normal world. However, due to portability reasons, vendors and OEMs continue deploying TAs in commercial TEEs, therefore exposing many devices to potential attacks. Another approach is to confine the TEE stack inside a secure world sandbox, e.g., by leveraging same privilege isolation [110, 111] or hardware virtualization [82]. However, these approaches do not solve the fundamental problem of the excess of privileges granted to S.EL1 (or S.EL2).

This paper presents ReZone, a new security architecture that can effectively counter ongoing privilege escalation attacks by reducing the privileges of a potentially compromised trusted OS. We build our solution specifically for restricting the privileges of S.EL1 on Armv8-A platforms featuring the protection modes displayed in Figure 20, and then discuss how our techniques can be employed for reducing the excess of S.EL2 privileges in Armv8.4-A and ensuing architectures. With ReZone, a typical monolithic TEE can be restructured and partitioned into one or multiple *zones*, which consist of sandboxed domains inside the secure world. Zones have access only to private memory regions, and provide an execution environment for untrusted $S.EL_{1/0}$ code, i.e., trusted OS and TAs. ReZone restricts the memory access privileges of the code running inside a zone, preventing it from arbitrarily accessing memory allocated for: (a) the normal world REE, (b) other zones, and (c) the secure monitor. Figure 21 depicts a setup featuring two zones – 1 and 2 – each of them runs a trusted OS instance as in a library OS hosting local user- or kernel-level TAs. An attacker hijacking the trusted OS in zone 1 (as explained in Figure 20), will not be able to further escalate its privileges beyond its sandbox. Additional zones can be created, potentially hosting different trusted OS software.

A central novelty of ReZone's design is that we leverage TrustZone-agnostic hardware primitives available in COTS SoCs to restrict the privileges of S.EL1 (trusted OS) code. Existing systems such as Sanctuary [116] have already leveraged the TZASC for restricting access permissions to various physical memory regions in the normal world. However, S.EL1 code can still revert these permissions, rendering the TZASC insufficient for zone isolation. Instead, we use a platform resource controller such as the Resource Domain Controller (RDC) existing in i.MX8MQ and i.MX8QM SoCs, which allows specific bus masters to control memory access permissions. In ReZone, we harness this feature along with a low-end microcontroller also available in the SoC to control these permissions, effectively creating private memory regions for each zone. Resource Domain Controller (RDC) and microcontroller can then be used as building blocks for restricting the memory access privileges of a (potentially compromised) trusted OS, and enforcing strong zone isolation.

Based on this insight, we have built ReZone. Our system includes multiple non-trivial optimizations and fine-grained mechanisms to guarantee system-wide security (e.g., avoiding cache-based attacks based

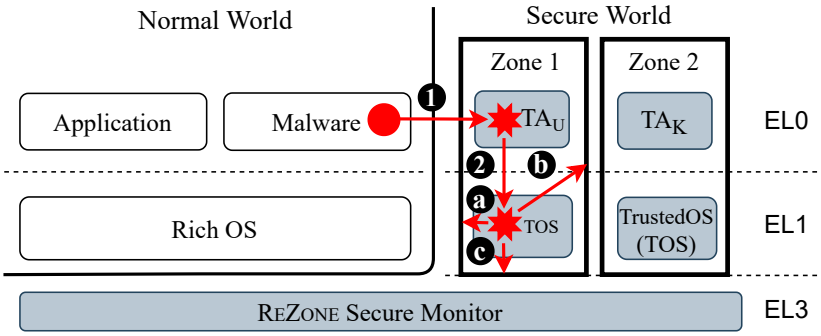


Figure 21: TEE privilege reduction with ReZone: an attacker cannot escalate privileges from the compromised trusted OS running in zone 1 to (a) the rich OS, (b) the kernel-level TA_K in zone 2, or (c) the secure monitor.

on cached content available for EL3 and S.EL1 code). To make our design generally applicable to various hardware platforms, we specify the general properties of SoCs that satisfy ReZone's requirements. We have fully implemented a ReZone prototype for the i.MX 8MQuad EVK, and integrated it with Android OS as REE and with OP-TEE as TEE. Our code is open sourced [304].

We evaluated ReZone using microbenchmarks and real-world applications. Our evaluation shows that despite the existence of non-negligible performance overhead in microbenchmark programs, ReZone does not sensibly degrade the performance of applications such as crypto wallets or DRM media decoders. The DRM application, which requires frequent transitions between the normal world and the zone, preserves its ability to replay video at high frame rates. We have surveyed 80 CVE vulnerability reports and estimate that ReZone can thwart 86.84% of potential privilege escalation attacks arising from exploiting these bugs.

6.2 Background and Motivation

6.2.1 Standard TrustZone Mechanisms

We focus on TrustZone for Armv8-A architectures not featuring S.EL2. CPU cores can operate in one of two states: *secure* and *non-secure*. At exception levels EL0 and EL1, a processor core can execute in either of these states. For instance, the processor is in non-secure exception level 1 (NS.EL1) when running REE kernel code, and in secure EL1 (S.EL1) if executing the trusted OS. EL3 is always in secure state. To change security states, the execution must pass through EL3.

TrustZone extends the memory system by means of an additional bit called *NS bit* that accompanies the address of memory and peripherals (see Figure 22). This bit creates independent physical address spaces for normal world and secure world. Software running in the normal world can only make *non-secure accesses* to physical memory because the core always tags the bus NS bit to 1 in any memory transaction generated by the normal world. Software running in the secure world usually makes only *secure accesses* (i.e., NS=0), but can also make non-secure accesses (NS=1) for specific memory mappings using the NS flags in its page table entries. The processor maintains separate virtual address spaces for the secure and

non-secure states. Trusted OS (or the secure monitor) can map virtual addresses to non-secure physical addresses by setting to 1 the NS bit in its page table entries.

TrustZone-aware memory controllers are usually configured to assign non-secure/secure physical address spaces to different physical memory regions. In the Arm architecture, the TZASC enforces physical memory protection by allowing the partitioning of external memory on secure and non-secure regions. (The TZPC performs a similar function for peripherals.) To date, Arm has released two TZASC versions: Arm CoreLink TZC-380 and TZC-400. They share similar high-level features, i.e., the ability to configure access permissions for memory regions (contiguous address space areas).

6.2.2 The Excess of Privileges of the Trusted OS

The TrustZone mechanisms described above were designed under the assumption that the S.EL1 (trusted OS) and EL3 (secure monitor) are trusted. However, if the trusted OS is hijacked, the attacks described in Figure 20 can be trivially mounted. For illustration purposes, suppose that the physical memory addresses PA_{OS}, PA_{TK}, and PA_{SM} are allocated to the rich OS, TA_K, and secure monitor, respectively. The attacker could mount said attacks by creating virtual address mappings in the S.EL1 page tables as indicated below, and use the virtual addresses VA_i to access (read/write) the memory pages of the rich OS, TA_K, or secure monitor. Notation $VA \longrightarrow \langle PA, NS \rangle$ denotes a page table entry translating virtual address VA to the physical address PA and associated NS bit value:

$VA_{OS} \longrightarrow \langle PA_{OS}, 1 \rangle$, $VA_{TK} \longrightarrow \langle PA_{TK}, 0 \rangle$, $VA_{SM} \longrightarrow \langle PA_{SM}, 0 \rangle$

Simply put, *the secure monitor cannot prevent the trusted OS from mapping these memory regions arbitrarily into the S.EL1 address space.* Given that the trusted OS does not require this level of control to sustain its typical operations, this means that S.EL1 is endowed with excessive privileges that can be abused. If the trusted OS is compromised, the attacker can leverage these privileges to take over every part of the system, including the monitor, TZASC, and normal-world components. For this reason, we argue that reducing the memory access privileges of S.EL1 code can significantly lower the impact of vulnerability exploits targeting the trusted OS. In this paper, we concentrate primarily on devising a solution to this problem. Then, in §6.8.2 and §6.8.3, we explain that Arm's architecture releases starting from Armv8.4 did not entirely solve the problem of over-privileged protection modes in the secure world, and discuss how our techniques can potentially be applied to these more recent architectures.

6.2.3 Platform Model for Restricting S.EL1

Reducing privileges of S.EL1 code to prevent unauthorized memory accesses is not trivial using standard TrustZone mechanisms alone. Even though TZASC controllers such as TZC-380 and TZC-400 can restrict memory accesses, since S.EL1 can reconfigure the TZASC, the trusted OS can always override any permissions for blocking the trusted OS's access to certain memory regions. Also, it is not possible to

Vendor	SoC Platform	PPC	ACU	RZ?	Sources
NXP	iMX8MQ	RDC	Cortex-M4	Yes	Ref. manual (https://www.nxp.com/webapp/Download?colCode=IMX8MDQLQRM)
	iMX8QM	xRDC	SCU	Yes	Ref. manual (https://www.nxp.com/webapp/Download?colCode=IMX8QMRM)
Xilinx	UltraScale+ MPSoC	XMPU, XPPU	PMU	Yes	Ref. manual (https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf)
Nvidia	Tegra X1 and X2, Xavier	SMMU	BPMP	Yes	Ref. manual (https://developer.nvidia.com/embedded/downloads)
Socionext	SC2A11	SMMU	SCP	Yes	TF-A V2.6 (https://github.com/Arm-software/arm-trusted-firmware) and U-boot v2022.01 (https://github.com/u-boot/u-boot)
Qualcomm	Snapdragon 845, 855, 865, 888, 8gen1	SMMU, XPU*	SPU	Yes	Linux 5.16 for SD855 (https://gitlab.com/sm8150-mainline/linux), CVE-2020-11252, CVE-2017-18311, [74]. *We lack the data to support that 845 and 8gen1 contain an XPU, though earlier platforms of the same segment feature one.
Broadcom	Stingray	SMMU	SCP	Yes	TF-A (Same as Socionext), Linux 5.16 (https://github.com/torvalds/linux)
Samsung	Exynos 990	Periph. MMUs	iSE	N/A	S20 and S21 Linux kernel source code
	Exynos 2100	Periph. MMUs	eSE	N/A	(https://opensource.samsung.com/main)
Mediatek	Dimensity 1200	✗	✗	No	OnePlus Nord 2 Linux kernel source (https://github.com/OnePlusOSS/android_kernel_oneplus_mt6893.git)
HiSilicon	Kirin 9000	✗	PMU	No	Huawei Mate 40 Pro Linux (https://consumer.huawei.com/en/opensource/)

Table 10: Availability of Platform Partition Controller (PPC)/ACU hardware primitives and applicability of ReZone on COTS platforms.

distinguish accesses from different privilege levels, thus secure monitor (VA_{SM}) and TAs (VA_{TK}) memory cannot be restricted.

Given the limitations of vanilla TrustZone mechanisms, we propose to leverage complementary, TrustZone-agnostic hardware features available in COTS platforms for preventing illegal memory accesses potentially performed by untrusted S.EL1 code. These features consist of three hardware primitives that we require to be offered by the underlying platform.

1. PPC. Consists of a memory and peripheral protection controller that can restrict physical accesses to a given resource, i.e., memory or peripheral (see Figure 22). On the i.MX8MQ SoC, the PPC is implemented by the RDC peripheral. We characterize the PPC by having four main properties. It must be capable of: (a) intercepting all physical requests regardless of both the NS bit signal injected into the system bus and the configurations of the TZASC, (b) setting fine-grained access control attributes (e.g., RO, RW) for configurable memory regions; (c) differentiating between multiple bus masters, and (d) protecting memory-mapped input/output (MMIO) regions, including the PPC's memory-mapped registers. Importantly, (c) and (d) help to ensure that the PPC's configurations cannot be arbitrarily changed by (untrusted) S.EL1 code executed by the processor. Intuitively, to provide this guarantee in the setup illustrated in Figure 22, the PPC would

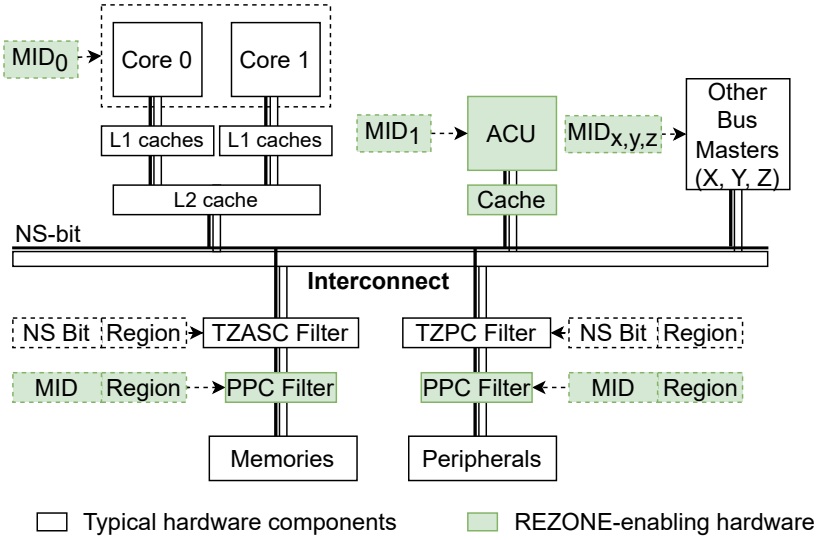


Figure 22: Hardware platform featuring TrustZone: ReZone-enabling hardware components are highlighted, i.e., ACU and PPC. PPC is configured by the bus master MID_1 (ACU). These configurations determine which access permissions are granted to a given MID for a specific physical memory region.

not allow the bus master identified with MID_0 – i.e., the processor cluster – to arbitrarily change the PPC configurations. Instead, as explained next, it is the bus master identified with MID_1 that will have that privilege.

2. ACU. ACUs are small microcontrollers like Cortex-M4 added into the SoC for housekeeping purposes. For example, Figure 22, the ACU is identified by MID_1. The ACU coordinates the configuration of memory access permissions in the PPC, ensuring that the main processor cores cannot tamper with the PPC permissions.

3. Secure boot. Secure boot validates the integrity and authenticity of the firmware, ensuring that the ACU has been securely bootstrapped and fully controls the PPC.

6.2.4 Hardware Support on COTS Platforms

The primitives presented above are readily available on i.MX8MQ and i.MX8QM SoCs, the first of which we use for building ReZone. In i.MX8MQ, PPC and ACU are implemented, respectively, by RDC and Cortex-M4 hardware. To further assess if ReZone can be applied at a large scale, although secure boot is already commonly available, we need to determine if COTS platforms include hardware components that can play the role of PPC and ACU. To this end, we analyzed 17 popular SoCs from nine different vendors. We based our study on several sources. In some cases, we consulted publicly available SoC reference manuals. In others, we inspected the source code of the Linux kernel, bootloader, and secure monitor looking for PPC/ACU driver support. CVEs were also useful for identifying a PPC-like mechanism on Qualcomm SoCs. Our main findings are presented in Table 10.

PPC hardware candidates. Apart from the RDC, we identified several other components that can be

used as PPC: SMMU, XPU, and XMPU in conjunction with XPPU. The Arm SMMU is an Input/Output Memory Management Unit (IOMMU) architecture that can translate an input address to an output address based on address mapping and memory attributes (i.e., translation tables). The main requirement for an IOMMU to be used as a PPC is the possibility of performing address translation functions for multiple masters, in particular the CPU. The Arm SMMU architecture, in its different versions, i.e., Arm SMMUv1, SMMUv2, SMMUv3, can fully provide such a feature. All of our studied platforms by Nvidia, Qualcomm, and Broadcom include a full-fledged SMMU. However, some custom SMMU implementations are designed to only interpose accesses of specific bus masters, e.g., media processing hardware. Such is the case of Mediatek 1200 and Kirin 9000. Samsung Exynos 900/2100 include dedicated SMMUs for specific peripherals. However, due to the lack of available information, we cannot ascertain that these SMMUs can also filter CPU accesses. The XPU [74] is a Qualcomm hardware component that entirely fills the PPC requirements. The same is true for Xilinx's XMPU and XPPU, which restrict accesses to memory and peripherals respectively [305]. In general, we observe that recent platforms are more likely to feature PPC-like mechanisms[2].

ACU hardware candidates. On the analyzed SoCs, we identified various ACU candidates. Qualcomm, Samsung, and NXP's i.MX8QM include security co-processors that can be used for this purpose: SPU [26], integrated/embedded Secure Element (iSE and eSE), and System Controller Unit (SCU). All other surveyed SoCs incorporate programmable microcontrollers usually reserved for platform management purposes, e.g., power management. With one exception, these microcontrollers can be used as a full-fledged ACU: Mediatek's SoC has a System Control Processor (SCP), but the SCP interface is based on MMIO registers and not a mailbox interface as the other analyzed ACUs. Therefore, this SoC is likely too restrictive for ReZone. In summary, our findings are consistent with Arm's reports [306], which point to a clear trend in COTS SoCs to include companion processing units for power management and security operations.

Platform support for ReZone. In our study, we found that 13 SoCs have suitable PPC and ACU hardware, which make the deployment of ReZone possible on devices featuring these platforms. In contrast, the SoCs from Mediatek and HiSilicon lack at least one of the required PPC or ACU mechanisms to accommodate ReZone. In the case of Samsung's SoCs, we miss important information to issue a final judgment. In general, our study shows that our defense mechanisms can further be ported to platforms beyond the i.MX8 SoC family, which we use for our implementation. All NXP's i.MX8, Xilinx's Zynq, and Qualcomm's Snapdragon SoCs deployments span a wide range of edge computing settings (e.g. automotive) [307]. It is interesting to note that most of the analyzed Qualcomm platforms are compatible with ReZone, which means that our system can potentially be deployed on many commodity mobile

[2]At a first glance, it may seem that TZASC fits all the PPC requirements enumerated in §6.2.3. However, a TZASC only protects memory regions. Therefore, it misses the PPC requirement (d). Additionally, only TZASC-400 can differentiate between bus masters, a highly specific hardware component not widely deployed. In fact, looking at the latest TF-A release [96] only two platforms support TZASC-400: NXP's lx2160a and ls1028a. To control access to peripherals, the TZPC is also needed. However, available documentation [71] shows that the TZPC cannot distinguish between multiple bus masters or prevent accesses to itself, missing PPC requirements (c) and (d).

devices.

6.3 Design Goals and Threat Model

We aim to design a security architecture that leverages the primitives presented above to create secure world sandboxes. We refer to these sandboxes as *secure world zones*, or simply *zones*. A zone is meant to host a partitioned TEE stack running at S.EL$_{1/0}$, i.e., trusted OS and TA software (see Figure 21). Since multiple zones z_i can co-exist within the secure world, we use the superscript notation S.EL$^{i}_{1/1}$ to identify zone i. More concretely, we strive to attain four subgoals:

1. Reduce the privileges of the trusted OS. As depicted in Figure 21, zones must thwart the escalation of privileges of an attacker from the trusted OS to other memory regions of the system. To this end, a zone z_i is expected to enforce three security properties: (P1) protection of the normal world, i.e., software running at NS.EL$_{1/0}$, (P2) protection of the secure-monitor, i.e., EL3 code, and (P3) protection of co-located zones, i.e., software running at S.EL$^{j}_{1/0}$, where $i \neq j$.

2. Depend on a small TCB. To implement zones in the secure world, we will rely on the secure monitor as root of trust, and incorporate as little additional code as possible into the TCB, e.g., for managing PPC and ACU.

3. Maintain TEE software portability. Legacy TEE and TA software abound. Vendors and OEMs should easily be able to port preexisting TEE/TA stacks into zone-based environments, requiring minimal to no changes in the code.

4. Offer a good performance/security trade-off. The overheads incurred by our solution should not significantly slow down typical real-world TrustZone-based TEE workloads. In other words, we are willing to trade some loss of performance, as long as the performance degradation is not detrimental to the user experience, for improved security guarantees.

Threat model. As for the threat model, the attacker's main goal is to subvert the security properties of zones (i.e., P1-3 shown above in subgoal 1). We assume the adversary has already compromised the trusted OS of a given zone z_i, and aims to escape it (see Figure 21). Running in S.EL1^{i}, the attacker may attempt to access (read/write) external memory addresses outside the z_i's private memory (§6.2.2). These access requests may target i) the physical address space of the victim at NS.EL$_{1/0}$, EL3, or S.EL$^{j}_{1/0}$, or ii) the PPC or memory regions used by the ACU. In the first case, the objective is to override the PPC's memory-mapped registers containing the permissions that forbid access to the victim's memory regions. In the second case, the idea is to subvert the ACU's behavior, e.g., causing the ACU to hand over the control of the PPC to the processor core executing the attacker's code. This would indirectly allow the attacker to disable the PPC's restrictions. The attacker may also leverage the effect of caches to read or write cached memory cells for which S.EL1^{i} does not currently have permissions.

In this work, we do not consider software attacks exploiting vulnerabilities in the secure monitor or the ACU software. We assume that these components are correct and belong to the system's TCB. The platform's secure boot initializes the system to a known state. We do not consider physical attacks that tamper with hardware, e.g., fault injection. Our solution has the side-effect of providing cache side-channel protection between zones. However, microarchitectural side-channels are out of the scope of our work. Similar to a typical TrustZone deployment, we do not consider denial-of-service (DoS) attacks including those caused, for example, by a malicious trusted OS not relinquishing control to the normal world.

6.4 Design

Figure 23 represents the architecture of ReZone. Hardware-wise, our system relies on a typical TrustZone-enabled platform. For controlling memory access permissions, in addition to a TZASC controller, ReZone relies on a PPC hardware component. The PPC is dynamically configured to block secure world accesses from the processor based on the processor's bus master ID (MID_0). The PPC can be reconfigured only by a bus master, predefined at bootstrapping time. In ReZone, this bus master is the ACU (MID_1). ACU and processor can communicate with each other efficiently using a message queue (MQ) implemented by a hardware peripheral.

Software-wise, ReZone comprises the *secure monitor* and the *gatekeeper*. The former consists of standard secure monitor software (e.g., implemented by Arm Trusted Firmware) augmented with a ReZone-specific sub-component named *trampoline*. The secure monitor (and trampoline) run on the main processor core and the gatekeeper on the ACU; the PPC protects their private memory regions, which store security-sensitive context information. Taken together, trampoline and gatekeeper manage the execution of zones in the system. They ensure that each zone can access only a private physical memory address space assigned to the zone, and take care of all context-switching tasks involving zone entering and exiting operations. These operations occur when an REE application makes a call to a zone's guest TA (*zone entry*), and the TA returns the results of the call (*zone exit*). REE and zone can share data through a shared memory region. ReZone's software components are shipped with the platform firmware. When the system bootstraps, the firmware configures the memory layout and statically creates one or multiple zones indicating the composition of their respective software stacks, i.e., trusted OS and TAs. Next, we explain our design decisions and how the system works. We refer the reader to Figure 24.

6.4.1 Memory Partitioning and Permissions

The starting point in ReZone is to partition the physical memory layout into regions, allocating these regions to different protection domains of the system, and specifying regions' memory access permissions accordingly. In doing this, we are primarily concerned about restricting secure world accesses when the processor runs zone code, i.e., $S.EL_{1/0}$. The final memory layout and access permissions are

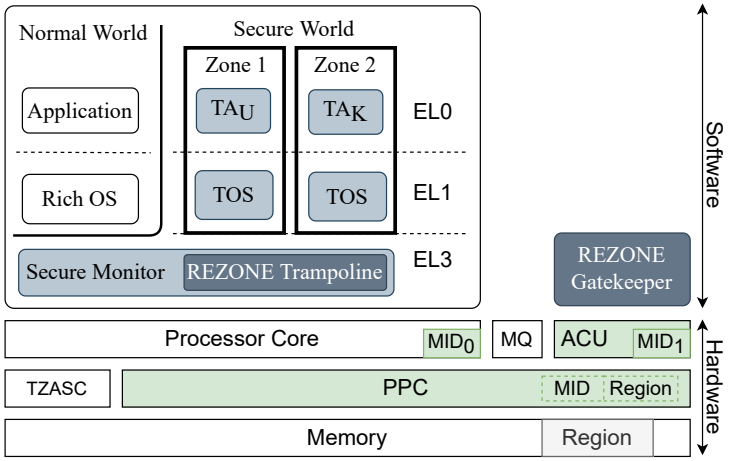


Figure 23: ReZone architecture: Its software components are colored in dark shade and consist of secure monitor (which includes the trampoline) and gatekeeper. Secure monitor runs on the main processor cores; gatekeeper runs on the ACU. ReZone software controls memory accesses via the PPC.

depicted in Figure 24. We progressively explain these regions and permissions, beginning with a vanilla TrustZone-enabled platform, and then extending it with ReZone's protections.

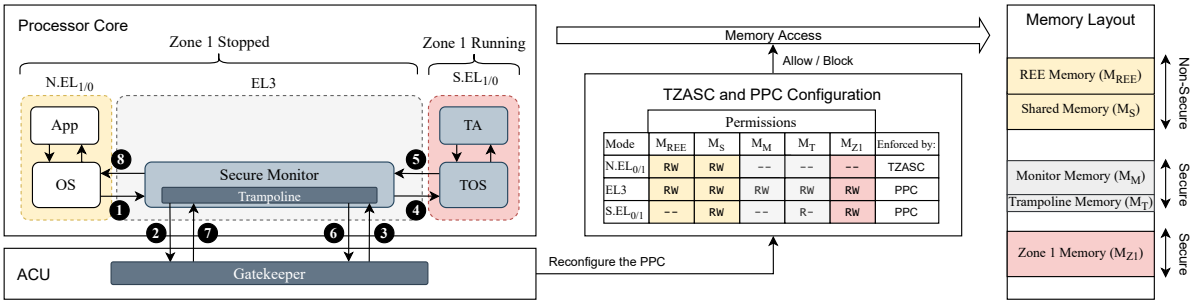


Figure 24: ReZone execution model, memory access permissions, and memory layout: ReZone ensures that the code running inside zone 1 – i.e., at $S.EL_{1/0}$ – can only access zone 1's private memory region (M_{Z1}). This protection is achieved by reconfiguring the PPC to restrict the processor's access permissions when entering zone 1 (steps 3&4), and reverting these configurations when exiting zone 1 (steps 5&6).

Just like in a typical TrustZone setup, in ReZone, the normal world includes regions for the REE OS (M_{REE}) and shared memory (M_{SH}). The secure world has independent regions for the TEE software. The TZASC is then configured to prevent the normal world software from accessing secure world regions. The first line in the permissions table (see Figure 24) shows how the normal world has only read/write access permissions to normal world memory regions.

Then, we further separate the secure world memory into independent regions for the secure monitor and for each zone i, e.g., zone 1 (M_{Z1}). We need to ensure that the secure monitor software (EL3) has full access permissions, but the code running at $S.EL_{1/0}$ is confined to (i) the zone's private memory M_{Z1} and (ii) the REE shared memory for TA calls. To this end, we leverage the PPC to restrict secure world accesses. However, using the PPC for this purpose is not trivial. Given that TrustZone does not mark memory accesses with the privilege level from which they originate (being tagged per world only) the PPC

cannot distinguish secure accesses coming from EL3 or from S.EL$_{1/0}$. As a result, PPC's permissions need to be dynamically changed by the monitor when entering a zone (i.e., reprogramming the PPC to remove RW access permissions to $\mathrm{M_{REE}}$ and to secure world memory as per line three in the permissions table) and when leaving a zone (i.e., reverting to the permissions in line two of the table).

However, a difficulty arises when exiting a zone. If the S.EL1 code running inside a zone is wholly prevented from accessing the secure monitor memory, this protection will still be in place by the time the zone's trusted OS traps into EL3; consequently, the secure monitor will be blocked by PCC resulting in memory access aborts and a system crash. To solve this problem, we create a dedicated secure memory region containing code and data that are necessary only to facilitate this transition securely. This code is part of the trampoline, which still belongs to the secure monitor because it runs in EL3. We then segregate a region of the trampoline's memory ($\mathrm{M_T}$) from the rest of the secure monitor memory ($\mathrm{M_M}$) because these regions need to be configured with different memory access permissions. $\mathrm{M_M}$ is always denied access from S.EL$_{1/0}$. In contrast, $\mathrm{M_T}$ is marked read-only by the PPC, therefore, allowing the processor to both execute trampoline code from $\mathrm{M_T}$ and load context data also from $\mathrm{M_T}$ while still running in S.EL1. The trampoline code will then engage the ACU to remove the PPC memory access restrictions to $\mathrm{M_M}$, thus allowing the secure monitor to freely run at S.EL3 and restore the normal world execution. These reasons justify the PPC permissions in the last two rows of the table in Figure 24.

6.4.2 Securing the PPC Memory Permissions

To guarantee the enforcement of the memory access permissions as described above, the PPC must be (re)configured exclusively by a trusted component. In ReZone, this component is the gatekeeper/ACU ensemble. We configure the PPC so that only the ACU is authorized to perform this operation. Since the PPC can differentiate requests based on the bus master ID, we program the PPC to accept configuration requests only by a bus master identified with the ACU's MID. Requests originating from other bus masters, e.g., the processor, will be denied. The ACU runs the gatekeeper code and dynamically updates the PPC permissions when entering or leaving a zone by serving the specific requests sent by the trampoline, e.g., to enable $\mathrm{M_M}$ RW permissions at zone exits.

ReZone also needs to defend against a potential attack. Processor and ACU communicate via a message queue unit (MQ in Figure 23) which is agnostic to the current exception level of the processor. As a result, an attacker running at S.EL1 in a zone could craft a request to disable the memory access restrictions enforced by the PPC. To thwart this threat, the gatekeeper must foretell that this request is malicious and therefore refuse it. To this end, trampoline in EL3 and gatekeeper share a secret token that is initialized at boot time and copied to their private address space. This token is then used by the trampoline for authenticating PPC reconfiguration requests sent to the gatekeeper. Since the secret token is not accessible at S.EL1, an attacker can no longer spoof legitimate requests, making the system robust against these attacks.

6.4.3 Preventing Cross-zone Interference

The mechanisms discussed so far secure all memory transactions observed outside the processor cluster according to the permissions matrix in Figure 24. These mechanisms require additional measures to safeguard against potential attacks within the processor itself. We address two concrete concerns.

First, the PPC acts only at the system bus level. Therefore, its memory access protections do not extend to the internal caches of the processor cluster. For this reason, if the caches contain data that a given zone is not authorized to read or write, and the core enters that particular zone, the content will be accessible to that zone, which constitutes a security violation. To prevent these problems, the cache hierarchy of the core executing the zone is flushed by the trampoline.

Second, the PPC cannot differentiate between individual cores within a cluster, because the PPC observes requests as being from a single bus master: the cluster's shared cache. As a result, ReZone's memory restrictions enforced by the PPC are collectively applied across all cores of the same cluster. However, blocking access to the normal world memory when a core enters a zone will also prevent other co-located cluster cores from accessing normal world memory, which may lead to crashes if they are currently executing REE code. To prevent this problem, upon zone entry, ReZone must halt other co-located cluster cores running in the normal world until the core exits the zone. Despite the non-negligible performance overheads potentially introduced by these measures, our experiments show that these overheads can be tolerated due to the infrequent TEE invocation patterns and short TA execution times inside the secure world (see §6.6), essentially trading some performance degradation for increased TEE security.

6.4.4 Zone Entry and Exit Workflows

The protection mechanisms described above act together whenever an REE application invokes a TA hosted in a zone.

A zone entry begins when the normal-world OS issues a request through the invocation of the smc instruction (1), causing the execution to be handed over to the secure monitor. If the request needs to be handled by the trusted OS, ReZone will first differentiate to which zone the requests to that particular TA have been statically assigned. Then, it forwards the request to the respective zone, i.e., zone 1 in Figure 24. For this, the secure monitor starts by notifying all other cores in that cluster. The other cores will then enter an idle state. The trampoline first flushes the core's and shared cache content and then continues by issuing a request (2) to the gatekeeper to configure the PPC to block access of the application core. To authenticate the request as coming from the secure monitor processor mode, the gatekeeper expects to receive the secret token, which was handed to the monitor during boot. After a successful reply from the gatekeeper (3), the trampoline will then (4) jump to the trusted OS.

A zone exit occurs once the trusted OS returns execution to the secure monitor (5). At this point, the trampoline issues a new request to the gatekeeper (6) to unblock secure memory accesses from the core. The gatekeeper ensures this by reconfiguring the PPC to disable the zone's protection policies, i.e., granting RW for all memory regions. The secure monitor will then (7) resume execution, notify other cluster cores

to, finally, return to the normal world context (8) executing under traditional TrustZone restrictions enforced by the TZASC.

6.5 Implementation

The implementation of ReZone is tightly dependent on the targeted hardware. We used the i.MX 8MQuad Evaluation Kit (EVK), which features an i.MX8MQ SoC by NXP. This SoC is powered by (i) one cluster of four Cortex-A53 application processors (@1.5 GHz) and (ii) one microcontroller, a Cortex-M4 (@266 MHz). The Cortex-A53 cluster has 32KiB level-1 (L1) instruction and data caches per core and a shared 2MiB level-2 (L2) cache. The Cortex-M4 has a 16KiB instruction cache and a 16KiB unified cache.

We use the Cortex-M4 to act as ReZone's ACU, and an SoC-embedded controller named RDC to play the role of PPC. This controller enables fine-grained control of access permissions through the creation of security domains. There are at most four domains, each identified by an ID (DID_i). A domain represents a collection of platform resources (e.g., memory regions, peripherals), and it is tagged with a set of per-resource access permissions (e.g., read/write, read-only). Bus masters, such as the processor cluster, microcontroller, DMA, etc., can be associated with one or multiple domains. A bus master in DID_i can only access a resource if the resource is also associated with the bus master's DID_i, and its access privileges will be determined by the permissions of DID_i.

Based on the security domain scheme provided by this PPC, we use the following insight to implement the memory partitioning and access permissions described in §6.4.1. Since the PPC is itself a platform resource, we need to reserve a domain (e.g., DID_1) with full read/write permissions to the PPC's registers, and associate the ACU as the single bus master to DID_1. Doing that, we guarantee no other bus master in the system will be able to tamper with PPC's security permissions. If DID_1 can be specified on the system prior to any tampering from an adversary, we ensure that the ACU remains in full and exclusive control of the PPC. To achieve this, we can rely on the secure boot to run a customized initialization routine (i.e., gatekeeper code) that will fully initialize the ACU before starting the zones' trusted OSes.

Our implementation configures two security domains when booting the system: DID_0, and DID_1. DID_0 contains all memory (and other) resources exposed to the cluster. DID_1 mainly contains the PPC and the gatekeeper's private memory region. The processor is associated with DID_0, and the ACU to DID_1. As a result, only the ACU can reconfigure the PPC as per the access permission matrix depicted in Figure 24 (last two rows). In the i.MX 8M family, Cortex-A clusters are assigned a unique master ID. This means that DID_0's permissions apply indistinguishably to every Cortex-A53 core of the cluster.

We need to establish a secure channel between the cluster and ACU. Upon zone entry/exit events, they exchange message requests (reconfiguration) via the platform's MQ (see Figure 23). In the i.MX8MQ SoC, the MQ is composed of MU_A and MU_B. The PPC is configured to assign MU_A exclusively to DID_0, and MU_B to DID_1, ensuring that only the processor and ACU can communicate with each other through the local MU interfaces. To authenticate the requests sent by the processor, trampoline and gatekeeper

share a secret token. To preserve the secrecy of the token, the trampoline stores it in the $\mathrm{TPIDR\_EL3}$ 64-bit register, which is exclusively available to EL3 code (not even S.EL1 code can read this register) leaving the attacker a 1 in 2^{64} chance of guessing its value correctly. The token is randomly generated at boot time.

Given that the ACU runs at a clock speed 5.6× slower than the cluster, we implemented an optimization to offload workload from the gatekeeper (ACU) to the trampoline (cluster). We observed that the reconfiguration of the PPC's permissions DDI_0 resources become slower as the number of resources grows. To shift most work to the trampoline, the gatekeeper temporarily associates the PPC to DDI_0, allowing the trampoline to update PPC's permissions on the gatekeeper's behalf. Once finished, gatekeeper removes PPC from DDI_0, reacquiring its former condition of sole guardian of the PPC.

6.5.1 Cross-core Synchronization

While one core is in a zone (secure world), the PPC blocks access from the whole cluster to several memory regions (e.g., normal world memory). Thus, specific actions need to be taken into account to prevent other cores from hanging and crashing. Below are the instances where such scenarios might occur (for the sake of simplicity we consider only two cores):

1. Two cores c_1 and c_2 execute secure monitor code at EL3, and c_1 issues a zone entry, requesting a shift to S.EL1 while c_2 still runs at EL3. This can be a problem because if c_1's request is attended without proper synchronization, c_2 will not be able to keep accessing monitor memory.

2. One core (c_1) is executing the trusted OS at S.EL1 while another (c_2) is sleeping. Meanwhile, the sleeping core c_2 awakens into monitor code (e.g. due to an interrupt). Again this is a problem because c_2 will not be able to access monitor memory while c_1 executes the trusted OS.

3. Two cores c_1 and c_2 are executing normal world code, and c_2 requires a service from a TA hosted inside a zone. This is a problem because, when c_2 enters the zone, c_1 will be denied access to normal world memory.

We solve this problem by synchronizing the cores using spin locks and IPI interrupts. To address scenario 1, the monitor tracks which cores are currently executing the secure monitor. If a core intends to enter a zone, they all wait on a barrier when leaving EL3. At that point, c_2 sits on a spin lock while c_1 is allowed to jump into the trusted OS. Once c_1 concludes the zone call and returns to EL3, c_2 can continue. We solve scenario 2 by waking the core c_2 into a trampoline's read-only memory region (not-writable by the running trusted OS) and waiting for the trusted OS in c_1 to cease execution by using a spin lock ($\mathrm{rz\_lock}$). To solve scenario 3, we use an IPI interrupt and a spin lock ($\mathrm{rz\_lock}$). First, when c_1 issues a request from normal world to enter a zone, it traps into the monitor (EL3) where it fires an IPI interrupt and grabs the $\mathrm{rz\_lock}$. The goal of this interrupt is to notify other cores (i.e., c_2) that c_1 wishes to enter a zone. Core c_2 is then interrupted and jumps into trampoline code, an exception handler running in EL3, where

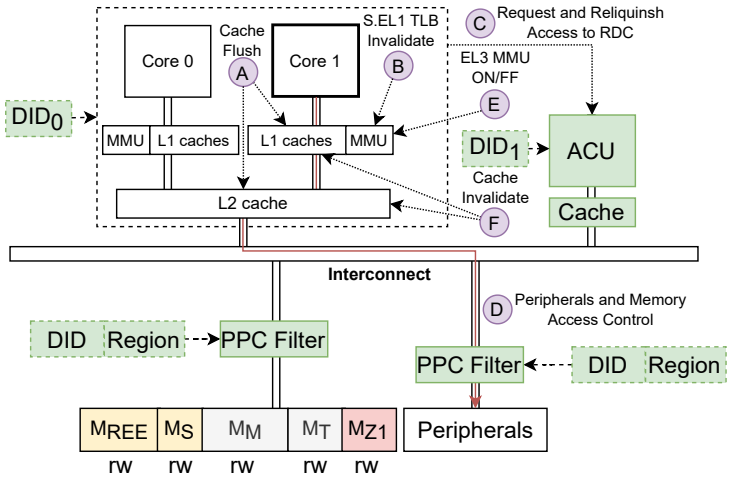


Figure 25: Context-switch between EL3-S.EL1 requires microarchitectural maintenance operations (A, B, E). Entering a zone (EL3⟶S.EL1) requires A, B, C, D and E. Exiting a zone (S.EL1⟶EL3) requires C, D, E and F.

it unlocks c_1 letting it continue to enter the zone; c_2 now waits on the spin lock until c_1 exits the zone, allowing c_2 to resume its execution in the normal world.

6.5.2 Microarchitectural Maintenance

Triggered by a zone entry/exit event, context switching between EL3-S.EL1 requires the trampoline to coordinate not only cross-core synchronization, but also perform important cleanup tasks (see Figure 25) and PPC reconfiguration operations. We now focus on microarchitectural maintenance that needs to be carried out, and then in §6.5.3 we describe how the trampoline dynamically reconfigures the PPC.

When entering a zone (EL3⟶S.EL1), the trampoline running on the core needs to ensure that the trusted OS (after the switch is finished) cannot access resources outside of the zone. After taking all measures to prevent interference from other cores (§6.5.1), the trampoline needs to invalidate the L1 instruction and data caches and the shared L2 cache (A). Unless this operation is performed, data or instructions left in the cached will be accessible to the trusted OS, thus violating zone isolation. Also, right before jumping into the trusted OS, the trampoline disables SMP coherency. Recall that at this point, all other cores in the cluster are suspended. Each core has data and instruction in its own L1 caches. With SMP coherency enabled, a request to access cached data would cause the coherency protocol to fetch the data from another core's L1, resulting in a cache hit; thus, data could be accessed by the present core, violating the zone's isolation properties.

In addition to cache maintenance, when switching between the trusted OSes of different zones, the trampoline needs to perform TLB maintenance (i.e., TLB invalidation) for the current core (B). The main reason is not due to concerns about violation of zone isolation, as this is already enforced by the PPC by setting each zone's memory region to be private. However, given that (i) the TLB for S.EL1 is shared by the trusted OSes of active zones, and (ii) the TLB has no way to tell which of the cached page table

entries belong to each zone (unless a TLB invalidation is performed), a TLB lookup by the trusted OS could translate into a physical address that belongs to another zone, thus causing unexpected faults. As an optimization, we perform TLB maintenance only when switching between different trusted OSes, and it does not affect TLB entries of the REE OS. With all microarchitectural maintenance done, the trampoline will restore the general-purpose register state and exit to the trusted OS.

6.5.3 Dynamic PPC Reconfiguration

At the center of the context switch operation between EL3-S.EL1 is the dynamic configuration of PPC's memory access permissions. PPC reconfiguration guarantees that the trusted OS cannot access resources unless they are attributed to its zone. Specifically, the trampoline reconfigures the PPC to prevent S.EL1 code running inside a zone from accessing resources belonging to normal world, monitor, and other zones.

The reconfiguration of PPC's permissions is executed by the trampoline in EL3 after performing micro-architectural maintenance, as doing so before would prevent normal world and monitor memory from being written back to main memory. To perform the reconfiguration, the trampoline must first obtain access to the PPC. This step is achieved by sending a request to the ACU with the secret token to authenticate the trampoline (step C in Figure 25). The ACU will then allow the application cores to access the PPC. At this point, only the trampoline can access the PPC. The trampoline can then perform the necessary PPC configurations (step D) as per the permissions matrix presented in Figure 24. After the PPC reconfiguration is concluded, the trampoline sends another request to the ACU. This time the ACU configures the PPC to prevent any accesses to the PPC by any bus master other than the ACU itself. Once the ACU replies to the trampoline, control can be safely handled over to the trusted OS.

6.5.4 Handling Exceptions at S.EL1 Exits

In contrast to zone entries, exiting a zone from S.EL1 to EL3 precludes steps A and B (see Figure 25). Flushing caches (A) is not necessary, since we selectively invalidate monitor memory in step F (explained below), thus preventing malicious cache manipulation to gain code execution privileges. TLB maintenance (B) is not required because S.EL1 TLB entries only affect S.EL1. However, we need special precautions in dealing with exceptions from S.EL1. Two problems may arise:

1. TLB-miss: Upon entering the vector table to handle an S.EL1 exception in EL3, if the TLB misses a page table entry that maps to the memory region of the exception handler, then the MMU must perform a page table walk. As the caches have been flushed before entering S.EL1, this operation will trigger an access to main memory to load the page table entry. However, since the EL3 page tables are stored in the secure monitor's memory region, whose access by EL3 code is still blocked by the PPC at this stage (recall only the trampoline memory is marked read-only), the code for handling the exception cannot be loaded and the system will hang.

2. EL3 code injection from S.EL1: From 1) it follows that the code on the EL3 vector table must be marked as read-only by the PPC, allowing also the trusted OS to read the handler code from memory. However, since the vector table code will be cached in the L1/L2 caches, write operations are still permitted to cache lines tagged as secure, as long as they do not reach the main memory (note the PPC acts only at the bus level). An attacker with S.EL1 privileges can use this feature to modify the exception handler code to its advantage, e.g., by modifying the SMC handler, an SMC call can trigger the secure monitor to load malware from the cache.

To solve both problems, we disable the MMU for EL3 code (E in Figure 25). We do this in the last steps of exiting from EL3 to S.EL1 when entering a zone. Note that this measure only applies to the MMU for EL3 code: we do not disable the MMU for S.EL1 code. This solves problem 1, as, from that point on, memory accesses at EL3 will not be subject to MMU translation; thus, no page table accesses are needed, and the exception handler can then be executed. Disabling the MMU will also disable L1 and L2 caches for EL3 address spaces, and this will partially solve problem 2 as the exception handling code will always be fetched from main memory. This memory is protected read-only by the PPC, hence no modifications will be possible; however, this solution does not entirely solve the whole problem. Note that during the switch from EL3 to S.EL1, after flushing all caches, the trampoline still executes with caches enabled. This will inevitably leave some traces of trampoline data and instructions in the caches, which a malicious trusted OS can try to leverage for an attack by modifying them locally while running in S.EL1 as described above. To clean up potentially tampered cache content and avoid these attacks, we invalidate on-cache trampoline-related memory when the exception handler in EL3 is executed on an S.EL1 exit (F in Figure 25). After this cache management operation completes, the exception can be handled securely.

6.5.5 Software Implementation

The ReZone software is a firmware bundle based on TF-A (v2.0). TF-A implements basic bootstrapping and secure monitor functionality. We modified TF-A to incorporate the trampoline, encompassing 303 and 1220 Source Lines of Code (SLOC) of C and assembly code, respectively. We also used the TF-A to bootstrap the gatekeeper. The gatekeeper is implemented as a bare-metal application that runs on the Cortex-M4. It was written in 417 SLOC of C code and packaged with essential drivers to interface with the RDC. The system image was compiled with the GNU Arm toolchain for the A-profile Architecture (v9.2.1) and GNU Arm Embedded Toolchain (v9.3.1).

We also changed the TF-A's context management routines and data structures to support two zones, each running a trusted OS instance based on OP-TEE (v3.7.0). To run OP-TEE inside a zone, no intrusive modifications were necessary; we have disabled only the support for dynamic shared memory. We create the second OP-TEE instance in a different address space and updated the values of SMC IDs to be uniquely attributed to each OP-TEE instance. We created two full-blown stack builds featuring different REE: one running the Linux kernel (5.4.24) and the second the Android OS (AOSP 10.0). It was not necessary

to modify REE code to support ReZone; however, since we added support to run multiple trusted OSes, we added drivers to interface with both of them. We also duplicated the OP-TEE Linux kernel driver and modified the SMC_IDs used to make calls to the second OP-TEE instance. There are two tee-supplicant services, one for each OP-TEE.

6.6 Performance Evaluation

We evaluated ReZone using the i.MX8 platform and software described in §6.5. On the i.MX8 platform, all cores, i.e., the Cortex-A53 APU cluster (4x) and the Cortex-M4, were running. Our performance evaluation covers three vectors:

1. Microbenchmarks. We evaluate the performance overheads at increasing levels of REE-TA interaction. Firstly, we measure the *world switch time*, i.e. the time to transition from normal world to the first instruction of the trusted OS. Secondly, we measure the *round-trip execution time of the GlobalPlatform TEE Client API* (v1.0), i.e., the standard API for an REE application to interact with the TEE. We tested 9 APIs. Lastly, we gauge the *execution time of OP-TEE xtest*, a suite of regression tests provided by OP-TEE to verify the correct implementation of the trusted OS and related APIs. The suite has a total of 95 tests, grouped into nine groups. The suite also has seven benchmarks to assess the performance of cryptographic- and storage-related operations.

2. Real-world applications. We evaluated two TAs: a Bitcoin wallet and a DRM player service. The open-source Bitcoin wallet TA [308] provides services, e.g., for creation and deletion of a master key, access the mnemonic for a master key, sign a transaction, and get the address of the wallet. The DRM player service is implemented with the open-source OP-TEE Clearkey plugin [309] and the media player ExoPlayer [310]. This service approximates to a security level 2 (L2) DRM compatible setup [9], where video content is split into subsamples and decrypted within the TEE, but processed in the normal world. The L2 DRM setup represents the worst-case scenario for performance since the process is split among the two worlds. We measured the (i) execution time for decrypting one subsample for three different resolutions and frame rates and (ii) the time elapsed between decoded content.

3. Impact on REE performance. We used PCMark for Android benchmark (v3.0), which assesses the performance of smartphones by testing everyday activities. The suite consists of five real applications (e.g., web browsing, data and video editing) and it does not invoke secure world functionality. The benchmark provides an overall score (Work 3.0) translating the performance of the whole system into a quantifiable scoring system: a higher score indicating better performance.

Methodology. To measure world switch time, we used the AArch64 generic timer, a 64-bit cycle counter running at 8.33MHz. For the GlobalPlatform API and xtest suite, we used the Linux time API with the CLOCK_REALTIME parameter enabled. For the Bitcoin wallet, we used the Linux time utility. For DRM service, we used (i) the Linux time API for the subsamples decryption and (ii) the Java System.nanoTime

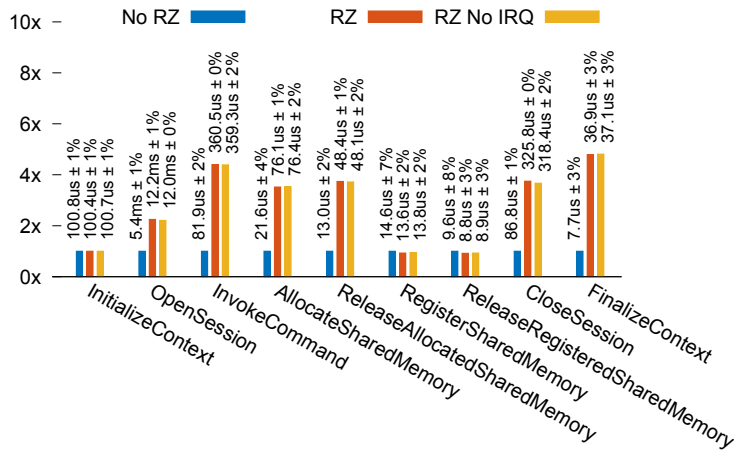


Figure 26: ReZone overhead, time, and normalized standard deviation (coefficient of variance) of GlobalPlatform API.

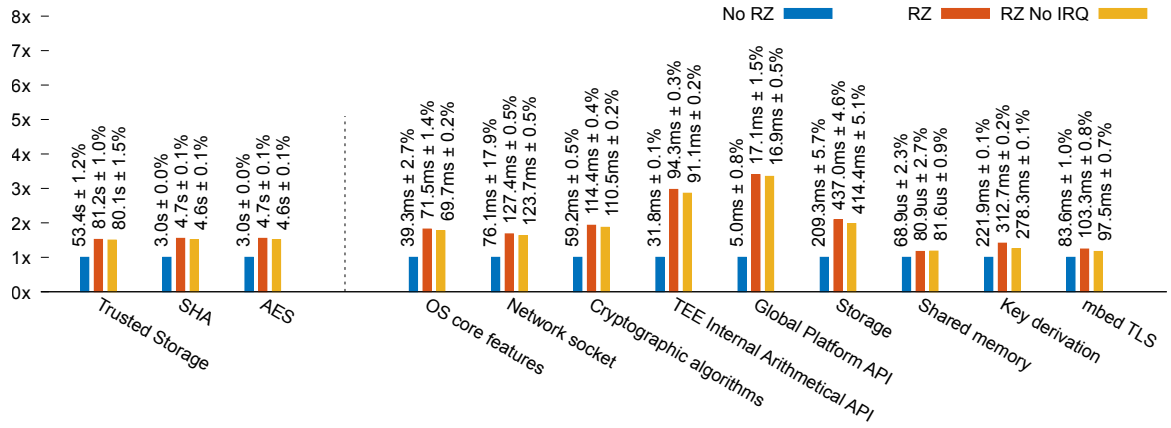


Figure 27: Geometric mean of ReZone overhead for xtest. Benchmarks (left) and unit tests (right).

for the processing. Excepting world switch measurements, for all other experiments, we compare three system configurations: (i) standard OP-TEE deployment (No RZ); (ii) ReZone deployment (RZ); and (iii) ReZone deployment with no trusted OS preemption from the REE (RZ no IRQ). We collected 1000 samples and present the average of values within the 95^{th} percentile, which removes the effect of occasional Linux interference. For microbenchmarks, we flush the caches at the end of each test. This reduces the hot-cache effect of the previous iteration and approximates a realistic TEE setup.

6.6.1 Microbenchmarks

We identified two main factors impacting the performance overhead, i.e., (i) the number of trusted OS calls and (ii) the complexity/duration of the operation running in the secure world (mainly the logic of the TA). Workloads encompassing simple TA operations/logic with a higher number of world switches will translate into significant performance overhead. In contrast, workloads encompassing complex TA operations with small number of world switches will translate into negligible performance overhead. Multiple zones support

has also a negligible impact on the overall performance of the system.

World switch time. The world switch time increased from an average of 5.7±0.33% to 72.5±1.7% microseconds, i.e., 13×. This increase is the cumulative penalty of (i) additional ReZone trampoline and gatekeeper execution time, (ii) cluster-microcontroller requests, (iii) penalty of running a portion of the trampoline with caches disabled, and (iv) time of cache maintenance operations. Despite the high overhead of raw world switches, they often translate into relatively small slowdowns in real-world TEE usage scenarios (see §6.6.2).

GlobalPlatform API. Figure 26 presents our main results. The geometric mean of the normalized performance overhead for the nine API benchmarks is 2.327×, i.e., a significant decrease compared to the 13× overhead from the world switch. We can observe that the three APIs (i.e., *OpenSession*, *InvokeCommand*, and *CloseSession*) that explicitly call the secure world have a larger penalty, ranging from 2.25× to 4.79×. The *InvokeCommand* and *CloseSession* APIs are the ones that take less time to execute, and thus the ones with the bigger overhead, due to the over-penalty of cache-related operations (see xtest evaluation). Third, from the remaining six APIs, two have almost no penalty. This fact is related to the OP-TEE configuration which disables dynamic shared memory. With this option, *Allocate* and *Register* API implementations are identical. While calling these APIs, there is no switch to the secure world. Therefore, after the execution of *Allocate* and *ReleaseAllocatedSharedMemory*, code and data are already cached for the *Register* and *ReleaseRegisteredSharedMemory* calls. Lastly, we can observe that the system configuration without preemption (RZ No IRQ) slightly decreases the overhead, reducing the geometric mean from 2.327× to 2.325×.

xtest suite. Figure 27 summarizes our results, represented by the geometric mean of all tests for the same group. Unit tests are represented in nine groups on the left, while benchmarks are represented in three groups on the right. A detailed view of the results per test/benchmark can also be found in the supplementary material published online [304]. The group of tests that has a bigger impact is the TEE Internal API and Global Platform API, i.e., 2.97× and 3.40×, respectively. Note that this group of tests are the ones with smaller execution times and simpler secure world operations. On the other hand, tests such as mbed TLS and key derivation perform fewer but longer operations in the secure world. Since the execution time of the operations performed in the secure world is large, the performance overhead is relatively lower when compared to the TEE API groups. To complete, we tested a potential application-level optimization. Specifically, we modified one particular unit test (4007 symmetric) which consists of performing different symmetric encryption schemes (e.g., AES, DES) for incremental key sizes (e.g., 64-, 128-bit). The standard implementation has a total of 3884 secure world calls per run. We modified the implementation to perform all iterations in a single TA call. By simply batching all operations and processing them in a one-time operation, we reduced the number of calls to the secure world to 96 and were able to decrease the performance overhead from 3.65× to 1.09×.

Multiple zones penalty. To assess the performance penalty of supporting multiple zones, we built

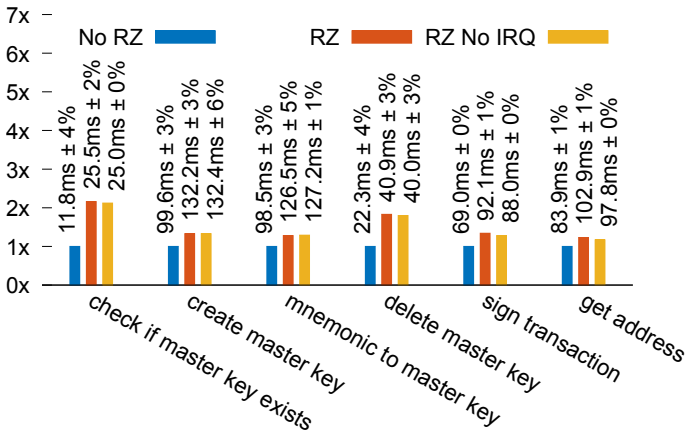


Figure 28: Relative, absolute execution time, and normalized standard deviation (over 100 runs) for Bitcoin wallet.

two specific test cases using xtest. The main difference, implementation-wise, to schedule a different zone, is the extra operation required to invalidate the TLB. Thus, in the first experiment, we measured the execution time of running xtest 4001 on trusted OS_2, while previously running also trusted OS_2. For the second test, we measured the execution time of running xtest 4001 on trusted OS_2, while previously running trusted OS_1. For the former, it takes, on average, 76.3ms to complete. For the latter, it takes, on average, 78.0ms to complete, i.e., a 1.7ms (2.2%) increase.

6.6.2 Real-world Applications

Our findings suggest that ReZone will not significantly affect the user experience in real-world applications.

In the case of the Bitcoin wallet, the geometric mean of the performance overhead is 1.49× for the vanilla ReZone implementation and 1.46× for a non-preemptible configuration (Figure 28). The most impacted wallet services are related to simple one-time operations. For instance, the absolute execution time of the operation checking if a master key exists is considerably smaller (11.8ms for the standard OP-TEE deployment) than the other services, and thus the penalty is higher. Notwithstanding, the results clearly demonstrate that the impact of ReZone in longer time-consuming operations, e.g., Bitcoin sign transaction (the one likely to be executed more frequently) is low (1.34×) and one order of magnitude smaller than the overhead observed in the raw world switch.

Regarding the DRM player, Figure 29 plots our results. Overall, the geometric mean of the normalized performance overhead for the DRM subsample decryption is 1.59×. For instance, Figure 29 (left) shows that the performance penalty of individual subsample decryption for 1920x1080@60 added by ReZone (red bar) in comparison to the bare TrustZone setup (blue bar) is 1.59×. This operation is performed every time the secure world is invoked to decrypt a single subsample and return to normal world. These per-subsample overheads tend to be dwarfed when we consider the total processing time that it takes to render a video (Figure 29, right). In this case, computing the time overheads for all configurations gives

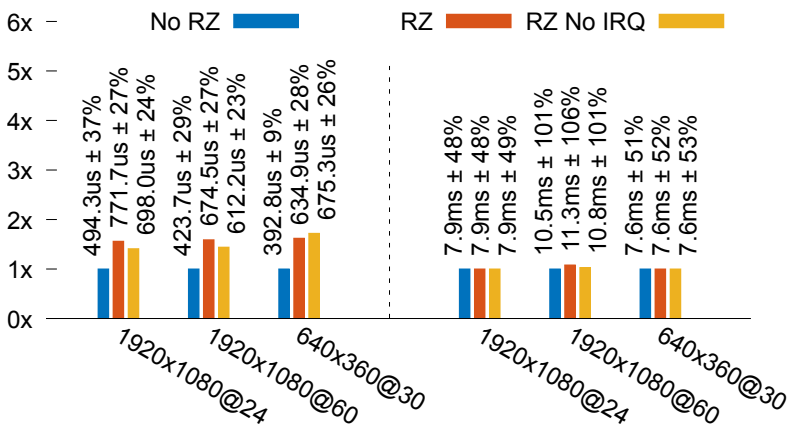


Figure 29: Performance overhead, execution time, and normalized standard deviation (over N runs) for DRM subsample decryption (left) and processing (right). N=1000, 1000, 400, for 1080@24, 1080@60, 360@30 setups, respectively.

time increases of 0.2%, 0.4%, and 7.7%, respectively, for 24, 30, and 60 frames per second. The higher the frame, the higher the overhead due to the larger number of trusted OS calls and the smaller duration of secure operations (smaller subsample size). Second, the ReZone configuration without preemption (RZ No IRQ) slightly improves the performance of the decryption process, reducing the geometric mean of the normalized performance overhead to 1.52×. This benefit is also noticeable in the processing of the decoded frames and the rendering of the final image (Figure 29, right). Third, although there is a slowdown in decryption, this penalty is in the order of 100 μseconds. Hence, it is not noticeable in the playback of the video and does not impact the user experience.

6.6.3 Impact on the REE Performance

We observed that the performance impact from ReZone on the REE is low, even when stressing secure world calls and the number of parallel calls from concurrent CPUs. We configured the REE to run PCMark (in the quad-core configuration), while concurrently running a toy TA at different intervals (10, 100, and 1000 ms). TA performs a simple addition and returns to normal world. We compared with a setup running vanilla OP-TEE and without calling the TA while running PCMark. Table 11, on the left, presents the results. For reasonable intervals between the toy TA calls (1000 and 100), the performance penalty is low, 0.74% to 4.86%, respectively. When the interval is 10 ms (similar to a DRM workload), the impact is reasonable (12.97%) and in the order of magnitude as the ones presented in §6.6.2. To understand the impact on REE for having multiple CPUs issuing concurrent TA calls, we fixed the interval rate of calling the toy TA in 1000ms. Table 11, on the right, shows that the impact is low and proportional to the number of concurrent cores, achieving the largest penalty (3.73%) when the four cores are issuing concurrent TA calls.

Memory usage. Given that ReZone needs to reserve memory for the secure monitor and for allocating zones where trusted OS stacks will be deployed, less memory becomes available to the REE. In our

Interval (ms)	Score	Penalty
10	4369	12.97%
100	4776	4.86%
1000	4983	0.74%

# of Cores	Score	Penalty
1	4983	0.74%
2	4938	1.63%
4	4833	3.73%

Table 11: Impact of ReZone on the PCMark performance score assessing the REE. On the left, the impact of decreasing the interval between calls for one core. On the right, the effect of issuing TA calls (1000 ms intervals) with multiple cores.

experiments, the secure monitor requires 192KB of memory, and the default OP-TEE configuration that we use for initializing a single zone requires only 32MB. We used this OP-TEE configuration for all our tests, including running the Bitcoin wallet and DRM player (see §6.6.2). Considering that our platform features 4GB of RAM, this means that the memory footprint of ReZone for hosting a single trusted OS is residual. Naturally, the amount of reserved memory can be increased in order to accommodate i) single trusted OS setups where TAs have higher memory demands, or ii) multiple trusted OS stacks enclosed inside independent zones.

6.7 Security Evaluation

6.7.1 Theoretical Security Analysis

To evaluate the security of ReZone, we first provide a theoretical assessment of how our system enforces the security properties P1-3 stated in §6.3. We discuss the main attacks that may be attempted to subvert said properties and highlight ReZone's mechanisms responsible for thwarting them:

A1. Direct memory mapping violations. As described in §6.2.2, an attacker controlling the trusted OS running in S.EL1 of a given zone, may attempt to directly map unauthorized memory regions pertaining to monitor, other zones, or normal world. We prevent these attacks using a PPC to restrict access to memory resources depending on whether a zone is executing (see Figure 24). To protect the normal world, we further need to overcome some limitations of the PPC (see §6.4.3) by having only one active core in the cluster while executing a zone, and performing cross-core synchronization (see §6.5.1).

A2. PPC hijacking. The adversary may attempt to circumvent these restrictions by controlling the PPC and disabling memory protections through reconfiguration of PPC's permissions. To prevent this attack, we ensure that only the trampoline can access the PPC by requiring the trampoline to authenticate itself before using the PPC (see §6.5.3). This authentication involves a shared secret token established at boot time. A malicious zone can submit a request to unlock the PPC; however, an attacker has a 1 in 2^{64} chance of guessing the token correctly.

A3. Unintended cache data leakage. Instead of trying to compromise the PPC through trampoline impersonation, the attacker may leverage cached data to gain access to the secret token. To mitigate this attack, we make sure that the token is only accessible to EL3. This is guaranteed by: i) flushing the

token from the caches, ii) having the PPC deny access to the token, and iii) storing the secret token in an exclusive EL3 register while a zone is executing, to allow the trampoline to authenticate itself after a zone exits. The adversary may also leverage cached data to access other memory content from monitor, other zones, or normal world. We perform cache flushes thus evicting sensitive cached data. Memory accesses will then result in direct accesses to main memory, which the PPC can control and block in case of permission violations.

A4. Cache code injection. The attacker can try to bypass the protections described above by tampering with the trampoline at runtime. Although direct access to monitor or gatekeeper memory is not possible, he may try to inject code into the trampoline by leveraging the fact that the PPC does not perform access control at cache level (see §6.5.4). To prevent this attack, we disable the MMU for EL3 before entering a zone, thus preventing the trampoline from fetching maliciously manipulated code, and data from being fetched upon a zone exit, allowing the trampoline to sanitize data before continuing.

A5. TCB tampering. Lastly, the adversary may try to disable the protections described above by tampering with ReZone's TCB, i.e., monitor, gatekeeper, and trampoline, before the system bootstraps, e.g., by replacing a legitimate binary image with another one. Secure boot prevents this attack by aborting the system boot sequence if firmware integrity checks fail.

6.7.2 CVE Mitigation Analysis

Lastly, beyond our theoretical security assessment, we aim to study how deploying ReZone in real-world systems could help mitigate privilege escalation attacks resulting from the exploitation of vulnerabilities located in TEEs. As part of this study, we leverage the database of TEE-related CVEs from our previous work [43]. We then analyzed 80 CVEs reported as critical and grouped them according to the affected component: trusted OS (TO), TA, cryptographic implementation (CI), hardware issue (HW), and bootloader (BL). CVEs that do not clearly identify the vulnerable component (i.e., trusted OS or TA) are labeled as TO/TA. We excluded four CVEs that lack enough information about the affected component. Table 12 summarizes our analysis.

Mitigation of attacks in scope. In total, we identified 66 CVEs that are in scope. These refer to vulnerabilities affecting the trusted OS or trusted applications that have the potential to be successfully exploited and may lead to privilege escalation attacks as indicated in Figure 20. In all these cases, ReZone can help counter successful exploits of these bugs by sandboxing the trusted OS inside a zone and preventing privilege escalation into normal world, secure monitor, or other TAs. In some cases, the attacker may be originally driven by a slightly different goal. For instance, CVE-2018-5210 describes a vulnerability that allows an attacker to gain TEE privilege execution, which leads to retrieving device unlocking information at the TA level. This can then be used to obtain the device unlocking code. We consider this vulnerability sandboxed because an attacker cannot affect other system components.

CVEs out of scope. In ten cases, CVEs refer to vulnerabilities that fall outside ReZone's scope, and

CVE	C	CVE	C	CVE	C	CVE	C
2014-9979	TO ✓	2017-11011	TO ✓	2015-9000	TA ✓	2015-8997	TO/TA ✓
2015-8999	TO ✓	2017-14912	TO ✓	2015-9002	TA ✓	2015-8998	TO/TA ✓
2015-9070	TO ✓	2017-14916	TO ✓	2015-9162	TA ✓	2015-9005	TO/TA ✓
2015-9071	TO ✓	2017-14917	TO ✓	2015-9174	TA ✓	2015-9007	TO/TA ✓
2015-9072	TO ✓	2017-17176	TO ✓	2015-9183	TA ✓	2016-2432	TO/TA ✓
2015-9073	TO ✓	2017-18071	TO ✓	2017-6293	TA ✓	2016-10297	TO/TA ✓
2015-9108	TO ✓	2017-18128	TO –	2017-18310	TA –	2017-6289	TO/TA ✓
2015-9112	TO ✓	2017-18129	TO ✓	2017-18312	TA ?	2017-14913	TO/TA ✓
2015-9113	TO ✓	2017-18132	TO ✓	2017-18317	TA ?	2017-18293	TO/TA ✓
2015-9198	TO ✓	2017-18133	TO ✓	2018-5210	TA ✓	2017-18296	TO/TA ?
2015-9199	TO –	2017-18311	TO ✓	2018-5885	TA ✓	2017-18297	TO/TA ✓
2015-9200	TO ✓	2017-18314	TO ?	2014-9932	TO/TA ✓	2017-18298	TO/TA ✓
2016-2431	TO ✓	2017-18315	TO ✓	2014-9935	TO/TA ✓	2018-5866	TO/TA ✓
2016-10238	TO ✓	2018-3588	TO ✓	2014-9936	TO/TA ✓	2017-18282	HW –
2016-10432	TO ✓	2018-5870	TO ✓	2014-9937	TO/TA ✓	2015-9003	CI –
2017-6290	TO ✓	2016-10239	TO ✓	2014-9945	TO/TA ✓	2016-10398	CI –
2017-6292	TO ✓	2018-11950	TO ✓	2014-9948	TO/TA ✓	2017-14907	CI –
2017-6294	TO ✓	2015-4422	TA ✓	2014-9949	TO/TA ✓	2017-18146	CI –
2017-8274	TO ✓	2015-6639	TA ✓	2015-8995	TO/TA ✓	2016-10458	BL –
2017-11010	TO ✓	2015-6647	TA ✓	2015-8996	TO/TA ✓	2017-14911	BL –

In Scope?	TO/TA	TO	TA	HW	CI	BL	Total	Percentage
Y	21	34	11				66	86.84%
N		2	1	1	4	2	10	13.16%
Total	21	36	12	1	4	2	76	

Table 12: Mitigation analysis of selected CVEs with critical CVSS scores: CVEs in scope (✓), out of scope (–), insufficient information (?).

		Security (Can defend against)			Scalability (Does not depend on)			Programming (Unmod. code)		Performance (Sources of overhead)			TCB (Subcomponents and total size)			
Enclave Type	System	TOS↮NW (P1)	TOS↮Mon (P2)	TOS_1↮TOS_2 (P3)	PPC	ACU	Non-COTS Features	Runtime	API	S2-TLB	Micro. Arch. Maintanance	Trap & Emul.	Monitor	Trusted OS	Other SLOC	Total kSLOC
NW	PrivateZone [296]	–	–	–	○	●	●	○	○	○	○	●	TF-A^+	OPTEE	-	259.9
	OSP [117]	–	–	–	○	●	●	○	○	○	○	●	TF-A^+	OPTEE	OSP Hyp	255.5
	vTZ [121]	–	–	–	○	●	●	●	●	○	●	○	Custom	-	-	2.0
	Sanctuary [116]	–	–	–	●	●	○	○	○	●	○	●	TF-A^+	OPTEE$^+$	TA 2x	256.0
	TrustICE [115]	–	–	–	●	●	●	○	○	●	○	●	TF-A^+	OPTEE	-	255.0
SW	TEEv [110]	◐	◐	◐	●	●	●	○	●	●	●	○	TF-A^+	-	TEE-Visor	28.8
	PrOS [111]	◐	◐	◐	○	●	●	○	●	●	●	○	TF-A^+	-	PVisor	27.6
	ReZone	●	●	●	○	○	●	●	●	●	○	●	TF-A^+	-	gatekeeper	30.0

Table 13: Analysis of ReZone and similar systems: ● indicates that the system fares positively, ○ that it fares negatively.

therefore our system offers limited defenses. CVE-2015-9199 only affects specifically shared memory regions. In CVE-2017-18310, a TA exposes too many services to the normal world, which may allow the normal world to perform abusive actions. ReZone is not meant to control which operations are available to the normal world. CVE-2017-18128 reports a bug in the configuration of a hardware component that can expose secret information. ReZone is not designed to not prevent the trusted OS from exposing secrets, e.g., to the normal world. In CVE-2017-18282, a hardware bug allows the normal world to perform secure data accesses; ReZone does not protect against hardware bugs. Cryptographic-related vulnerabilities such as CVE-2017-14911 are out of scope. ReZone relies on the correct implementation of cryptographic primitives and schemes. Bootloader-related vulnerabilities, CVE-2016-10458 for example, are also out of scope. ReZone relies on the platform's secure boot to correctly initialize the system.

6.8 ReZone in Perspective

6.8.1 ReZone and Related Systems

In this section, we put ReZone in perspective with closely related research on TrustZone-aware TEE systems. We surveyed seven representative projects that have been designed to improve the security of TEEs along different (and sometimes complementary) dimensions. Next, we provide an overview of these systems and then compare them with ReZone according to five distinct criteria. Table 13 guides our discussion.

Related TrustZone-aware TEE systems. One class of solutions aims to provide normal world enclaves using virtualization. Notable examples include vTZ [121], OSP [117], PrivateZone [296] which leverage normal world's virtualization extensions (NS.EL2) to create isolated environments in normal world. While vTZ virtualizes the full TrustZone hardware (enabling the execution of full-fledged trusted OSes in the normal world), OSP and PrivateZone provide a custom runtime environment per TA. Other systems like Sanctuary [116] and TrustICE [115] provide normal world enclaves using the TZASC. They rely on monitor and trusted OS to dynamically program the TZASC and create normal world enclaves isolated from the REE. A third class of systems aims to create software-enforced secure world enclaves. For instance, TEEv [110] and PrOS [111] aim to support multiple TEE stacks on Arm platforms where S.EL2 is not available. These solutions rely on a "hypervisor" running in S.EL1 (TEEv) or EL3 (PrOS) and perform binary instrumentation of the trusted OSes running in S.EL1 to guarantee isolation. In our work, we introduce a fourth category of systems aimed at creating hardware-enforced secure world enclaves. By relying on PPC/ACU hardware, ReZone creates secure world enclaves (i.e., zones) to effectively restrict S.EL1 privileges.

Security. To better understand the security properties offered by ReZone, we consider the threat model defined in §6.3. Assuming that an attacker manages to hijack the trusted OS running in S.EL1, we intend to analyze if the studied systems can prevent further privilege escalation attacks to normal world (P1), secure monitor (P2), or other trusted OSes (P3). For systems that create normal world enclaves, given that the trusted OS hosted by an enclave runs in NS.EL1 (not S.EL1), these attacks are out of scope. Focusing on systems designed to create secure world enclaves, TEEv and PrOS rely on same privilege isolation techniques (binary instrumentation) to shield trusted OS instances running in S.EL1. Although these systems can offer protection against said attacks, they require substantial manual engineering effort or the employment of binary instrumentation tools for removing privileged memory instructions from the trusted OS. In contrast, ReZone offers similar protections without the need to re-engineer existing trusted OSes, being able to secure unmodified TEE stacks.

Scalability. By scalability we mean to identify important deployability barriers on real-world platforms. We base our analysis on assessing system dependencies on specific hardware components, namely: PPC, ACU, or hardware unavailable on COTS platforms. The fewer these dependencies, the better a solution can scale. Most systems, including ReZone, depend on some PPC-like component, e.g., SMMU, to restrict accesses from system bus masters. ReZone also depends on an ACU. Sanctuary precludes PPC or ACU,

but relies on the unrealistic assumption that the core ID is propagated to the bus, i.e., this solution is not practical for real-world platforms.

Programming overheads. TrustZone-aware TEE systems may require modifications to standard TEE APIs (e.g., Global Platform) and/or runtime components, i.e., trusted OS and normal world software. Some systems require modifications of trusted OS [110, 116] or normal world hypervisor [121], or implementation of custom drivers [111, 115–117, 296]. Conversely, ReZone requires no modifications to TEE APIs or runtime, being able to function with existing TEE/REE stacks.

Performance overheads. Given the heterogeneity of the studied solutions (e.g., target different hardware) and the lack of standardized benchmarks for TrustZone-aware systems, it is difficult to quantitatively compare the performance of these systems. Alternatively, we conducted a qualitative analysis by identifying three types of expensive sources of overhead: TLB misses due to using stage-2 virtualization (S2-TLB), micro-architectural maintenance operations, and trap and emulation. ReZone does not leverage any hardware virtualization support and does not require trap and emulation. However, it relies on micro-architectural maintenance operations which may sensibly impact the system performance.

Trusted computing base. Different systems depend on various software components to function properly and enforce the security properties for which they were originally designed. In Table 13, we identify the components that pertain to the TCB of each system and indicate their respective sizes. We collect this information from the original papers and added reference sizes for TF-A (24,607 SLOC) and OP-TEE (230,094 SLOC). SLOC values were computed using the SLOCCount tool. Modifications to software are marked with "+". In ReZone, the TCB covers a TF-A version that includes the trampoline (1,523 SLOC) and the gatekeeper (3,822 SLOC). ReZone features a TCB size of about 30 kSLOC that approximates the TCB size of systems that achieve comparable security goals, i.e., TEEv and PrOS. Given that ReZone's gatekeeper leverages readily available board support package (BSP) code, we foresee that an optimized assembly implementation would reduce gatekeeper's size to just a few hundred SLOC.

6.8.2 ReZone and Armv8.4 S.EL2

So far, we have discussed how ReZone can reduce the excess of privileges of S.EL1 on Armv8-A platforms prior to the Armv8.4 release. In this specific release, Arm introduced hardware virtualization in the secure world, by extending the hardware architecture with S.EL2, i.e., a new protection mode for hosting a secure hypervisor. Sitting on top of the secure monitor, the secure hypervisor virtualizes the TEE stack by allowing multiple TEE instances to run in isolation. This means that, from releases Armv8.4 onward, S.EL2 seems to provide an alternative solution to implement ReZone's zones without the need for extra hardware components – PPC and ACU. By hosting TEE stacks inside independent guest virtual machines (VMs), S.EL2 apparently offers similar security properties as ReZone's preventing a compromised trusted OS from escaping its VM, therefore protecting normal world (P1), secure monitor (P2), and other TEE instances (P3).

However, despite the introduction of S.EL2, a fundamental violation of the principle of the least privilege still persists: *the secure hypervisor has unlimited privileges to arbitrarily map memory regions into the S.EL2 address space* – including memory pertaining to monitor and normal world. As a result, if an attacker manages to exploit a bug in the secure hypervisor and run code at S.EL2, Armv8.4 offers no defense mechanism that can further prevent the attacker from violating all properties P1-3 and controlling the entire system. Hijacking the secure hypervisor in such a way is not unrealistic given its relative complexity. For instance, we computed the TCB size of Hafnium [82], Arm's reference implementation for the secure hypervisor, and it features over 20 KSLoC. ReZone can still be leveraged in this setting to isolate the secure hypervisor thus fully guaranteeing P1 and P2. From a technical point of view, repurposing ReZone should not require major challenges. ReZone's trampoline can even be simplified as it only needs to shield a single S.EL2 component (akin to a single zone) and interpose context-switch events.

6.8.3 ReZone and Armv9 CCA

In Q2 2021, Arm introduced the Confidential Computing Architecture (CCA) whereby Armv9 CPUs are augmented with a so-called Real Management Extension (RME) [311]. RME allows ordinary software developers to instantiate a new type of isolation environments named *realms* where they can run application code. To support realms, RME extends the TrustZone architecture with two additional worlds: *root world* and *realm world*. The root world is exclusively dedicated to EL3 and it hosts the secure monitor. The realm world corresponds to a second independent instance of the secure world which is now dedicated to hosting realms. Realm world, secure world, and normal world can run code in EL0, EL1, and EL2 protection modes. In the realm world, EL2 can house a *realm manager* – equivalent to secure hypervisor – which in turn can instantiate multiple realms – i.e., confidential VMs.

Compared to Armv8, CCA introduces important security improvements. For one, the root world prevents access to EL3 memory from any other world. This architectural change effectively solves the excess of privileges of Armv8's S.EL2 or S.EL1 by preventing a compromised secure hypervisor or trusted OS from hijacking the secure monitor, therefore enforcing P2. Similar to Armv8, CCA can also guarantee P3 given that S.EL2 can virtualize S.EL1.

Notwithstanding, *the normal world can arbitrarily be accessed by S.EL2 in secure or realm worlds*. This excess of privileges opens the door for P1 violations due to a compromised realm manager or secure hypervisor. Given CCA's early stage, it is premature to make a definitive assessment of this technology. Nevertheless, our preliminary analysis leads us to infer that ReZone may be sided with RME to further protect the normal world. Moreover, given that *RME is an optional feature starting only from Armv9.2* [312], it is not yet clear how widely OEMs will adopt RME. As of early 2022, the only announced SoCs with Armv9 CPUs feature Armv9.0 [313], which omit RME. Without RME, CPUs lack root world protection for EL3 [314] thus having the same security limitations of Armv8.4 CPUs, which may justify deploying ReZone.

6.8.4 Discussion

ReZone limitations and optimizations. ReZone, has two central limitations. First (a), since the core ID is not propagated to the bus, only a single TA can simultaneously run per cluster (see §6.4.3), which may limit TA concurrency and occasionally increase TA response time. Second (b), frequent invocation of (small duration) TAs may lead to non-negligible performance overheads due to numerous world-switch calls. To improve (a), we propose to enforce a fair-cluster scheduling policy, which balances TA execution across clusters. Since multi-core platforms typically provide multi-cluster configurations (e.g., Qualcomm 8 series SoCs since 2016 [315, 316]), this optimization allows concurrently running as many TAs as the available clusters. To improve (b), we suggest batching requests from the rich OS and serving each batch through a single secure world call. This approach reduces the number of world transitions, and consequently the number and impact of ReZone's microarchitectural maintenance operations.

Effects of ReZone's appropriation of PPC/ACU. We deduce that ReZone can leverage a platform's PPC and ACU without jeopardizing the utilization of these components for their original purposes and legacy use cases. In particular, PPC/ACU can be shared, e.g., using a spare security domain in the PPC or enabling the co-existence of the gatekeeper code with other critical or non-critical functionalities already embedded in the ACU. Numerous works in the literature provide strong isolation for software running in microcontrollers [19, 317–320]. Modern low-end TEEs such as MultiZone [319] leverage minimal hardware primitives ready-available in almost all Arm Cortex-M microcontrollers to provide high-performing, robust, and lightweight enclaves.

Coexistence between PPC/ACU and TZASC. In our current design, ReZone still leverages the TZASC to protect the secure world from normal world accesses, such as in vanilla TrustZone deployments. Without the TZASC, CPU accesses cannot be differentiated from secure and non-secure. Thus, to protect the normal world, ReZone would need to perform additional expensive operations (e.g., cross-core synchronization, cluster suspension, cache maintenance operations, etc), with an extra restriction on lack of concurrency with the monitor. A notable exception is TLB maintenance, which would not be necessary given the existence of independent TLB entries for the normal world (that are not shared). In summary, leveraging the TZASC in tandem with the PPC/ACU improves performance while providing security guarantees between normal world and zones.

Using Arm SMMU as ReZone's PPC. To leverage an SMMU as PPC, we envision minor changes in the execution flow while entering (EL3→S.EL1) and exiting (S.EL1→EL3) a zone. Trampoline's dynamic access control configuration code (see §6.5.3) must explicitly create page tables that enforce the required access permissions for each zone, instead of configuring permissions through memory-mapped registers. The trampoline must also invalidate the SMMU TLB to ensure the configured page tables enter in effect. The ACU must also prevent the CPU from accessing the SMMU or the page tables otherwise a zone could undo the established access control policy and fully compromise the system.

6.9 Related Work

Several security-oriented hardware architectures provide underlying primitives for building TEE systems [19, 43, 321], namely CPU extensions [19, 84, 299], separate co-processors [25, 26], dedicated security chips [29], and secure virtualization [87, 311]. Recently, academia has been focused on exploring RISC-V to propose novel TEE hardware architectures [322, 323]. In our work, we leverage COTS hardware primitives to provide augmented isolation within the TEE.

As for hardware-enforced TEE systems, Komodo [101] implements a small TEE monitor which provides sealed storage and remote attestation per the SGX specification. Lightweight secure world runtimes [8, 103] aim at shielding security-critical applications from untrusted OSes. TrustICE [115] and Sanctuary [116] leverage the TZASC to create enclaves within the normal world. LTZVisor [47] and Voilá [19] have leveraged TrustZone hardware to virtualize system resources for a dual OS system configuration while addressing real-time guarantees. ReZone leverages hardware orthogonal to TrustZone to provide containerization for multiple trusted OSes, while providing the same availability and real-time guarantees as existing TrustZone-assisted TEEs. CaSe [105] and SecTEE [106] allow TAs to run entirely from the cache and on-chip memory, respectively. Keystone [245] and MultiZone [324] leverage standard hardware primitives from the RISC-V ISA, i.e. physical memory protection (PMP), to isolate individual enclaves.

vTZ [121], OSP [117], and PrivateZone [296] leverage the virtualization extensions available in the normal world (NS.EL2) to create isolated environments. TEEv [110] and PrOS [111] use same privilege isolation [325] to secure a minimalist hypervisor from trusted OSes, due to the lack of secure virtualization support prior to Armv8.4-A. Our solution does not rely on any hardware virtualization support and thus can be used in combination with existing hypervisors and legacy systems.

6.10 Conclusion

With ReZone, we present a new security architecture that can reduce the privileges of a TrustZone-assisted TEE by leveraging hardware primitives available in modern hardware platforms. We have implemented and evaluated ReZone for the i.MX 8MQuad EVK and the results demonstrated that the performance of applications such as DRM is not significantly affected. We have also surveyed 80 CVE reports and estimate that ReZone could help mitigate 86.84% of them.

7

AnyTEE: An Open and Interoperable Software Defined TEE Framework

Publication Data

D. Cerdeira, J. Martins, N. Santos, and S. Pinto. Not published yet.

AnyTEE: An Open and Interoperable Software Defined TEE Framework

David Cerdeira†, José Martins†, Nuno Santos‡, Sandro Pinto†

†Centro Algoritmi, Universidade do Minho

‡INESC-ID / Instituto Superior Técnico, Universidade de Lisboa

Abstract

TEEs are fundamental in securing a broad spectrum of applications and are prevalent on various platforms, from mobile and embedded devices to cloud servers. However, the existing landscape of TEE technologies is notably heterogeneous, posing substantial interoperability and compatibility challenges for application developers and system designers. These hurdles are not limited to well-established technologies like Arm TrustZone and Intel SGX, but also extend to newer generations of TEE models. In this context, we introduce AnyTEE, a new framework that allows application developers to use multiple TEE models in their applications and system designers to engineer customized TEE models across multiple ISAs. Essentially, AnyTEE harnesses widely available hardware virtualization primitives to facilitate the development of Software-Defined Trusted Execution Environments (sdTEEs). Our reference implementation of AnyTEE includes sdTEE versions of the SGX and TrustZone models for Arm and RISC-V, respectively, along with an sdTEE variant of SGX that incorporates fine-grained isolation mechanisms. Our evaluation shows that AnyTEE achieves near-native performance (<3% overhead).

7.1 Introduction

TEEs are a pivotal and continuously evolving technology for protecting security-sensitive software. Available on most Commercial Off-The-Shelf (COTS) platforms, TEEs rely on hardware-based primitives to isolate the execution of said software from the untrusted main operating system (or hypervisor). Well-established TEE technologies from major CPU vendors include Intel SGX [84], Arm TrustZone [19], and AMD SEV [87], and target a wide spectrum of applications ranging from cloud servers and desktops, to mobile and embedded / Internet of Things (IoT) devices [19, 54, 84]. Further, steamed by the ever-growing push for confidential computing, new TEE-enabling technologies have recently emerged, such as Intel TDX [23], Arm CCA [22], and RISC-V CoVE [24].

However, system designers and integrators are today hampered by significant interoperability and compatibility challenges due to proprietary features and peculiarities of TEE technologies. In fact, to cope with stronger threat models and deliver richer functionality while providing a competitive market advantage, TEE vendors have developed independent solutions, resulting in heterogeneous TEE enforcement mechanisms and protection models. For example, while TrustZone mirrors the CPU privilege levels, Intel SGX

preserves the same privilege level while providing additional protections that prevent arbitrary access to the TEE by system-level software. In practice, these differences translate into separate software development niches for TrustZone and SGX, preventing software reusability between these communities and forcing developers to be proficient in both TEE-enabling technologies. Unfortunately, this fragmentation of TEE software development niches is expected only to grow as the next generation of TEE hardware also exhibits its specific traits. For instance, while SGX and TrustZone offer fine-grained protection at the application logic, Intel TDX and Arm CCA enable coarser isolation units spanning entire virtual machines (VM).

To cope with TEE heterogeneity, researchers have proposed to emulate specific TEE technologies on platforms that do not natively support them [101, 114, 243, 244]. For example, while Komodo [101] mimics the SGX protection model on Arm platforms, HyperEnclave [243] provides SGX compatibility on AMD servers. These works, however, do not allow the coexistence of different TEE programming models (e.g., emulated SGX and TrustZone environments) nor they allow the development of novel or customizable TEEs. Another approach to dealing with TEE heterogeneity consists of developing elementary hardware abstractions for building customizable TEEs. Keystone [245] pioneered this concept by providing such abstractions atop raw hardware security primitives, i.e., the RISC-V physical memory protection (PMP). The PMP is present in all RISC-V computers (is part of the ISA) and is a per-CPU protection unit with a limited number of configurable memory regions (typically 16). However, since PMP is only available in RISC-V, Keystone cannot not directly work on other ISAs, hampering the porting effort cross-platforms. Furthermore, Keystone does not provide seamless compatibility with existing TEE programming models, e.g., precluding the integration of legacy Arm TrustZone applications.

In this paper, we complement these existing approaches, and make the case for *Software Defined TEEs*, i.e., a TEE emulation and customization paradigm that simultaneously offers three fundamental properties: (1) *interoperability*, (2) *extensibility*, and (3) *portability*. Firstly, Software Defined TEEs should provide interoperability among TEE solutions (e.g., application- and VM-level), while enabling developers to use multiple TEE models simultaneously. Secondly, they should provide extensible primitives and building blocks, enabling developers to implement their own custom TEEs tailored to specific requirements. Finally, it should be possible to run legacy stacks (binaries, applications, and APIs) across various platforms and architectures. To the best of our knowledge, no existing solution has achieved this so far.

To materialize this concept, we propose AnyTEE, an open-source framework that leverages the widely available hardware virtualization primitives existing in well-established computing platforms (e.g., Arm VE, Intel-VT, RISC-V Hypervisor extension) to create Software Defined TEEs (sdTEEs). sdTEEs consist of (virtual) isolated execution environments linked to an associated context and set of resources (e.g., memory, interrupts) that can be hierarchically related. To this end, AnyTEE introduces the concept of *hierarchical TEE execution modeling*, i.e., a *parent* sdTEE oversees a *child* sdTEE, facilitating, for example, the ability to mimic the vanilla TrustZone model, where the secure world has higher privileges (and control) the normal world. This design enhances composability, nesting, and/or customization, facilitating the support of legacy, modern, and future TEE models. TEE models for confidential computing can be replicated by resorting to nested virtualization techniques, and new sdTEEs conceived featuring additional fine-grained

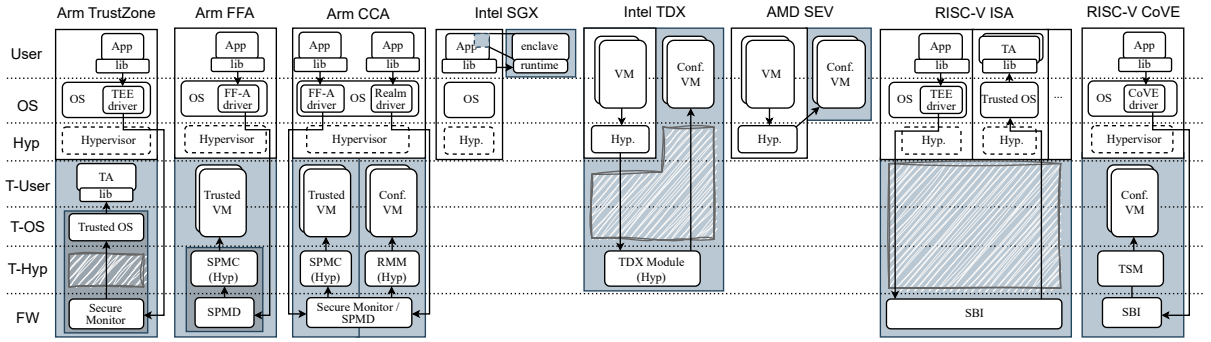


Figure 30: Software stacks and privilege levels of various TEE technologies. Inpired by related work [43], we depict privileges vertically and in ascending order, with the lowest privilege at the top. **TrustZone**: the OS in the normal world requests services to the TEE in the secure world through the secure monitor. **FFA**: a trusted hypervisor is added to manage trusted VMs. **CCA**: the Realm world is added into TrustZone, hosting the Realm Manager (RMM) which instantiates confidential VMs. **SGX**: enclaves are tied to a process and are protected from all other system software. **TDX** and **CoVE**: a trusted hypervisor creates confidential VMs with resources donated by the untrusted Hypervisor and performs context switching. **SEV**: an untrusted hypervisor manages VMs, which can either start up fully protected or select which memory is protected. **RISC-V ISA**: the physical memory protection (PMP) is used to create security domains, with SBI managing context switching.

security properties (e.g., intra-privilege isolation) by manipulating page tables.

We developed a reference implementation of AnyTEE for (i) the SGX protection model (sdSGX) on a COTS Arm platform (NXP i.MX8QM) and for (ii) the TrustZone protection model (sdTZ) on a RISC-V SoC running on an FPGA. In both cases, we run legacy TAs without modifying the source code, e.g., we seamlessly run a TrustZone Bitcoin wallet TA on a RISC-V machine. We also support the simultaneously execution of sdTZ and sdSGX TEE models. Furthermore, our implementation demonstrates how to augment the SGX and TrustZone protection models, for example, by restricting the memory access privileges of the Trusted OS running in the sdTZ [44]. We also implement fine-grained security enhancements by enhancing sdSGX enclaves with rapid permission context-switching, akin to VMFUNC [326]. We evaluated AnyTEE using microbenchmarks and legacy real-world applications. Our evaluation shows that, in general, our system achieves near-native performance (average less than 3%), mainly due to memory virtualization translation at the hardware level. We will open source our work.

Contributions. In summary, with AnyTEE, we make the following contributions:

- Conceptualization of *Software Defined TEEs* (sdTEEs) with a focus on three essential requirements: interoperability, extensibility, and portability.
- *AnyTEE framework design*, outlining the building blocks that enable the support of existing and novel TEE models on platforms with hardware-supported virtualization.
- *Open-source reference implementation* for SGX (sdSGX) and TrustZone (sdTZ) protection models, accommodating diverse ISAs such as Armv8-A and RISC-V. Additionally, an sdTEE-based intra-environment isolation mechanism.

- Comprehensive *evaluation and benchmarking of AnyTEE* across SGX and TrustZone TEE models, and *theoretical security analysis*, highlighting how AnyTEE mitigates potential attacks from various vectors.

7.2 Background and Overview

7.2.1 Heterogeneity of TEE Technologies

A common denominator between existing TEE technologies lies in their ability to protect the execution state of security-sensitive software, isolating it from an untrusted host operating system or hypervisor. Yet, they possess unique features that make them incompatible with each other. To better understand their primary sources of incompatibility, in this section, we briefly review some representative TEE technologies.

Representative TEE technologies. As illustrated in Figure 30, major hardware manufacturers have developed various TEE technologies. Arm is one of the major players, creating Arm TrustZone (TZ) [19], which is widely deployed in numerous mobile and embedded/IoT devices. Evolving from TZ, Arm Firmware Framework for Arm (FFA) [225] and Confidential Computing Architecture (CCA) [22] ensued. From Intel, we find Software Guard Extensions (SGX) [84] for desktop and server platforms, and Trust Domain Extensions (TDX) [23] for cloud and data center environments. AMD engineered their own TEE technology for the cloud, namely Secure Encrypted Virtualization (SEV) [87]. Importantly, RISC-V also includes primitives for building TEE software stacks in its baseline specification (ISA) [89] and is now standardizing the Confidential Virtual Machine Extension (CoVE) [23].

Diversity of TEE protection models. Conceptually, TEE technologies offer different protection abstractions, which are then reflected into the programming and usage models presented to software developers. One approach aims to protect a specific piece of application logic, typically a *trusted function*, that provides a well-defined interface for invocation by the unprotected application component. This trusted function is referred to as a TA in Arm TrustZone or RISC-V ISA, or as an enclave program in Intel SGX. This model is particularly well-suited for desktops, mobile devices, and low-end hardware. A second approach offers a coarser level of protection by enabling the isolation of an entire VM, commonly known as a *confidential VM*. As illustrated in Figure 30, confidential VMs are supported by Arm FFA, Arm CCA, Intel TDX, AMD SEV, and RISC-V CoVE. This model is a crucial component for enabling confidential computing.

Diversity of TEE enforcement mechanisms. For protecting TEE memory and execution state, TEE technologies typically leverage CPU privilege levels. One way consists of introducing *new trusted CPU privilege levels* to protect the TEE. For instance, Arm TrustZone mirrors the CPU privilege levels of the untrusted environment, where the operating system or hypervisor runs, into a trusted environment where TAs execute. The former is called the normal world, and the latter, the secure world. The OS must perform a secure context switch to transition from the normal world, execute the TA with user-level privileges in the secure world, and then transition back to the normal world once the TA execution finishes. Two trusted software components are necessary to facilitate this process, both hosted in the secure world: the

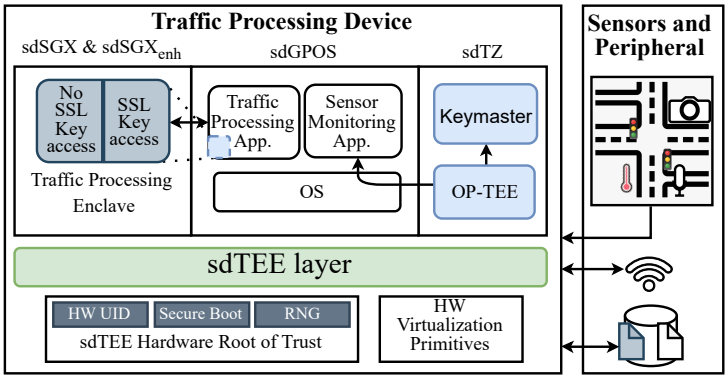


Figure 31: Usage scenario: A smart traffic analysis system using performing traffic data processing within an enclave, and using OPTEE TA keymaster for storage cryptographic key management.

secure monitor and the trusted OS. Subsequent Arm TEE technologies largely build upon the trusted CPU privilege levels introduced by Arm TZ [44]. In contrast, other TEE technologies *preserve the same CPU privilege level layout but enhance it with additional protections* that prevent arbitrary access to the TEE by system-level software. Notably, Intel SGX allows the creation of user-level TEE domains, called enclaves, which prevent the host operating system or hypervisor from inspecting or modifying their internal state, although they retain their privilege to manage the memory allocation of the enclaves. Only a small trusted library needs to be linked as part of the enclave program to enable context switch. Similarly, AMD SEV adopts a similar approach for protecting confidential VMs at the OS-level privilege level.

Other TEE features. In addition to isolated execution, TEEs tend to include two additional features: (1) *verifiable launch*, which involves the ability to generate evidence that the TEE is running unaltered software through the use of trusted boot and remote attestation protocols; and (2) *trusted I/O*, enabling TEEs to access external trusted devices, including secure storage, random number generators, cryptographic engines, and more. Memory encryption is an optional feature for protecting against physical attacks, such as cold boot attacks or bus snooping. It is especially relevant in cloud deployments, hence its adoption into SGX, SEV, and TDX. TrustZone-based platforms typically lack encrypted memory. For a comprehensive analysis of these features, we refer the reader to [54].

7.2.2 Overview of Software Defined TEEs

To enable multiple and various TEE protection models on the same host, we introduce the notion of Software Defined TEE (sdTEE), a trusted execution environment that can be configured by software to implement a specific TEE protection model. For instance, an sdTEE can emulate a pre-existing TEE technology, e.g., SGX's enclaves, or implement an exploratory TEE model such as the SGX enclave variant proposed by Gu et al. [327]. This variant supports intra-enclave isolation akin to Intel MPK's memory domains.

System model. As shown in Figure 31, sdTEEs harness two main hardware capabilities from the platform:

(*1*) virtualization to manage the resources and security protections of the TEE protection model, and (*2*) a minimal set of hardware primitives to provide a root of trust for the system. The figure illustrates the system model of an sdTEE usage scenario applied to a hypothetical smart city application. This use case leverages an embedded device equipped with sensors and cameras to collect and process real-time traffic data. The sdTEE (virtualization) layer implements the core mechanisms for supporting multiple sdTEEs. In this case: sdTZ, which emulates Arm TrustZone; and $\text{sdSGX}_{\text{enh}}$ which emulates SGX extended with intra-domain partitioning (see §7.4.3). The OS runs its own protection domain identified as sdGPOS.

Motivating applications. This device runs two TEE-powered applications, but each application uses a different sdTEE model. The traffic processing app«lication is required to handle security-sensitive information, such as license plate numbers and vehicle locations, and store it in a database. Given the sensitivity of this data, it must be processed inside a TEE. For this task, the developer chose $\text{sdSGX}_{\text{enh}}$. This choice was driven by the existence of a legacy SGX port of a legacy image processing library. It was also necessary to use an SGX-enabled SQLite database for secure data storage, and an SGX version of OpenSSL for secure data transmission. To further isolate the cryptographic keys used by OpenSSL from other software components, the developer allocated them into two separate intra-enclave domains. This ensures that the cryptographic keys are available only when SGX OpenSSL needs to send or receive data over the socket. On the other hand, the sensor monitoring app measures air quality, temperature, and noise levels. Although this data is not security-sensitive, it must be encrypted before being stored in the database, just like the data from the traffic processing app. To manage the encryption keys for both applications, the developers opted for the OP-TEE keymaster TA, creating distinct sets of keys for encrypting the encryption keys of both databases. This requires the instantiation of an sdTZ TEE, pre-configured with OP-TEE and the keymaster TA.

Benefits of software defined TEEs. As illustrated by this scenario, sdTEEs aim to offer the following benefits:

1. Interoperability. Allow applications to benefit from different TEE models co-existing simultaneously on the same platform, e.g., while the traffic processing app accesses $\text{sdSGX}_{\text{enh}}$, the sensor monitoring app leverages sdTZ.

2. Extensibility. As illustrated with the $\text{sdSGX}_{\text{enh}}$ example, developers can use sdTEEs to implement their own custom TEEs, tailored to their specific requirements.

3. Portability. Ease of transitioning TEE software stacks between various platforms and architectures (e.g., the legacy SGX port of the image processing library).

7.3 Design

AnyTEE is our proposed solution to offer sdTEE support on commodity platforms. In this section, we

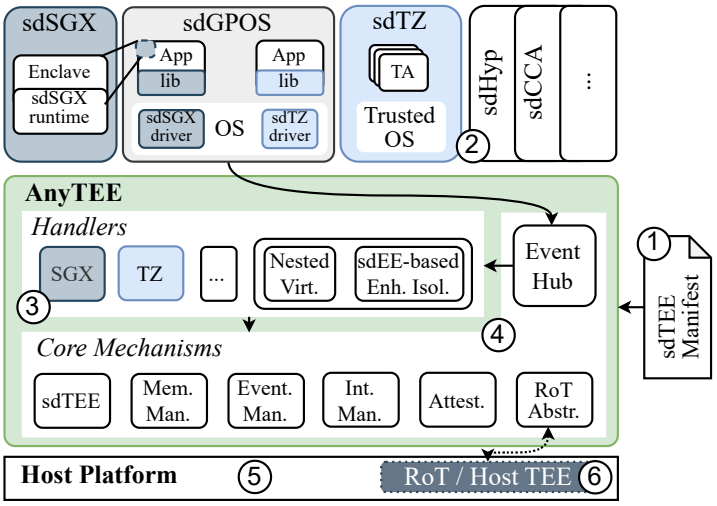


Figure 32: AnyTEE architecture, with multiple sdTEE types.

presents the design of AnyTEE, its components, and the approach to create cross-platform, legacy-compatible, and configurable TEEs. Figure 32 presents AnyTEE architecture and its sdTEE interface. In our system, sdTEEs are defined through an *sdTEE Manifest* ① and host TEE software ②. sdTEEs are managed by AnyTEE *Handlers* ③, which use the internal core mechanisms ④ as building blocks. AnyTEE executes on a host platform ⑤, that may feature its own host TEE or a Root-of-Trust ⑥, which AnyTEE leverages to provide features such as monotonic counters, secure storage, and true random numbers.

In AnyTEE, an attacker's goal is to violate the access control policy, either to compromise sdTEEs or other parts of the system. We assume AnyTEE implementation is correct and that it is securely integrated into the platform's boot process, which includes secure loading of AnyTEE and sdTEE Manifest. The sdTEE software stacks are not trusted. Side-channel, micro-architectural, or physical attacks are out of our scope, as mitigations to these attacks are orthogonal to our work.

7.3.1 Software Defined TEEs (sdTEEs)

Through sdTEE configuration and management, AnyTEE allows the creation of TEE programming models. Handlers oversee sdTEE execution, and integrate into AnyTEE as plugins written in C programming language that handle sdTEE requests and events implementing TEE functionality.

sdTEE access control and state. sdTEEs comprise an execution context with access to memory (including MMIO), interrupts (configured through the interrupt controller), and execution priority relative to other sdTEEs. sdTEEs execution context is a VM with multiple virtual CPUs (vCPUs), one per physical CPU. The execution context includes CPU registers, system registers, nested page tables, and a virtual interrupt control state. Nested page tables are used to enforce sdTEE's permissions over the platform's memory and MMIO, enabling granular access control. sdTEEs may need access to interrupts to respond to asynchronous events, such as timer and asynchronous device events, and in this cases sdTEE's state includes a virtual interrupt controller. sdTEE execution priority determines execution precedence among

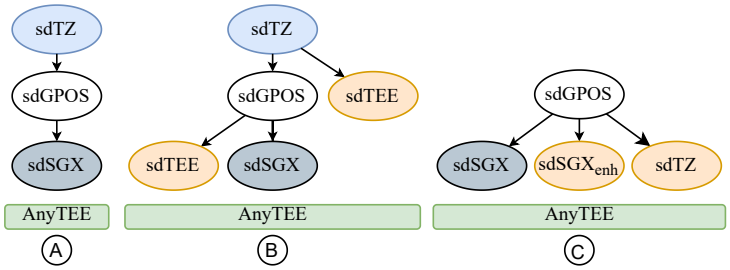


Figure 33: Hierarchical sdTEEs enable TEE composability (A), nesting (B), and customization (C).

sdTEEs. For instance, in TrustZone, the secure world can preempt normal world execution by setting interrupts as secure, resulting in higher priorities interrupt handling. On the other hand, in SGX, enclave execution priority is lower than the GPOS's, allowing the GPOS to take control of the enclave.

Hierarchical modeling of TEE execution. AnyTEE introduces a hierarchical modeling approach for TEEs based on an execution control hierarchy. This approach categorizes controlling partitions as *parents*, while controlled partitions are identified as *children*. In TrustZone, for instance, the secure world serves as the parent of the normal world. This approach allows for the concurrent execution of multiple TEEs, exemplified in Figure 33 (A), where sdSGX is controlled by sdSGPOS, which is, in turn, controlled by sdTZ. Furthermore, our modeling approach supports nested TEEs, as depicted in Figure 33 (B). This enhances system compartmentalization and security by adhering to the least privilege principle. For example, it enables the execution of a TEE with exclusive resource access that sdTZ cannot reach. The hierarchical view of TEEs also allows a reconsideration of the TEE execution model, as shown in Figure 33 (C). This facilitates novel configurations, such as a system running a TrustZone environment under sdGPOS control, preventing sdTZ from taking control of sdGPOS execution thus improving sdGPOS availability. Additionally, it can be used to enhance the properties of sdSGX enclaves, for example, to ensure minimum enclave execution times.

sdTEEs configuration. The configuration of each sdTEE access to platform memory, MMIO, interrupts, and its position in the execution hierarchy is done through a file named the sdTEE Manifest. For instance, establishing a TrustZone TEE involves configuring the manifest with two instantiated sdTEEs: one for the normal world (sdGPOS) and one for the secure world (sdTZ). The sdTZ manifest is defined first, modeling sdTZ as the parent of sdGPOS. Using the partition field, sdTZ is granted access to all memory, MMIO, and interrupts except for AnyTEE's private memory. The trusted OS binary is set as the manifest payload. The main handler is the sdTZ handler. Subsequently, the sdGPOS manifest is created as sdTZ's child. sdGPOS's partition fully overlaps with sdTZ but excludes resources exclusive to sdTZ. The payload for sdGPOS is set as the normal world bootloader (e.g., u-boot), responsible for booting the GPOS. The main handler for sdGPOS is the sdGPOS handler, and sdGPOS can make requests to the sdTZ handler.

Module	API	Mechanisms	Module	API	Mechanisms
sdTEE	create	Create an sdTEE using an sdTEE manifest	Mem. Man.	add_mem_region	Add memory region to an sdTEE dynamically
sdTEE	destroy	Destroy an sdTEE	Mem. Man.	rm_mem_region	Remove memory region from an sdTEE dynamically
Event	sub_fw	register callback on sdTEE firmware calls	Mem. Man.	add_dev_region	Add MMIO region to an sdTEE dynamically
Event	sub_hvc	register callback on sdTEE hypervisor calls	Mem. Man.	rm_dev_region	Remove MMIO region from an sdTEE dynamically
Event	sub_abort	register callback on sdTEE access faults	Mem. Man.	mem_translate	Translate an sdTEE address to a physical address
Event	sub_mem_region	register callback on sdTEE access to region	Int. Man.	inject_interrupt	Inject interrupt into sdTEE vCPU
Event	sub_interrupts	register callback on sdTEE interrupt events	Exec. Man.	push	Add sdTEE vCPU to the execution stack
Attest.	get_quote	Obtain an sdTEE attestation quote	Exec. Man.	pop	Remove sdTEE vCPU from the execution stack

Table 14: AnyTEE main API for Handlers.

7.3.2 Support for TEE Models

AnyTEE employs a modular approach to emulate TEEs, by introducing *sdTEE Handlers*. All sdTEEs require a main handler to deliver interrupts and handle events according to TEE-specific logic. Handlers integrate with AnyTEE by using AnyTEE's core mechanisms and event-based system, similar to XMHF [328]. During sdTEE setup, handlers register callbacks to events for sdTEE instances. When events occur or a request is made for an sdTEE instance, AnyTEE iterates over a list of the registered callbacks and invokes them iteratively allowing each handler to react appropriately. The sdTEE manifest field *accessible handlers* controls which requests are allowed for a each sdTEE. Handlers can register to handle events and requests by using the sub_* AnyTEE functions.

sdTEE execution management. sdTEE execution is managed through a FIFO data structure, i.e., stack, where each invocation of a child sdTEE (sdTEE$_C$) involves pushing the sdTEE onto the execution stack using push. Upon completing the requested operations, the child sdTEE performs a pop operation to return execution to its parent (sdTEE$_P$). In case of an interrupt, depending on priority, the stack may need to be unwound to sdTEE$_P$. Resuming execution of sdTEE$_C$ requires sdTEE$_P$ to issue a call to AnyTEE, that results in the sdTEE$_P$ handler performing a push operation.

Full-featured hypervisor support. AnyTEE operates in hypervisor mode, posing challenges in supporting software stacks featuring hypervisors. To overcome this, we introduce a hypervisor sdTEE (sdHyp) capable of hosting and creating guest VM sdTEEs (sdHyp$_{VM}$) as shown in Figure 34 (A). Hypervisor support is achieved through nested virtualization techniques [329, 330], which can be implemented through para-virtualization, trap-and-emulation, or leveraging hardware support. Integration of sdHyp with other TEEs is possible and illustrated in Figure 34 (B), for example, having sdTZ controlling the execution of sdHyp, and each sdHyp$_{VM}$ hosting sdSGX enclaves. Integration with confidential computing TEEs, e.g., CoVE, is shown in Figure 34 (C). CoVE features a Trusted Security Monitor (TSM) which creates and manages confidential VMs. The TSM is represented by sdTSM and controlled by sdHyp. sdHyp invokes the sdTSM to interact with confidential VMs (cVMs).

sdTEE-based security enhancements. By leveraging nested page tables, AnyTEE can leverage typical OS-level privileges and augment them to create flexible security primitives for applications. We propose three: 1) intra-privilege isolation, 2) read-only and write-once memory regions, and 3) immutable page

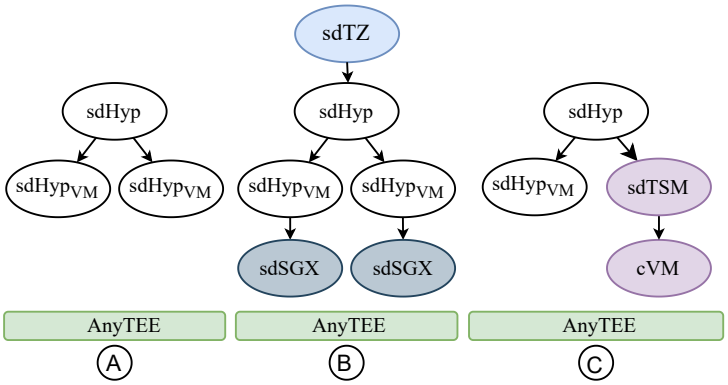


Figure 34: AnyTEE support for Hypervisor and TEE integration.

tables. *1) Intra-privilege isolation* establishes different permissions based on the current execution context, to restrict access to secrets based on what part of the application is executing. *2) Read-only and write-once memory regions* enables setting memory regions as read-only to the sdTEE, even if the first stage page table maps that memory as writable or disables virtual memory. The application must issue AnyTEE requests to mark pages as no longer writable. This mechanism can used to maintain immutable log records. *3) Immutable page tables* renders the sdTEE page table memory region immutable and prevents the sdTEE from setting a new root page table. AnyTEE designates the page table memory region as read-only, inhibits modifications to the register holding the root page table address, and ensures that any writable memory region is not executable. This approach ensures no code injections are possible.

7.3.3 AnyTEE Core Mechanisms

The AnyTEE core mechanisms consist of building blocks for TEE handlers, providing abstractions for sdTEE creation, management, attestation, and access to the host TEE on the platform. AnyTEE provides an event-based system to allow handlers to react to sdTEE events, including hypervisor and firmware calls, interrupts, and aborts. Table 14 summarizes the main interfaces the mechanisms expose.

Core. sdTEEs can either be created at boot time or at run time. AnyTEE initialization routines will invoke create to create sdTEE. Handlers use create and $\mathrm{destroy}$ to dynamically create and destroy sdTEEs. create takes as input an sdTEE Manifest, while $\mathrm{destroy}$ takes an sdTEE UUID.

Memory management. Spatial isolation is critical to prevent unauthorized access to TEE memory. TEEs have distinct requirements, ranging from fine-grain granularity (e.g., 4 kiB pages) to coarse-grain memory regions (e.g., 16 configurable memory regions for the entire system). Additionally, support for run time modifications of the access control may also be necessary. To achieve this AnyTEE resorts to nested page tables, and provides $\mathrm{add\_mem\_region}$ and $\mathrm{remove\_mem\_region}$ functions. Memory management also supports exclusive IO device assignment to sdTEEs. To control direct memory accesses (DMA) by IO devices, memory management relies on platform protection mechanisms such as IOMMU or IOMPU. A challenge arises for sdTEEs with non-continuous physical address spaces (e.g., SGX, VM-based TEEs). In

these cases AnyTEE leverages the IOMMU. In other cases, AnyTEE may be unable to support IO binding. To transfer resources between sdTEEs it is necessary for a handler to know the physical address of a resource. *Handlers* must use the $\mathrm{mem\_translate}$ function to obtain the physical address corresponding to an sdTEE address.

Event manager. AnyTEE provides a mechanism for handlers to subscribe to sdTEE events. Subscribing to firmware calls is possible using the $\mathrm{sub\_firmware}$, allowing handlers to react to requests with little or no modifications to existing software. Subscribing to hypervisor calls, $\mathrm{sub\_hypercall}$, allows handlers to establish their own hypercall interface. For example, to emulate SGX operations at the request of the GPOS. Furthermore, by subscribing accesses to specific memory regions, $\mathrm{sub\_mem\_region}$, handlers can emulate IO devices. Events such as memory aborts can also be subscribed.

Interrupt management. AnyTEE receives all system interrupts. A handler subscribes to its sdTEE interrupts through $\mathrm{sub\_interrupts}$. It also subscribes interrupts of other sdTEEs, using the same function, e.g., to handle context switching between sdTEEs. Interrupts are then injected on the sdTEE by its main handler using $\mathrm{inject\_interrupt}$.

Attestation. To provide sdTEE attestation, we resort to an attestation evidence list that includes platform and sdTEE code UUIDs, software versions, resources attributed to the sdTEE, and a cryptographic hash of the sdTEE payload. Handlers leverage AnyTEE's attestation API, $\mathrm{get\_quote}$, to create attestation evidence. AnyTEE takes advantage of the host platforms' TEE or RoT to create hardware-bound public/private key pair, to store the private key, and to digitally sign the evidence list. The evidence list is then provided to a remote party, which will use it to decide whether to trust the sdTEE. Similar attestation approaches have already been proposed [243].

Host TEE interface. In some TEE models (e.g., TrustZone), primitives such as random number generation, secure storage, and cryptographic accelerators, may be hardwired to the platform's TEE. AnyTEE can provide these same features to sdTEEs through the Host TEE abstraction. For example, on an Arm TrustZone-enabled platform, access to the replay-protected memory block (RPMB) needs to be mediated via a TA responsible for issuing secure storage requests. Alternatively, AnyTEE can use external RoT such as TPMs. To support unmodified binaries it may be necessary for AnyTEE to emulate security related IO devices that offer these features.

7.4 Implementation

In this section, we describe the implementation details of AnyTEE to support (i) SGX and (ii) TrustZone TEE models, as well as (iii) sdTEE-based security enhancement. We implement sdSGX on NXP's i.MX8MQ as Armv8-A target, sdTZ on both the i.MX8MQ and Digilent's Genesys 2 FPGA with a RISC-V CVA6 bitstream [331, 332], and sdTEE-based security enhancement on i.MX8MQ. In terms of software, we used the (Armv8-A) TF-A version 2.7, (RISC-V) OpenSBI 1.1, OPTEE version 3.21, and Linux 5.11.

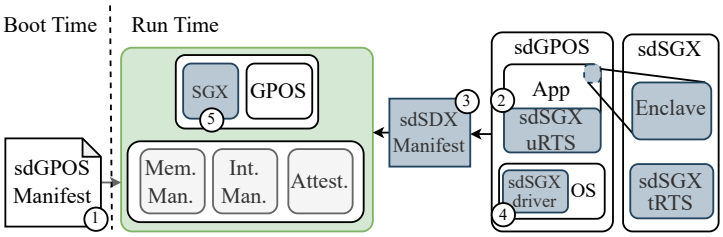


Figure 35: AnyTEE SGX implementation, sdSGX, including boot and run-time aspects, as well as sdSGX software.

7.4.1 sdSGX

In this section, we present our SGX TEE model (i.e., sdSGX) implementation for AnyTEE. Figure 35 depicts its architecture within AnyTEE. We also describe how sdSGX augments SGX security properties and extend SGX's programming model.

Software components. sdSGX implementation relies on a modified version of the SGX SDK and a custom SGX kernel driver. The untrusted runtime system (uRTS) was modified, creating sdSGX uRTS, to interact with our sdSGX kernel module. These modifications replace SGX function with calls to SGX handler. We focus on the key functions: $\mathrm{sgx\_create\_enclave}$, $\mathrm{sgx\_destroy\_enclave}$ and $\mathrm{sgx\_ecall}$, showcasing AnyTEE core features while not aiming for a complete SGX emulation. The kernel module is then responsible for interacting with the SGX handler. We built a unikernel runtime environment to support enclave execution in an sdTEE, based on the SGX SDK's trusted runtime system (tRTS), creating sdSGX tRTS. sdSGX tRTS interacts with the SGX handler, performs enclave initialization, and manages invocation of enclave functions and calls to the application.

Execution flow. Figure 35 illustrates the execution flow. AnyTEE loads an sdTEE Manifest to create an untrusted OS sdTEE (sdGPOS) during boot ①. The process of creating an sdSGX enclave involves creating the sdTEE manifest ②, loading it into memory ③, and allocating the enclave memory regions. The application then requests AnyTEE SGX driver to create an enclave using the manifest ④. AnyTEE delivers requests to SGX handler ⑤, which authenticates the manifest, creates the sdTEE, and initializes it, setting the sdSGX as a child of sdGPOS. Ecalls and Ocalls allow applications to invoke enclave operations and vice-versa. In sdSGX, Ecalls are executed by requests to the sdSGX handler, through hypervisor calls, triggering context switches to the corresponding enclave. Ocalls require enclave access to the application's stack pointer to allocate Ocall arguments but otherwise operate similarly to Ecalls. In SGX, GPOS handles enclave exceptions and faults. In AnyTEE, the SGX handler handles interrupts and memory access faults, and signals such events to sdGPOS. The sdGPOS can then resume enclave execution through a resume call to AnyTEE SGX handler.

Augmented SGX. Side-channel protection (e.g., through cache and DRAM bank coloring) can be used to improve confidentiality. Some side-channel attacks can be minimized [333, 334] by removing sdGPOS' ability to map and unmap enclave memory regions at will and preventing granular enclave interrupts. sdSGX can improve availability by allowing prioritizing enclave execution or allocating custom time windows

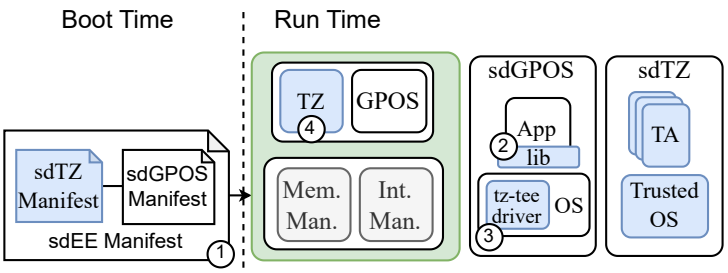


Figure 36: AnyTEE TrustZone implementation, sdTZ, including boot and run-time aspects, as well as sdTZ software.

regardless of OS scheduling policies. We can also improve application confidentiality and integrity by preventing the enclave from having arbitrary access to the application's address space. Performance-wise, as sdSGX does not feature memory encryption, it improves native enclaves' performance. Additionally, there are no hard limits for memory region size available to sdSGX enclaves. sdSGX can also benefit from exclusive device assignment, enabling reduced IO-related overhead compared to original SGX enclaves.

Compatibility and limitation. Our implementation adheres to the SGX programming model, allowing the execution of legacy enclaves without modifying architecture-independent enclave source code. However, the complete implementation tRTS, uRTS, enclave standard library code, and SGX Platform Software (PSW) has not been undertaken due to the effort required for support. Notably, multi-threaded enclaves are not supported in our current implementation. It's worth mentioning that sdSGX necessitates the pinning of application page tables to prevent swap operations. Given the lack of hardware encrypted memory, sdSGX also features a weaker threat model than SGX, e.g., an attacker with bus snooping capability could access the sdSGX's enclave state. However, AnyTEE could leverage memory encryption engines or on-chip RAMs and SoC bound execution to achieve this property [106, 107, 245].

7.4.2 sdTZ

We now discuss AnyTEE implementation of sdTZ, illustrated in Figure 36, including support for OP-TEE Trusted OS in RISC-V. Additionally, we explore enhancements to address some of TrustZone's architectural limitations [43].

Software components. AnyTEE supports OP-TEE Trusted OS on RISC-V, building upon existing porting efforts[335]. Some implementation details differ from the Arm architecture. For example, our OP-TEE RISC-V port doesn't support registering callbacks to different events, e.g., standard call (i.e., interruptible call) and fast call (i.e., uninterruptible call). Instead, in our implementation, the trusted OS distinguishes between events and invokes the appropriate handler. The TrustZone handler implements the context-switching logic to support communication between untrusted OS and OP-TEE. We modify OPTEE's Linux kernel driver to perform RISC-V SBI calls instead of Arm SMCs. The AnyTEE handler then handles SBI calls. For exclusive access to MMIO devices, AnyTEE exclusively maps the MMIO region and binds associated interrupts to sdTZ.

Execution flow. We instantiate two sdTEEs at boot time following TrustZone's boot flow, booting sdTZ software (secure world) before sdGPOS (normal world). The sdTEE manifest establishes resources belonging to each world ①. AnyTEE requires secure integration into the platform's secure boot. A request for TrustZone typically originates from an application ②. The application interacts with the trusted OS driver ③, which will then invoke the TrustZone handler ④, through secure monitor calls (SMC). The TrustZone handler then performs a context switch to the trusted OS ⑤. Our OP-TEE RISC-V port does not handle receiving interrupts meant for the GPOS. To address this, the *TZ handler* identifies such interrupts and forwards the event to the *GPOS handler*.

Augmented TrustZone. An important security limitation in TrustZone is the Trusted OS's access privileges [43, 44]. The Trusted OS can access any normal and secure world resource, while the normal world can only access normal world resources. With AnyTEE it is possible to restrict the memory access privileges of Trusted OSes running in sdTZs, preventing them from arbitrarily accessing resources of the sdGPOS sdTEE. Finally, AnyTEE allows for the instantiation of multiple TrustZone-TEEs. Several works have targeted this topic [44, 110, 111, 121], with FFA on Armv8.4 also enabling this.

Compatibility and limitations. AnyTEE currently only supports OP-TEE Trusted OS. Other Trusted OSes may require direct support. We executed OPTEE's xtest (refer to §7.5.2), demonstrating wide compatibility.

7.4.3 Intra-sdTEE Isolation

Recent CPU generations introduced mechanisms like Intel's Memory Protection Keys (MPK), nested page table switching with VM functions (VMFUNC), and Arm's Memory Tagging Extensions (MTE), for memory isolation [326]. In our project, we enhance sdSGX enclaves with rapid permission context-switching, akin to VMFUNC. This allows efficient management of access to sensitive information during execution addressing potential security risks in enclave implementations. Proposals to modify hardware for integrating isolation primitives into enclaves have been already been explored [327].

The VMFUNC instruction on recent Intel processors enables VMs to modify the second stage page table root. Hypervisor-configured root pointers are stored in a table, allowing VM guests to select a specific root pointer. Invoking the VMFUNC instruction modifies the Guest Physical Addresses to Host Physical Addresses mappings by switching root pointers, allowing VMs to establish mappings that enable access to sensitive information only when needed, and efficiently grant and revoke access to it during execution. In AnyTEE we implement an approach similar to VMFUNC, but that is implemented by software running at the OS level.

Key insight. Our implementation starts from the observation that it is possible to efficiently switch address space, i.e., domains in this context, with one write to root page table address register, TTBR0_EL1 in Armv8. An alternate approach would require updating page tables on every domain switch requiring page table modification and TLB invalidation. The key optimization to avoid page table modifications and TLB invalidation relies on pairing page tables and ASIDs, such that each Address Space Identifier (ASID) has

	Creation	Destruction	Context Switch
Average	82.25 ms	212.40 ms	17.14 µs
Std. Dev.	0.83 ms	3.81 ms	1.71 µs
Coeff. of Var.	1.01%	1.79%	12.37%

Table 15: Microbenchmarks for creation, destruction, and context switch operations, for an enclave with a binary size of 8.3 MiB.

a one-to-one relation with a page table (and vice-versa), and leveraging non-global page table property. Marking pages as non-global sets page table as valid only for the respective ASID. Each page table and ASID pair forms an execution context with different access permissions. This allows the TLB to select the correct page table entry for each domain. Not leveraging ASIDs and relying only on page table modifications can lead to incorrect memory access permissions. Cached page table entries in the TLB may result in using outdated entries, as the TLB deems them valid for the requested address translation, despite the page table switch.

Establishing domains. Each ASID represents a combination of permissions for each domain. For a setup with one domain where A is Accessible, W is Writable and * is Any, the following permissions are possible: $D_1\{AW\}$, $D_1\{A-\}$, $D_1\{-*\}$. To create this domain, three ASID and page table pairs are required. Supporting all combinations of N number of domains requires 3^N page table and ASID pairs. For 8-bit ASIDs, all combinations of 5 domains are supported, requiring 3^5 (243) page tables. For 16-bit ASIDs, all combinations of 10 domains require 3^{10} $(59,049)$ page tables. The number of page tables grows exponentially with N. It is up to the sdTEE developer to evaluate the trade-offs in speed in execution and number of page tables, deemed memory utilization.

7.5 Evaluation

We evaluate AnyTEE on the platforms cited previously (§7.4), covering microbenchmarks and real-world applications[1].

7.5.1 sdSGX

For the **microbenchmarks**, we measure three key micro-operations: i) enclave creation, ii) enclave destruction, and iii) context-switching. The enclave under test was configured with a memory region of 8.3 MiB (enclave binary size). Then, we evaluate AnyTEE impact on CPU-intensive workloads by running nbench-SGX [243, 327]. Next, we measure the execution time of read operations of an application implementing an SSL server, using SGX-OpenSSL and native versions. Client requests originate from co-located client applications running on the same OS instance. At the **application level** we create an application that

[1]We validated the execution of simultaneous TEEs. However, for the sake of simplicity we evaluate sdTZ and sdSGX independently.

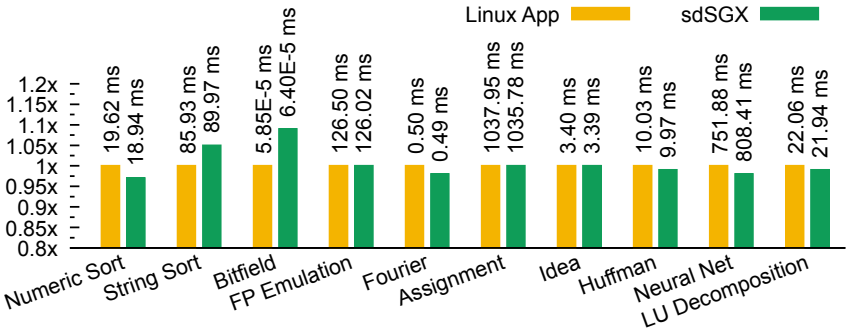


Figure 37: Relative and absolute execution time for nbench benchmark workloads for Linux App and sdSGX.

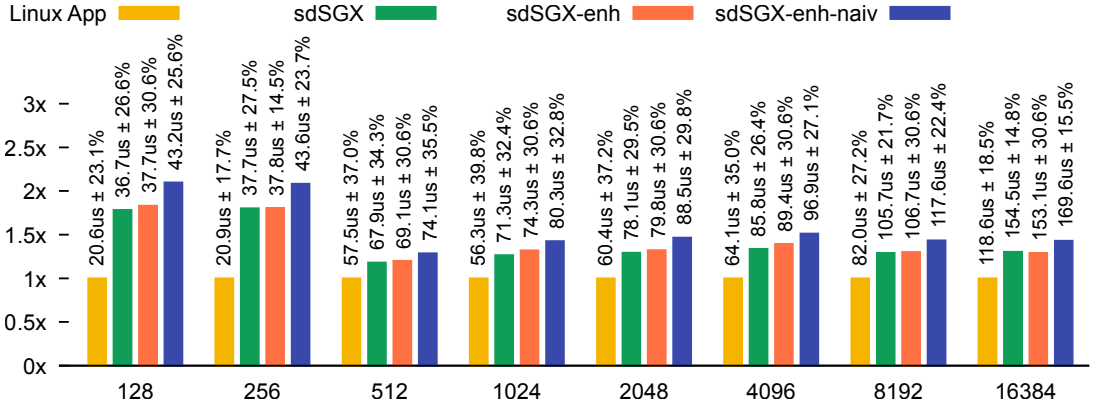


Figure 38: Execution of OpenSSL read operations for different buffer sizes. The four scenarios are: Linux application (native), sdSGX enclave (sdSGX), sdSGX enclave optimized enhancement (sdSGX-enh), sdSGX enclave with naive enhancement (sdSGX-enh-naiv).

receives 300 kiB of encrypted data in 16 kiB chunks and for every chunk an acknowledgment message is sent, and evaluate enclave and native versions.

Methodology. To measure creation, destruction and context switch operations, we use Linux time API and register elapsed time for the execution of each test (CLOCK_REALTIME parameter). For nbench we look only at the reported score values. To measure the execution of SSL operations we use the architectural timer to avoid context switches between enclave and application. For nbench and SSL, we compare execution between Linux and enclave versions of the same applications.

Microbenchmarks. Table 15 summarizes the results for the micro-operations. Enclave creation requires nested page table modifications and, thus, TLB invalidation. In enclave destruction, the entire enclave memory is zeroed to prevent leaks of security-sensitive data, and second-stage page tables are modified. Finally, the context-switch involves context switches from the application to the OS, from the OS to AnyTEE, and from AnyTEE to sdSGX. SGX-nbench results show negligible overhead, Figure 37, with the geometric mean of the relative execution time for the sgx-nbench being approximately 0.01× when compared to the nbench for Linux. These results align with recent reports on the performance of hardware virtualization [336]. Lastly, for the read operation, Figure 38, the results indicate that the context-switch operation has a non-negligible impact, 1.39×, especially for smaller buffers.

	Overhead	Avg. (ms)	Coeff. of Var.	Ctx. Switches
Linux App	-	25.80	2.30%	–
sdSGX	3.16%	26.61	2.20%	81
sdSGX-enh	3.20%	26.64	2.90%	81
sdSGX-enh-naiv	13.70%	29.34	2.40%	81

Table 16: The OpenSSL operation reads 300KB of data in 16KB chunks. The scenarios are: Linux (native), sdSGX enclave (sdSGX), sdSGX enclave with optimized enhancement (sdSGX-enh), and sdSGX enclave with naive enhancement (sdSGX-enh-naiv).

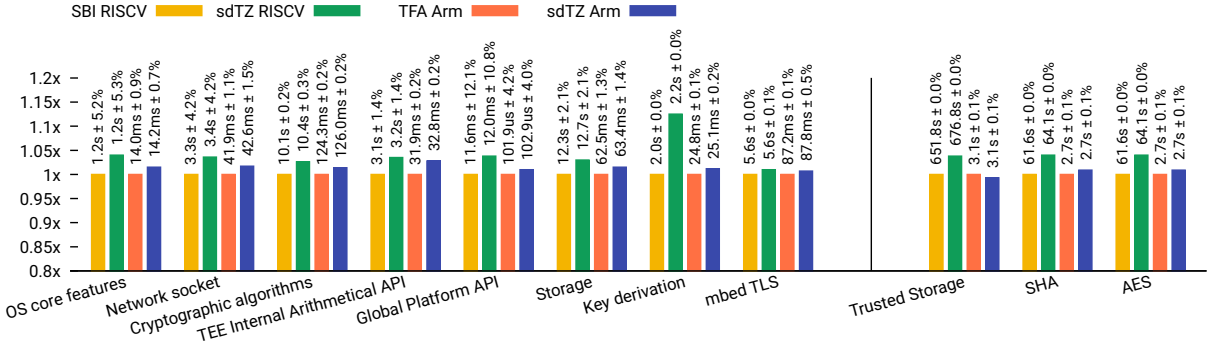


Figure 39: Geometric mean of sdTZ overhead for xtest. Each bar represents the geometric mean of overheads for the tests within a group. The left side of the figure presents the unit tests for the nine groups. The right side of the figure presents results for the benchmarks in three groups.

Application. Table 16 presents results for the SGX-OpenSSL enclave application. Overall the performance is near-native, introducing on average less than 3% overhead. This is justified by the residual impact the context-switch and hardware virtualization has on the overall execution time of the enclave.

7.5.2 sdTZ

Due to potential deviations between the performance overhead on an FPGA-based soft-core[2] and a hard-core in silicon, we have complemented the results with experiments conducted on the COTS Arm NXP platform, which is endowed with a sophisticated and performing microarchitecture. For **microbenchmarks**, we measure world switch time and the execution time of OP-TEE xtest 4.0.0, a suite of OP-TEE regression tests (grouped into nine categories) to validate the implementation of the trusted kernel and APIs. xtest also includes benchmarks to assess the performance of cryptographic and storage-related operations. At the macro level, we measure the execution time of a **real-world application**, an open-source Bitcoin wallet TA [308].

Methodology. We use the architectural timer to measure context switch time. For xtest and Bitcoin TA, we used Linux time API with CLOCK_REALTIME parameter. We compare sdTZ to our modified OpenSBI and to TF-A.

Microbenchmarks. World switch time with sdTZ (Table 17) is reduced compared to OpenSBI by 3.5%. This improvement is mostly due to the need to perform TLB invalidation on context switch using OpenSBI.

[2]Previous work has exposed that hardware virtualization support on CVA6 is not optimized for performance due to the simple microarchitecture [331].

	Average	Std. Dev.	Coeff. of Var.
sdTZ RISCV	209.03 µs	17.26 µs	8.26%
SBI RISCV	216.74 µs	20.18 µs	9.31%
Relative	0.96x	-	-

Table 17: Context switch for transition from GPOS to trusted OS.

On average, the performance overhead was 5%, with a maximum of up to 13% (Figure 39). We hypothesized this is mostly due to CVA6's simple microarchitecture, and corroborated this hypothesis by running sdTZ on Arm (with hardware virtualization support). For the AnyTEE Aarch64, we observed only a performance overhead of around 1%; this is in line with other recent studies [336], and the expectation for RISC-V silicon when hardware virtualization extensions become available in COTS platforms (e.g., SiFive P800 Series, Microchip PolarFire 2).

Application. Figure 40 presents the results for the Bitcoin wallet TA. The computed geometric mean of the performance overhead is 1.16% for RISC-V and 0.2% for Aarch64, better than the overhead assessed with xtest. As a result, our measurements suggest that AnyTEE has negligible impact in real-world applications.

7.5.3 Intra-sdTEE Isolation

We evaluate the performance of sdSGX with additional intra-sdTEE isolation (Figure 38 and Table 16). We evaluate an optimized implementation, i.e., page table switches are performed using a single instruction, denoted as sdSGX-enh. The naive implementation, involving modifications to page table entries and TLB invalidation operations, is sdSGX-enh-naiv.

Microbenchmark. Figure 38 shows the overheads for the two implemented intra-sdTEE isolation approaches. The results show that the optimized version performs 9.14% better than the naive approach, explained by the cost of performing TLB invalidations. Overall, sdSGX-enh incurs 1.87% overhead compared to the plain sdSGX enclave.

Application. In Table 16, the overheads of both implemented intra-sdTEE isolation approaches are presented for receiving a large buffer. Results report that the optimized version outperforms the naive approach by 9.2%. Again, this speedup can be justified by the reduced number of expensive microarchitectural operations, i.e., TLB invalidation and subsequent decrease in TLB misses. Overall, compared to the vanilla sdSGX enclave, sdSGX-enh adds 0.08% overhead.

7.6 Security Evaluation

We evaluate AnyTEE security through a theoretical analysis of its enforcement on security properties. We address the vectors that may be attempted to subvert security properties and highlight the protective mechanisms enforced by AnyTEE.

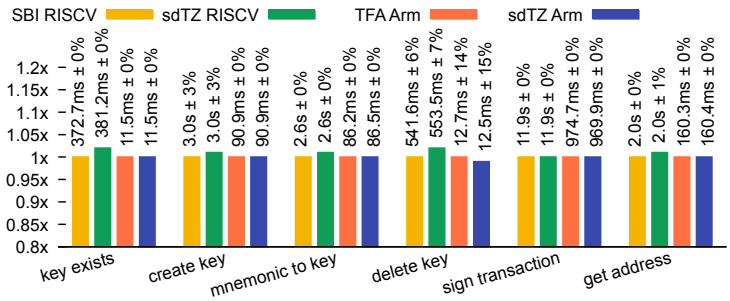


Figure 40: Relative, absolute execution time, and normalized standard deviation for Bitcoin wallet commands.

A1. Compromised GPOS. In case the GPOS is compromised, AnyTEE's nested page tables and IO protection mechanisms prevent it from making arbitrary accesses. A GPOS can abuse enclave loading. To prevent this the sdSGX handler confirms resource ownership, and attests the enclave binary. The GPOS can try to scan enclave's secrets after destruction. These attempts are twarthed by zeroing the enclave's memory and invalidating the cache upon enclave destruction.

A2. Compromised enclave. A compromised enclave may target the host application within its threat model. However, attempts to compromise the GPOS, AnyTEE's hypervisor, or other sdTEE components are prevented due to nested page tables and IO protection mechanisms.

A3. Compromised Trusted OS. A compromised trusted OS may try to gain control over the GPOS, as allowed in TrustZone's threat model. However, with proper configuration, AnyTEE allows restricting trusted OS access, and creating multiple sdTEEs for additional trusted OS environments. Recent works have addressed the importance of sandboxing trusted OSes to mitigate several critical CVEs [44, 110].

A4. TCB tampering. AnyTEE safeguards against TCB tampering by employing secure boot to abort the system boot sequence in case of firmware integrity check failure. Additionally, AnyTEE relies on one-time programmable memories, such as eFuses, to prevent attackers from modifying the platform's certificates and compromising the chain of trust.

A5. Physical access. AnyTEE does not protect against physical access attacks. However, similarly to other works, AnyTEE can use software-based solutions to encrypt memory assuming the presence of on-chip memory [107, 245], or use memory encryption accelerators if present.

A6. Side channel. While side-channel attacks are outside the scope of AnyTEE, mitigation techniques (e.g., cache coloring) can be used at the expense of some performance degradation. Controlled channel attacks can be mitigated by adopting more secure behavior for TEE emulation, for example, removing the ability of GPOS to unmap enclave memory [333] or to interrupt enclave with high-precision interrupts [334].

		TEE Model				TEE Compatibility								HW Dependency		Attack				TCB
		User	Kernel	Intra	Nested	TZ	FFA	CCA	SGX	TDX	SEV	RISC-V	CoVE	COTS	Cross.	Phys.	SC	DoS	CC	kLoC
TEE Emulation	vTZ [121]	○	●	○	○	●	○	○	○	○	○	○	○	●	○	○	○	●	○	2
	TEEVseL4 [337]	○	●	○	○	●	○	○	○	○	○	○	○	●	◐	○	○	○	○	n.p.
	MyTEE [244]	○	●	○	○	●	○	○	○	○	○	○	○	●	◐	○	○	○	○	5.5
	TwinVisor [113]	○	●	○	○	○	○	◐	○	○	○	○	○	●	○	○	○	○	○	8.7
	virtCCA [114]	○	●	○	○	○	○	●	○	○	○	○	○	●	○	○	○	●	○	11.9
	vSGX [338]	●	○	○	○	○	○	○	●	○	○	○	○	●	○	●	○	○	○	16.3
	Komodo [101]	●	○	○	○	○	○	○	◐*	○	○	○	○	●	○	○	○	○	●	3.6 f.v.
	HyperEnclave [243]	●	◐	○	○	○	○	○	●	○	○	○	○	●	◐	◐	◐	○	●	7.5
Configurable TEE	Keystone [245]	●	●	○	○	○	○	○	○	○	○	○	○	●	○	◐	◐	◐	●	10.7
	Cure [323]	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	◐	◐	●	3.1
Configurable TEE & Emulation	**AnyTEE**	●	●	●	●	●	◐	◐	●	◐	◐	◐	◐	●	●	◐	◐	●	●	9.4+

Table 18: TEE emulation and configurability state of the art comparison. Key: Phys. - Physical; SC - Side-Channel; DoS - Denial-of-Service; CC - Controlled Channel; ● - fares positively; ◐ - fares partially positively, ○ - fares negatively; f.v. - formally verified; n.p. - not provided.

7.7 Discussion

Confidential computing model support. Confidential computing solutions leverage an untrusted hypervisor that manages the platform's resources and allocates memory to create confidential VMs. Although we currently don't implement support for this, the core mechanisms provide a foundation that enables its implementation. In SEV the untrusted hypervisor allocates and manages confidential VMs. In AnyTEE, this can be implemented by extending the nested-virtualization handler with the ability to create VMs that are managed but inaccessible to the untrusted hypervisor. In TDX, CCA, and CoVE, an untrusted hypervisor interacts with a second hypervisor, referred to as a monitor. In this case, a new AnyTEE handler dedicated to emulating the interaction with the confidential VM monitor must be added.

Per-core software stacks assignment. While not the primary objective of AnyTEE, its design inherently allows for the assignment of software stacks on a per-core or set of cores basis, each equipped with its individual sdTEEs, thereby enabling distinct security properties for each stack. Compared to simultaneous execution of independent software stacks on the same cores, micro-architectural isolation is simplified by reducing shared functional units. Architectural isolation is also streamlined, eliminating the need for complex scheduling mechanisms. Minimizing the sharing of hardware devices between software stacks further simplifies implementation. Another advantageous use case is the allocation of dedicated cores for security functionality. The incorporation of heterogeneous software stacks enhances system security by facilitating fine-grained decomposition of system functionality.

7.8 Related Work

7.8.1 AnyTEE in Perspective

We compare AnyTEE with works that aim to provide TEE compatibility or emulation and configurable TEEs. This comparison is presented in Table 18. We categorize these systems by evaluating the features they introduce. If a feature falls outside the work's defined scope or is left unaddressed, we interpret the system

as performing negatively in that aspect.

TEE model and compatibility. TEE emulation has mainly focused on single TEE compatibility. For SGX, efforts like HyperEnclave [243], vSGX [338], and Komodo [101] extend enclave support beyond Intel platforms. CCA and confidential VM emulation enable confidential computing on Arm TZ platforms without realm extensions [113, 114]. vTZ [121] and TEEVsel4 [337] emulate TrustZone for per VM secure environments, while MyTEE [244] addresses TrustZone controller absence in low-cost Arm platforms. AnyTEE can simultaneously emulate multiple TEEs, hosting unmodified TEE software binaries, if running on the same architecture, provided that the handler emulates the TEE behavior and the necessary devices. Keystone uses custom runtimes for enclave applications, accommodating unmodified GPOS applications, Keystone enclave applications, and isolated OSes. Cure supports various TEE models but without support for existing TEE software stacks. AnyTEE provides similar levels of TEE models, and uses them to support multiple existing TEEs. Although support for all mentioned TEE models is not implemented, AnyTEE lays a foundation that enables future support. AnyTEE also allows nesting TEEs such that each TEE can delegate execution to its own TEEs offering new degrees of configuration and flexibility. Additionally, AnyTEE's flexibility allows modeling of potential future TEE models.

HW dependencies. Although most systems leverage COTS platforms, they often cannot be deployed across ISAs, as they leverage specific TEEs such as TrustZone [101, 113, 114, 121] and AMD SEV [338]. Hyperenclave targets x86 platforms and can benefit from AMD SME or Intel MKTME to offer memory encryption; however, support for Arm and RISC-V ISAs is not implemented. MyTEE leverages common virtualization and debug features, but requires significant engineering effort on IO access interposition for DMA-capable devices. Current configurable TEE solutions are not scalable, i.e., Keystone leverages RISC-V ISA's PMP restricting the solution to RISC-V platforms and to the limitations of the mechanism, mainly the limited number of regions. Cure on the other hand proposes specialized hardware to create TEEs. AnyTEE capitalizes on the widespread hardware support for virtualization available in all ISAs and isolating DMA IO devices.

Attacks. Regarding attacks, most works consider side-channel mitigation out of scope. Preventing denial-of-service requires that the system can take control of execution, and multiple works can implement this, but for the majority it falls outside of the scope. To prevent physical attacks, e.g., bus snooping, cold boot, systems must use memory encryption. Although Keystone offers protection for L1 cache side-channels it cannot provide L2 cache isolation without specific hardware support. Cure provides mechanisms for L1 and L2 cache isolation. AnyTEE does not implement L1 or L2 cache isolation, but it can be adapted to support isolation schemes. One example is using cache coloring and reserving one cache color for TEEs, with all other colors for untrusted sdTEEs, and forcing only one TEE to execute at a time.

TCB. Number of lines of code (LoC) ranges from 2 kLoC to 16.3 kLoC, including various layers from firmware to hypervisor. For TEEVseL4, seL4 is formally verified but the VMM, for which LoC is not provided, is not. Keystone TCB encompasses the secure monitor software and in Cure only a subset of the secure

monitor code. The core of AnyTEE is 9.4 kLoC, with additional TEE emulation handler requiring additional lines of code. For sdTZ, we add 131 LoC and for sdSGX we add 413 LoC. AnyTEE also trusts the host's TEE.

7.8.2 Other Works

Hardware-oriented systems offer specific hardware and firmware components for constructing TEE systems [54]. These include CPU extensions such as TrustZone, SGX, SEV, and others [19, 23, 87, 299, 311, 322, 323], dedicated co-processors [25, 26, 323], tiles of security processors [46, 339, 340], and security chips such as Titan-M and TPM [29, 341]. Academia has proposed to leverage the available TEE primitives to achieve improved security goals, creating isolated environments with improved security guarantees, e.g., Sanctuary, ReZone, and others [44, 101, 103, 115, 116]. Other works leverage virtualization in addition to the platform's security mechanisms [103, 118, 120, 342], with some, like TEEv [110] and PrOS [111] leveraging software-based virtualization techniques. Overshadow [343] and Inktag [344], among others [45, 345, 346] have proposed the use of virtualization as the single isolation mechanism. To improve on interoperability, initiatives such as Google's Asylo [347], Enarx [348] and others [226–228, 349–352], provide specific SDKs. However, having to re-write code for an SDK, and relying on the SDK to support the selected platform precludes using legacy TEE applications with minimal effort.

7.9 Conclusion

In this work, we propose AnyTEE, a framework for developing software-defined TEEs that builds upon a set of virtualization-based security mechanisms. We achieve cross-platform, simultaneous, and refined software-defined TEEs, compatible with legacy trusted applications. Our evaluation demonstrates overheads of less than 3% when compared to native TEE solutions while achieving cross-platform compatibility with two of the widely used TEEs.

Bibliography

[1] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2020 (cit. on p. 2).

[2] C. Details. *Linux Kernel Security Vulnerabilities*. www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/Linux-Linux-Kernel.html. 2020 (cit. on pp. 2, 3).

[3] Synopsys. *Chromium (Google Chrome)*. https://openhub.net/p/chrome/analyses/latest/languages_summary. 2024 (cit. on p. 2).

[4] C. Details. *Security Vulnerabilities, CVEs, by Chrome*. https://www.cvedetails.com/vulnerability-list/assigner-70/Chrome.html. 2020 (cit. on p. 2).

[5] T. C. Projects. *Chromium Security*. https://www.chromium.org/Home/chromium-security/. 2020 (cit. on p. 2).

[6] F. Zhang and H. Zhang. "SoK: A Study of Using Hardware-Assisted Isolated Execution Environments for Security". In: *Proc. of Hardware and Architectural Support for Security and Privacy*. 2016 (cit. on p. 3).

[7] S. Pinto and N. Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". In: *ACM Comput. Surv.* (2019) (cit. on pp. 3, 10, 35, 45, 66).

[8] N. Santos et al. "Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications". In: *SIGARCH Computer Architecture News* (2014). issn: 0163-5964 (cit. on pp. 3, 27, 63–65, 100).

[9] PallyCon. *About Google Widevine DRM*. www.pallycon.com/google-widevine-drm/. 2019 (cit. on pp. 3, 9, 88).

[10] HC Ma. *How Samsung Secures Your Wallet and How To Break It*. www.blackhat.com/docs/eu-17/materials/eu-17-Ma-How-Samsung-Secures-Your-Wallet-And-How-To-Break-It.pdf. 2017 (cit. on pp. 3, 47).

[11] C. C. Consortium. *Confidential Computing Consortium*. https://confidentialcomputing.io/. 2020 (cit. on p. 3).

[12] A. W. Services. *AWS Nitro Enclaves*. https://aws.amazon.com/ec2/nitro/nitro-enclaves/. 2024 (cit. on p. 3).

[13] G. Cloud. *Confidential VM Overview*. https://cloud.google.com/confidential-computing/confidential-vm/docs/confidential-vm-overview. 2024 (cit. on p. 3).

[14] M. Azure. *Azure Confidential Computing*. https://azure.microsoft.com/en-us/solutions/confidential-compute/. 2024 (cit. on p. 3).

[15] S. Volos, K. Vaswani, and R. Bruno. "Graviton: Trusted Execution Environments on {GPUs}". In: *Proc. of OSDI*. 2018 (cit. on p. 3).

[16] F. Tramer and D. Boneh. "Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware". In: *arXiv preprint arXiv:1806.03287* (2018) (cit. on p. 3).

[17] I. Stoica et al. "A Berkeley View of Systems Challenges for AI". In: *arXiv preprint arXiv:1712.05855* (2017) (cit. on p. 3).

[18] T. Alves and D. Felton. "TrustZone: Integrated Hardware and Software Security". In: *Tech. In-Depth* (2004) (cit. on pp. 3, 45).

[19] S. Pinto et al. "Virtualization on TrustZone-enabled Microcontrollers? Voilà!" In: *Proc. of RTAS*. 2019 (cit. on pp. 3, 45, 70, 72, 99, 100, 102, 105, 123).

[20] Intel. *Intel Software Guard Extensions*. www.software.intel.com/en-us/sgx/. 2019 (cit. on pp. 3, 66, 67).

[21] AMD. *Secure Technology*. www.amd.com/en/technologies/security. 2019 (cit. on p. 3).

[22] X. Li et al. "Design and Verification of the Arm Confidential Compute Architecture". In: *Proc. of USENIX OSDI*. 2022 (cit. on pp. 3, 18, 20, 102, 105).

[23] Intel. *Intel Trust Domain Extensions (Intel TDX)*. www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html. 2021 (cit. on pp. 3, 102, 105, 123).

[24] R. Sahita et al. "CoVE: Towards Confidential Computing on RISC-V Platforms". In: *Proc. of ACM International Conference on Computing Frontiers*. 2023 (cit. on pp. 3, 24, 102).

[25] T. Mandt, M. Solnik, and D. Wang. "Demystifying the Secure Enclave Processor". In: *Black Hat Las Vegas* (2016) (cit. on pp. 3, 66, 67, 100, 123).

[26] Qualcomm. *Qualcomm Secure Processing Unit SPU230 Core Security Target Lite*. www.commoncriteriaportal.org/files/epfiles/1045b_pdf.pdf. 2019 (cit. on pp. 3, 66, 67, 77, 100, 123).

[27] J. O. *Getting Started with Intel Active Management Technology (Intel AMT)*. www.software.intel.com/en-us/articles/getting-started-with-intel-active-management-technology-amt. 2019 (cit. on pp. 3, 66, 67).

[28] X. Xin. *Titan M makes Pixel 3 our most secure phone yet*. www.blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/. 2018 (cit. on pp. 3, 66, 67).

[29] T. C. Group. *Trusted Platform Module (TPM)*. www.trustedcomputinggroup.org/work-groups/trusted-platform-module/. 2018 (cit. on pp. 3, 66, 67, 100, 123).

[30] Arm. *Arm Holdings plc FYE24-Q2 Results Presentation*. Slide 18, https://investors.arm.com/static-files/b43123f6-9236-4e3c-bd7f-05c3aa2b26c1 (cit. on p. 3).

[31] Arm. "ARM Security Technology. Building a Secure System using TrustZone Technology ARM". In: *Arm white paper* (2009) (cit. on pp. 3, 18, 43, 66).

[32] Symantec. *iOS Malware, XcodeGhost, Infects Millions of Apple Store Customers*. www.us.norton.com/internetsecurity-emerging-threats-ios-malware-xcodeghost-infects-millions-of-apple-store-customers.html. 2018 (cit. on pp. 3, 43).

[33] Dan Goodin. *Malware Found Preinstalled on 38 Android Phones Used by 2 Companies*. www.arstechnica.com/information-technology/2017/03/preinstalled-malware-targets-android-users-of-two-companies/. 2017 (cit. on pp. 3, 43).

[34] Swati Khandelwal. *New Android Malware Framework Turns Apps Into Powerful Spyware*. www.thehackernews.com/2018/08/android-malware-spyware.html. 2018 (cit. on pp. 3, 43).

[35] X. Jiang and Y. Zhou. "Dissecting Android Malware: Characterization and Evolution". In: *2012 IEEE Symposium on Security and Privacy*. 2012 (cit. on pp. 3, 43).

[36] Trustonic. *Trustonic Application Protection Delivers Comprehensive Security for Mobile Financial Services*. www.trustonic.com/markets/financial-services/. 2018 (cit. on pp. 4, 9, 44, 70).

[37] Michael Lu. *TrustZone, TEE and Trusted Video Path Implementation Considerations*. www.arm.com/files/event/Developer_Track_6_TrustZone_TEEs_and_Trusted_Video_Path_implementation_considerations.pdf. 2018 (cit. on pp. 4, 44, 70).

[38] Google. *Android Compatibility Definition Document*. www.source.android.com/compatibility/10/android-10-cdd.pdf. 2020 (cit. on p. 4).

[39] Android. *Fingerprint HAL*. www.source.android.com/security/authentication/fingerprint-hal.html. 2018 (cit. on pp. 4, 31, 44).

[40] Gal Beniamini. *Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption (Jun 2016)*. www.bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html. 2016 (cit. on pp. 4, 37, 47, 71).

[41] Gal Beniamini. *QSEE Privilege Escalation Exploit using PRDiag* commands (CVE-2015-6639)*. http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html. 2016 (cit. on p. 4).

[42] Gal Beniamini. *TrustZone Kernel Privilege Escalation (CVE-2016-2431)*. http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html. 2016 (cit. on pp. 4, 37, 44, 46, 57, 71).

[43] D. Cerdeira et al. "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems". In: *Proc. of S&P*. 2020 (cit. on pp. 5, 8, 35, 41, 72, 94, 100, 104, 114, 115).

[44] D. Cerdeira et al. "ReZone: Disarming TrustZone with TEE Privilege Reduction". In: *Proc. of USENIX Security*. 2022 (cit. on pp. 6, 8, 26, 35, 41, 104, 106, 115, 120, 123).

[45] S. Pereira et al. "Bao-Enclave: Virtualization-based enclaves for arm". In: *WF-IoT*. 2022 (cit. on pp. 7, 123).

[46] S. Pereira et al. "Towards a Trusted Execution Environment Via Reconfigurable FPGA". In: *Under Review in Computer & Security* (2024) (cit. on pp. 7, 41, 123).

[47] S. Pinto et al. "LTZvisor: Trustzone Is The Key". In: *Proc. of ECRTS*. 2017 (cit. on pp. 7, 29, 100).

[48] S. Pinto et al. "Self-Secured Devices: High Performance and Secure I/O Access in TrustZone-based systems". In: *Journal of Systems Architecture* (2021) (cit. on pp. 7, 41).

[49] G. D. P. Regulation. "General Data Protection Regulation (GDPR)". In: *Intersoft Consulting* (2018) (cit. on p. 8).

[50] H. Unnibhavi et al. "Secure and Policy-Compliant Query Processing on Heterogeneous Computational Storage Architectures". In: *Proc. of SIGMOD*. 2022 (cit. on pp. 8, 31, 41).

[51] J. Martins et al. "Genesys: In the Beginning There Were Static Partitioning Hypervisors... Let There Be Light!" In: (2024) (cit. on pp. 8, 41).

[52] android. *android keystore system*. www.developer.android.com/training/articles/keystore. 2020 (cit. on pp. 9, 30).

[53] Global Platform. *GlobalPlatform Technology TEE System Architecture Version 1.2*. https://globalplatform.org/wp-content/uploads/2017/01/GPD_TEE_SystemArch_v1.2_PublicRelease.pdf. 2018 (cit. on pp. 9, 10).

[54] M. Schneider et al. "SoK: Hardware-Supported Trusted Execution Environments". In: *arXiv preprint arXiv:2205.12742* (2022) (cit. on pp. 9, 102, 106, 123).

[55] Arm. *TrustZone for Armv8-A*. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/TrustZone%20for%20Armv8-A.pdf?revision=c3134c8e-f1d0-42ff-869e-0e6a6bab824f. 2019 (cit. on p. 10).

[56] M. J. Flynn and W. Luk. *Computer System Design: System-on-Chip*. 2011 (cit. on pp. 11–13).

[57] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, 2018 (cit. on p. 11).

[58] Keegan Ryan. *Hardware-BackedHeist: Extracting ECDSA Keys from Qualcomm's TrustZone*. www.nccgroup.trust/globalassets/our-research/us/whitepapers/2019/hardwarebackedhesit.pdf. 2019 (cit. on pp. 11, 60, 62, 63, 66).

[59] M. Lipp et al. "Armageddon: Cache Attacks on Mobile Devices". In: *Proc. of USENIX Security*. 2016 (cit. on p. 11).

[60] J. Handy. *The Cache Memory Book*. 1998 (cit. on p. 12).

[61] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 2011 (cit. on pp. 12, 14).

[62] M. Lipp et al. "ARMageddon: Cache Attacks on Mobile Devices". In: *Proc. of USENIX Security*. USENIX Association, 2016 (cit. on pp. 12, 44, 60, 61, 63, 66, 67).

[63] Arm. *ARM Generic Interrupt Controller Architecture version 2.0 - Architecture Specification*. https://developer.arm.com/documentation/ihi0048/bb/?lang=en. 2013 (cit. on p. 13).

[64] Arm. *Arm Generic Interrupt Controller (GIC) Architecture Specification GIC architecture version 3 and version 4*. https://developer.arm.com/documentation/ihi0069/hb/?lang=en. 2024 (cit. on p. 13).

[65] Arm. *TrustZone for Armv8-A*. www.documentation-service.arm.com/static/602167b6873dd96c4deaf49b. 2019 (cit. on pp. 13, 14, 70).

[66] Arm. *Learn the Architecture - An Introduction to AMBA AXI*. https://developer.arm.com/documentation/102202/latest/. 2022 (cit. on pp. 13, 14).

[67] NXP. *Secure Data Path on Linux and NXP i.MX 8M*. https://optee.readthedocs.io/en/latest/general/presentations.html. 2018 (cit. on p. 14).

[68] R. Stajnrod, R. Ben Yehuda, and N. J. Zaidenberg. "Attacking TrustZone on devices lacking memory protection". In: *Journal of Computer Virology and Hacking Techniques* (2021) (cit. on p. 14).

[69] Arm. *CoreLink TrustZone Address Space Controller TZC-380*. https://developer.arm.com/documentation/ddi0431/. 2010 (cit. on pp. 14, 15).

[70] Arm. *CoreLink TrustZone Address Space Controller TZC-400*. https://developer.arm.com/documentation/ddi0504/. 2014 (cit. on pp. 14, 15).

[71] Arm. *PrimeCell Infrastructure AMBA 3 TrustZone Protection Controller (BP147)*. www.developer.arm.com/documentation/dto0015/a/. 2004 (cit. on pp. 14, 77).

[72] Arm. *PrimeCell Infrastructure AMBA 3 AXI TrustZone Memory Adapter*. https://developer.arm.com/documentation/dto0017. 2004 (cit. on p. 14).

[73] NXP. *MCUXpresso SDK API Reference Manual*. https://mcuxpresso.nxp.com/api_doc/dev/1411/a00057.html. 2016 (cit. on p. 15).

[74] Qualcomm. *An Introduction to Access Control on Qualcomm Snapdragon Platforms*. www.qualcomm.com/media/documents/files/an-introduction-to-access-control-on-qualcomm-snapdragon-platforms.pdf. 2020 (cit. on pp. 15, 75, 77).

[75] A. M. Azab et al. "HIMA: A Hypervisor-Based Integrity Measurement Agent". In: *2009 Annual Computer Security Applications Conference*. 2009 (cit. on p. 16).

[76] Kyle Orland. *The "Unpatchable" Exploit That Makes Every Current Nintendo Switch Hackable [Updated]*. https://arstechnica.com/gaming/2018/04/the-unpatchable-exploit-that-makes-every-current-nintendo-switch-hackable/. 2004 (cit. on p. 17).

[77] J. Wiklander. *Open Portable Trusted Execution Environment*. https://blog.linuxplumbersconf.org/2016/ocw/system/presentations/3675/original/LPC%202016%20-%20OP-TEE.pdf. 2016 (cit. on p. 18).

[78] Linaro. *OP-TEE*. www.op-tee.org/. 2016 (cit. on pp. 18, 27, 34).

[79] Google. *Trusty TEE*. https://source.android.com/docs/security/features/trusty. 2024 (cit. on pp. 18, 27).

[80] Arm. *Isolation using Virtualization in the Secure world: Secure world software architecture on Armv8.4*. 2018 (cit. on p. 18).

[81] Arm. *TrustZone technology for Armv8-M Architecture*. https://developer.arm.com/documentation/100690/0201/. 2016 (cit. on pp. 18, 20).

[82] TrustedFirmware. *Hafnium*. www.trustedfirmware.org/projects/hafnium/. 2021 (cit. on pp. 19, 35, 72, 98).

[83] Intel. *Intel*. https://www.intel.com/ (cit. on p. 21).

[84] V. Costan and S. Devadas. "Intel SGX Explained". In: *IACR Cryptology ePrint Archive* (2016) (cit. on pp. 21, 100, 102, 105).

[85] Intel. *Intel Trust Domain Extensions (Intel TDX)*. www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html. 2021 (cit. on p. 22).

[86] AMD. *AMD*. https://www.amd.com/ (cit. on p. 22).

[87] D. Kaplan, J. Powell, and T. Woller. *AMD Memory Encryption*. www.developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf. White Paper. 2021 (cit. on pp. 23, 66, 67, 100, 102, 105, 123).

[88] R.-V. Foundation. *RISC-V*. https://www.riscv.org/ (cit. on p. 23).

[89] R.-V. Foundation. *Privileged Architecture v1.12, Ratified*. www.github.com/riscv/riscv-isa-manual/releases/tag/Priv-v1.12. 2021 (cit. on pp. 24, 105).

[90] D. Cerdeira et al. "AnyTEE: An Open and Interoperable Software Defined TEE Framework". In: () (cit. on pp. 26, 41).

[91] Qualcomm. *Qualcomm Mobile Security*. www.qualcomm.com/solutions/mobile-computing/features/security. 2018 (cit. on p. 26).

[92] Huawei. *EMUI 10.1 Security Technical White Paper*. https://consumer-img.huawei.com/content/dam/huawei-cbg-site/common/campaign/privacy/whitepaper/emui-10-security-technical-white-paper-v1.pdf. 2020 (cit. on p. 26).

[93] Samsung. *Samsung TEEGRIS*. https://developer.samsung.com/teegris/overview.html. 2024 (cit. on p. 26).

[94] *SierraTEE*. www.sierraware.com/open-source-ARM-TrustZone.html (cit. on pp. 26, 34, 48).

[95] Trustonic. *Trustonic*. www.trustonic.com/solutions/trustonic-secured-platforms-tsp/. 2018 (cit. on p. 26).

[96] Arm. *Trusted Firmware-A*. www.git.trustedfirmware.org/TF-A/trusted-firmware-a.git (cit. on pp. 27, 77).

[97] X. Yang et al. "Trust-E: A Trusted Embedded Operating System Based on the ARM TrustZone". In: *Proc. of Ubiquitous Intelligence and Computing and Autonomic and Trusted Computing and Scalable Computing and Communications and Associated Workshops*. 2014 (cit. on p. 27).

[98] M. Paolino et al. "T-KVM: A Trusted Architecture for KVM ARM v7 and v8 Virtual Machines Securing Virtual Machines by Means of KVM, TrustZone, TEE and SELinux". In: *Proc. of Cloud Computing, GRIDs, and Virtualization*. 2015 (cit. on p. 27).

[99] B. McGillion et al. "Open-TEE-an Open Virtual Trusted Execution Environment". In: *arXiv preprint arXiv:1506.07367* (2015) (cit. on p. 27).

[100] N. Feske. *Genode Operating System Framework*. 2015 (cit. on p. 27).

[101] A. Ferraiuolo et al. "Komodo: Using verification to disentangle secure-enclave hardware from software". In: *Proc. of Operating Systems Principles*. 2017 (cit. on pp. 27, 29, 35, 63–65, 100, 103, 121–123).

[102] K. Rubinov et al. "Automated Partitioning of Android Applications for Trusted Execution Environments". In: *Proc. of Software Engineering*. 2016 (cit. on pp. 27, 28).

[103] L. Guan et al. "TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone". In: *Proc. of MobiSys* (2017) (cit. on pp. 27, 100, 123).

[104] M. H. Yun and L. Zhong. "Ginseng: Keeping Secrets in Registers When You Distrust the Operating System". In: *Network and Distributed Systems Security (NDSS) Symposium*. 2019 (cit. on pp. 27, 63–65).

[105] N. Zhang et al. "CaSE: Cache-Assisted Secure Execution on ARM Processors". In: *Proc. of S&P*. 2016 (cit. on pp. 27, 63, 64, 100).

[106] S. Zhao et al. “SecTEE: A Software-Based Approach to Secure Enclave Architecture Using TEE”. In: *Proc. of CSS*. 2019 (cit. on pp. 27, 100, 114).

[107] M. Zhang et al. “SoftME: A Software-Based Memory Protection Approach for TEE System to Resist Physical Attacks”. In: *Security and Communication Networks* (2019) (cit. on pp. 27, 114, 120).

[108] S. Wan et al. “RusTEE: Developing Memory-Safe ARM TrustZone Applications”. In: *Annual Computer Security Applications Conference*. 2020 (cit. on p. 28).

[109] J. Ménétrey et al. “Watz: a Trusted WebAssembly Runtime Environment with Remote Attestation for TrustZone”. In: *Proc. of ICDCS*. 2022 (cit. on p. 28).

[110] W. Li et al. “TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms”. In: *Proc. of VEE*. 2019 (cit. on pp. 28, 35, 63, 64, 72, 95–97, 100, 115, 120, 123).

[111] D. Kwon et al. “PrOS: Light-weight Privatized Secure OSes in ARM TrustZone”. In: *TMC* (2019) (cit. on pp. 28, 35, 63, 72, 95–97, 100, 115, 123).

[112] G. Cicero et al. “Reconciling Security with Virtualization: A Dual-Hypervisor Design for ARM TrustZone”. In: *Proc. of ICIT*. 2018 (cit. on p. 28).

[113] D. Li et al. “Twinvisor: Hardware-Isolated Confidential Virtual Machines for Arm”. In: *Proc. of the ACM SIGOPS*. 2021 (cit. on pp. 28, 121, 122).

[114] X. Xu et al. “virtCCA: Virtualized Arm Confidential Compute Architecture with TrustZone”. In: *arXiv preprint* (2023) (cit. on pp. 28, 35, 103, 121, 122).

[115] H. Sun et al. “Trustice: Hardware-assisted isolated computing environments on mobile devices”. In: *Proc. of DSN*. 2015 (cit. on pp. 28, 63, 95–97, 100, 123).

[116] F. Brasser et al. “SANCTUARY: ARMing TrustZone with User-space Enclaves”. In: *Proc. of NDSS*. 2019 (cit. on pp. 28, 35, 63, 64, 66, 72, 95–97, 100, 123).

[117] Y. Cho et al. “Hardware-Assisted on-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices”. In: *Proc. of USENIX ATC*. 2016 (cit. on pp. 28, 63, 95–97, 100).

[118] J. Jang et al. “PrivateZone: Providing a Private Execution Environment Using ARM TrustZone”. In: *IEEE TDSC* (2018) (cit. on pp. 28, 63, 64, 123).

[119] Z. Hua et al. “TZ-container: Protecting Container from Untrusted OS with ARM TrustZone”. In: *Science China Information Sciences* (2021) (cit. on p. 28).

[120] M. Zhu et al. “HA-VMSI: A Lightweight Virtual Machine Isolation Approach With Commodity Hardware for ARM”. In: *ACM SIGPLAN Notices* (2017) (cit. on pp. 28, 123).

[121] Z. Hua et al. “vTZ: Virtualizing ARM TrustZone”. In: *Proc. of USENIX Security*. 2017 (cit. on pp. 28, 43, 63, 95–97, 100, 115, 121, 122).

[122] S. Pinto et al. “IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices”. In: *IEEE Internet Computing* (2017). issn: 1089-7801 (cit. on p. 28).

[123] M. Cereia and I. C. Bertolotti. “Asymmetric Virtualisation for Real-Time Systems”. In: *Proc. of Industrial Electronics*. 2008 (cit. on p. 28).

[124] D. Sangorrin, S. Honda, and H. Takada. “Dual Operating System Architecture for Real-Time Embedded Systems”. In: (2010) (cit. on p. 28).

[125] J. Martins et al. “μ RTZVisor: a Secure and Safe Real-Time Hypervisor”. In: *Electronics* (2017) (cit. on p. 29).

[126] P. Lucas et al. “Vosysmonitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on Armv8-A”. In: *Proc. of ECRTS 2017*. 2017 (cit. on p. 29).

[127] P. Dong et al. “Tzdks: A New TrustZone-Based Dual-Criticality System with Balanced Performance”. In: *Proc. of RTCSA*. 2018 (cit. on p. 29).

[128] S. Pinto et al. “Lightweight Multicore Virtualization Architecture Exploiting ARM TrustZone”. In: *Proc. of IECON*. 2017 (cit. on p. 29).

[129] J. Wang et al. “RT-TEE: Real-Time System Availability for Cyber-Physical Systems Using Arm TrustZone”. In: *Proc. of S&P*. 2022 (cit. on p. 29).

[130] S. Pinto et al. “FreeTEE: When Real-Time and Security Meet”. In: *Proc. of ETFA*. 2015 (cit. on p. 29).

[131] Y. Ma et al. “Formal Verification of Memory Isolation for the TrustZone-Based TEE”. In: *Proc. of Software Engineering Conference APSEC*. 2020 (cit. on p. 29).

[132] R. Chang et al. “MIPE: a Practical Memory Integrity Protection Method in a Trusted Execution Environment”. In: *Cluster Computing* (2017). issn: 1573-7543 (cit. on pp. 29, 33, 63, 65).

[133] F. Brasser et al. “Regulating Arm TrustZone Devices in Restricted Spaces”. In: *Proc. of Conference on Mobile Systems, Applications, and Services*. 2016 (cit. on p. 29).

[134] S. W. Kim et al. “Secure Device Access for Automotive Software”. In: *Proc. of ICCVE*. 2013 (cit. on p. 29).

[135] R. Liu and M. Srivastava. “Protc: Protecting drone’s peripherals through arm trustzone”. In: *Proc. of Micro Aerial Vehicle Networks, Systems, and Applications*. 2017 (cit. on p. 29).

[136] H. Liu et al. “Software Abstractions for Trusted Sensors”. In: *Proc. of Mobile Systems, Applications, and Services*. 2012 (cit. on p. 29).

[137] M. Lentz et al. “SeCloak: ARM Trustzone-based Mobile Peripheral Control”. In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys ’18. Munich, Germany: ACM, 2018. isbn: 978-1-4503-5720-3 (cit. on pp. 29, 64).

[138] D. Shim and D. H. Lee. “SOTPM: Software One-Time Programmable Memory to Protect Shared Memory on ARM TrustZone”. In: *IEEE Access* (2020) (cit. on p. 29).

[139] D. Zhang and S. You. “iFlask: Isolate Flask Security System From Dangerous Execution Environment by using ARM TrustZone”. In: *Future Generation Computer Systems* (2020) (cit. on p. 29).

[140] C. Segarra, R. Delgado-Gonzalo, and V. Schiavoni. “MQT-TZ: Hardening IoT Brokers Using ARM TrustZone:(Practical Experience Report)”. In: *Proc. of SRDS*. 2020 (cit. on p. 29).

[141] A. Ahlawat and W. Du. “TruzCall: Secure VoIP Calling on Android Using ARM TrustZone”. In: *Proc. of MobiSecServ*. 2020 (cit. on p. 29).

[142] S. Gupta. “An Edge-Computing Based Industrial Gateway for Industry 4.0 Using ARM TrustZone technology”. In: *Journal of Industrial Information Integration* (2023) (cit. on p. 30).

[143] F. Schwarz. “TrustedGateway: TEE-Assisted Routing and Firewall Enforcement Using ARM TrustZone”. In: *Proc. of Research in Attacks, Intrusions and Defenses*. 2022 (cit. on p. 30).

[144] 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. IEEE. 2015, pp. 57–64 (cit. on p. 30).

[145] J. Shin et al. “DFCloud: A TPM-based secure data access control method of cloud storage in mobile devices”. In: *Proc. of Cloud Computing Technology and Science Proceedings*. 2012 (cit. on p. 30).

[146] S. Zhao et al. “Providing Root of Trust for ARM TrustZone Using On-Chip SRAM”. In: *Proc. of Workshop on Trustworthy Embedded Devices*. 2014 (cit. on p. 30).

[147] H. Raj et al. “{fTPM}: A {Software-Only} Implementation of a {TPM} Chip”. In: *Proc. of USENIX Security*. 2016 (cit. on p. 30).

[148] M. Gross et al. “Enhancing the Security of FPGA-SoCs via the Usage of ARM TrustZone and a Hybrid-TPM”. In: *TRETS* (2021) (cit. on p. 30).

[149] H. Xia and W. Yang. “Security Access Solution of Cloud Services for Trusted Mobile Terminals Based on TrustZone.” In: *Int. J. Netw. Secur.* (2020) (cit. on p. 30).

[150] V. Jyothi et al. “FPGA Trust Zone: Incorporating Trust and Reliability into FPGA Designs”. In: *Proc. of ICCD*. 2016 (cit. on p. 30).

[151] Linaro. *Keymaster and Gatekeeper*. https://github.com/linaro-swg/kmgk. 2023 (cit. on p. 30).

[152] S. Luo, Z. Hua, and Y. Xia. “TZ-KMS: A Secure Key Management Service for Joint Cloud Computing with ARM TrustZone”. In: *Proc. of SOSE*. 2018 (cit. on p. 30).

[153] X. Li et al. “DroidVault: A Trusted Data Vault for Android devices”. In: *2014 19th International Conference on Engineering of Complex Computer Systems*. 2014 (cit. on p. 30).

[154] H. Daniel, J. Winter, and A. Fitzek. “Secure Block Device-Secure Flexible and Efficient Data Storage for ARM TrustZone Systems”. In: *Trustcom/BigDataSE/ISPA* (2015) (cit. on p. 30).

[155] J. Liao, B. Chen, and W. Shi. “TrustZone Enhanced Plausibly Deniable Encryption System for Mobile Devices”. In: *Proc. of SEC*. 2021 (cit. on p. 30).

[156] S. Brenner, C. Wulf, and R. Kapitza. “Running {ZooKeeper} Coordination Services in Untrusted Clouds”. In: *Proc. of HotDep*. 2014 (cit. on p. 31).

[157] P. Hunt et al. “{ZooKeeper}: Wait-Free Coordination for Internet-Scale Systems”. In: *Proc. of USENIX ATC*. 2010 (cit. on p. 31).

[158] Android. *Gatekeeper*. https://source.android.com/docs/security/features/authentication/gatekeeper. 2020 (cit. on p. 31).

[159] H. Sun et al. “TrustOTP: Transforming smartphones into secure one-time password tokens”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015 (cit. on pp. 31, 33).

[160] K. Kostiainen et al. “On-board Credentials: an Open Credential Platform for Mobile Devices”. In: *SPACE* (2012) (cit. on p. 31).

[161] D. Liu and L. P. Cox. “Veriui: Attested Login for Mobile Devices”. In: *Proc. of workshop on mobile computing systems and applications*. 2014 (cit. on pp. 31, 33).

[162] R. A. Balisane and A. Martin. “Trusted Execution Environment-Based Authentication Gauge (TEE-BAG)”. In: *Proceedings of the 2016 New Security Paradigms Workshop*. 2016 (cit. on pp. 31, 33).

[163] R. v. Rijswijk-Deij and E. Poll. “Using Trusted Execution Environments in Two-Factor Authentication: Comparing Approaches”. In: *Open Identity Summit 2013* (2013) (cit. on pp. 31, 33).

[164] Y. Zhang et al. “Trusttokenf: A Generic Security Framework for Mobile Two-Factor Authentication Using TrustZone”. In: *IEEE Trustcom/BigDataSE/ISPA*. 2015 (cit. on p. 31).

[165] B. Zhao et al. “A Private User Data Protection Mechanism in TrustZone Architecture Based on Identity Authentication”. In: *Tsinghua Science and Technology* (2017) (cit. on p. 31).

[166] B. Yang et al. “DAA-TZ: An Efficient DAA Scheme for Mobile Devices using ARM TrustZone”. In: *Proc. of TRUST*. 2015 (cit. on p. 31).

[167] T. Feng et al. “Secure Session on Mobile: An Exploration on Combining Biometric, TrustZone, and User Behavior”. In: *Proc. of Mobile Computing, Applications and Services*. 2014 (cit. on p. 31).

[168] D. Zhang. “Trustfa: Trustzone-Assisted Facial Authentication on Smartphone”. In: *Tech. Rep* (2014) (cit. on p. 31).

[169] S. D. Yalew et al. “TruApp: A TrustZone-Based Authenticity Detection Service for Mobile Apps”. In: *Proc. of WiMob*. 2017 (cit. on p. 31).

[170] H. Jiang et al. “An Effective Authentication for Client Application using ARM TrustZone”. In: *Proc. of ISPEC*. 2017 (cit. on p. 31).

[171] Z. Wang, Y. Zhuang, and Z. Yan. “TZ-MRAS: a Remote Attestation Scheme for the Mobile Terminal Based on ARM TrustZone”. In: *Security and Communication Networks* (2020) (cit. on pp. 31, 33).

[172] W. H. W. Hussin, P. Coulton, and R. Edwards. “Mobile Ticketing System Employing TrustZone Technology”. In: *Proc. of Mobile Business ICMB*. 2005 (cit. on p. 31).

[173] M. Pirker and D. Slamanig. “A Framework for Privacy-Preserving Mobile Payment on Security Enhanced Arm TrustZone Platforms”. In: *Proc. of Trust, Security and Privacy in Computing and Communications*. 2012 (cit. on p. 31).

[174] X. Zheng et al. “TrustPAY: Trusted mobile payment on security enhanced ARM TrustZone platforms”. In: *Proc. of ISCC*. 2016 (cit. on pp. 31, 33).

[175] F. Akowuah and A. Ahlawat. “Protecting Sensitive Data in Android sqlite Databases using Trust-Zone”. In: *Proc. of Security & Management*. 2018 (cit. on p. 31).

[176] P. S. Ribeiro, N. Santos, and N. O. Duarte. “DBStore: A TrustZone-backed Database Management System for Mobile Applications.” In: *ICETE*. 2018 (cit. on p. 31).

[177] O. Benedito, R. Delgado-Gonzalo, and V. Schiavoni. “KeVlar-Tz: A Secure Cache for Arm TrustZone: (Practical Experience Report)”. In: *IFIP International Conference on Distributed Applications and Interoperable Systems*. 2021 (cit. on p. 31).

[178] S. Lee et al. “How to Securely Record Logs Based on ARM TrustZone”. In: *Proc. of Asia CCS*. 2019 (cit. on p. 31).

[179] S. Mirzamohammadi et al. “Ditio: Trustworthy Auditing of Sensor Activities in Mobile & IoT devices”. In: *Proc. of Conference On Embedded Network Sensor Systems*. 2017 (cit. on p. 31).

[180] Q. Zhang et al. “Design and Implementation of Trustzone-Based Blockchain Chip Wallet”. In: *Proc. of ICSIP*. 2021 (cit. on p. 32).

[181] M. Gentilal, P. Martins, and L. Sousa. “TrustZone-backed bitcoin wallet”. In: *Proc. of Cryptography and Security in Computing Systems*. 2017 (cit. on p. 32).

[182] W. Dai et al. “Trustzone-based Secure Lightweight Wallet for Hyperledger Fabric”. In: *Journal of Parallel and Distributed Computing* (2021) (cit. on p. 32).

[183] W. Dai et al. “SBLWT: A Secure Blockchain Lightweight Wallet based on TrustZone”. In: *IEEE Access* (2018) (cit. on p. 32).

[184] Z. Jian et al. “TSC-VEE: A TrustZone-Based Smart Contract Virtual Execution Environment”. In: *IEEE Transactions on Parallel and Distributed Systems* (2023) (cit. on p. 32).

[185] C. Müller et al. “TZ4Fabric: Executing Smart Contracts with ARM TrustZone:(Practical experience report)”. In: *Proc. of SRDS*. 2020 (cit. on p. 32).

[186] H. Park et al. “{StreamBox-TZ}: Secure Stream Analytics at the Edge With {TrustZone}”. In: *Proc. of USENIX ATC*. 2019 (cit. on p. 32).

[187] T. Brito, N. O. Duarte, and N. Santos. “Arm TrustZone for Secure Image Processing on the Cloud”. In: *Proc. of SRDSW*. 2016 (cit. on p. 32).

[188] Z. Ren et al. “Restricting the Number of Times that Data can be Accessed in Cloud Storage Using TrustZone”. In: *Proc. of CCGrid*. 2022 (cit. on p. 32).

[189] P. M. VanNostrand et al. “Confidential Deep Learning: Executing Proprietary Models on Untrusted Devices”. In: *arXiv preprint arXiv:1908.10730* (2019) (cit. on p. 32).

[190] F. Mo et al. “Darknetz: Towards Model Privacy at the Edge Using Trusted Execution Environments”. In: *Proc. of Mobile Systems, Applications, and Services*. 2020 (cit. on p. 32).

[191] Z. Liu et al. “Trusted-dnn: A Trustzone-Based Adaptive Isolation Strategy for Deep Neural Networks”. In: *Proc. of of the ACM Turing Award Celebration Conference-China*. 2021 (cit. on p. 32).

[192] R. Liu et al. “SecDeep: Secure and Performant on-device Deep Learning Inference Framework for Mobile and IoT Devices”. In: *Proc. of Internet-of-Things Design and Implementation*. 2021 (cit. on p. 32).

[193] M. S. Islam et al. “Confidential Execution of Deep Learning Inference at the Untrusted Edge with ARM TrustZone”. In: *Proc. of Data and Application Security and Privacy*. 2023 (cit. on p. 32).

[194] M. F. Babar and M. Hasan. “Real-Time Scheduling of TrustZone-enabled DNN Workloads”. In: *Proc. of CPS & IoT Security and Privacy*. 2022 (cit. on p. 32).

[195] M. Costa et al. “SecureQNN: Introducing a Privacy-Preserving Framework for QNNs at the Deep Edge”. In: *International Conference on Data Science and Artificial Intelligence*. 2023 (cit. on p. 32).

[196] A. A. Messaoud et al. “Shielding Federated Learning Systems Against Inference Attacks with ARM TrustZone”. In: *Proc. of International Middleware Conference*. 2022 (cit. on p. 32).

[197] E. Kuznetsov, Y. Chen, and M. Zhao. “SecureFL: Privacy Preserving Federated Learning with SGX and Trustzone”. In: *Proc. of SEC*. 2021 (cit. on p. 32).

[198] S. D. Yalew et al. “T2Droid: A TrustZone-Based Dynamic Analyser for Android Applications”. In: *Proc. of Trustcom/Bigdatase/icess*. 2017 (cit. on p. 33).

[199] S. Jeon and H. K. Kim. “TZMon: Improving Mobile Game Security with ARM TrustZone”. In: *Computers & Security* (2021) (cit. on p. 33).

[200] S. Han and J.-H. Park. “Shadow-box v2: The Practical and Omnipotent Sandbox for Arm”. In: *Blackhat Asia* (2018) (cit. on p. 33).

[201] X. Ge, H. Vijayakumar, and T. Jaeger. “SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture”. In: *IEEE Mobile Security Technologies Workshop*. 2014 (cit. on p. 33).

[202] A. M. Azab et al. “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. Scottsdale, Arizona, USA: ACM, 2014. isbn: 978-1-4503-2957-6 (cit. on p. 33).

[203] J. Williams. “Inspecting Data from the Safety of Your Trusted Execution Environment”. In: *BlackHat USA* (2015) (cit. on p. 33).

[204] S. Dong et al. “Kims: Kernel Integrity Measuring System Based on TrustZone”. In: *Proc. of Big Data Computing and Communications BIGCOM*. 2020 (cit. on p. 33).

[205] H. Beyers, M. Olivier, and G. Hancke. “Assembling Metadata for Database Forensics”. In: *IFIP International Conference on Digital Forensics*. 2011 (cit. on p. 33).

[206] X. Liu et al. “TZEAMM: An Efficient and Secure Active Measurement Method Based on TrustZone”. In: *Security and Communication Networks* (2023) (cit. on p. 33).

[207] T. Sechkova, E. Barberis, and M. Paolino. “Secure Location-Aware VM Deployment on the Edge Through OpenStack and ARM TrustZone”. In: *Proc. of EuCNC*. 2019 (cit. on p. 33).

[208] W. Li et al. “Building Trusted Path on Untrusted Device Drivers for Mobile Devices”. In: *Asia-Pacific Workshop on Systems*. ACM, 2014 (cit. on pp. 33, 64).

[209] A. Amiri Sani. “Schrodintext: Strong Protection of Sensitive Textual Content of Mobile Applications”. In: *Proc. of conference on mobile systems, applications, and services*. 2017 (cit. on p. 33).

[210] L. Guo and F. X. Lin. “Minimum Viable Device Drivers for ARM TrustZone”. In: *Proc. of Computer Systems*. 2022 (cit. on p. 33).

[211] H. Park and F. X. Lin. “Safe and Practical GPU Computation in TrustZone”. In: *Proc. of European Conference on Computer Systems*. 2023 (cit. on p. 33).

[212] C. M. Park et al. “Rushmore: Securely Displaying Static and Animated Images Using TrustZone”. In: *Proc. of Mobile Systems, Applications, and Services*. 2021 (cit. on p. 33).

[213] H. Sun et al. “Trustdump: Reliable Memory Acquisition on Smartphones”. In: *Proc. of Computer Security-ESORICS*. 2014 (cit. on p. 33).

[214] James Barclay and Robbie Small and Taylor McCaslin. *Humans Only: Duo Mobile and Android Protected Confirmation*. www.duo.com/blog/humans-only-duo-mobile-and-android-protected-confirmation. 2018 (cit. on p. 33).

[215] W. Li et al. “Adattester: Secure Online Mobile Advertisement Attestation Using TrustZone”. In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. 2015 (cit. on p. 33).

[216] K. Ying et al. “Truz-Droid: Integrating Trustzone With Mobile Operating System”. In: *Proc. of mobile systems, applications, and services*. 2018 (cit. on p. 33).

[217] K. Ying, P. Thavai, and W. Du. “Truz-View: Developing TrustZone User Interface for Mobile OS Using Delegation Integration Model”. In: *Proc. of Data and Application Security and Privacy*. 2019 (cit. on p. 33).

[218] K. Suzaki et al. “Ts-perf: General Performance Measurement of Trusted Execution Environment and Rich Execution Environment on Intel SGX, Arm TrustZone, and RISC-V Keystone”. In: *IEEE Access* (2021) (cit. on p. 33).

[219] C. Göttel, P. Felber, and V. Schiavoni. “iperftz: Understanding Network Bottlenecks for TrustZone-Based Trusted Applications”. In: *Proc. of Stabilizing, Safety, and Security of Distributed Systems*. 2019 (cit. on p. 33).

[220] H. Wang et al. “An OP-TEE Energy-Efficient Task Scheduling Approach Based on Mobile Application Characteristics”. In: *Intelligent Automation & Soft Computing* (2023) (cit. on p. 34).

[221] H. Wang et al. “ETS-TEE: An Energy-Efficient Task Scheduling Strategy in a Mobile Trusted Computing Environment”. In: *Tsinghua Science and Technology* (2022) (cit. on p. 34).

[222] Y. Li and D. Zeng. “Dependency-Aware Task Scheduling in TrustZone Empowered Edge Clouds for Makespan Minimization”. In: *IEEE Transactions on Sustainable Computing* (2023) (cit. on p. 34).

[223] B. Gowrisankar et al. “GateKeeper: Operator-Centric Trusted App Management Framework on ARM TrustZone”. In: *Proc. of CNS*. 2022 (cit. on p. 34).

[224] Global Platform. *Technology Document Library*. www.globalplatform.org/specs-library/?filter-committee=tee. 2018 (cit. on p. 34).

[225] Arm. *Arm Firmware Framework for Arm A-profile*. www.developer.arm.com/documentation/den0077/latest (cit. on pp. 34, 105).

[226] O. E. SDK. *Open Enclave SDK*. www.github.com/openenclave/openenclave. 2022 (cit. on pp. 34, 123).

[227] Teaclave. *Teaclave: A Universal Secure Computing Platform*. www.github.com/apache/incubator-teaclave. 2022 (cit. on pp. 34, 123).

[228] M. Brossard et al. *Private Delegated Computations Using Strong Isolation*. Technical report. Systems Research Group, Arm Research, 2022. doi: https://doi.org/10.48550/arXiv.2205.03322 (cit. on pp. 34, 123).

[229] Arm. *Trusted Services Documentation*. https://trusted-services.readthedocs.io/en/stable/. 2022 (cit. on p. 34).

[230] R. Pettersen, H. D. Johansen, and D. Johansen. “Secure Edge Computing with ARM TrustZone.” In: *Proc. of IoTBDS*. 2017 (cit. on p. 34).

[231] A. Atamli-Reineh et al. “Analysis of Trusted Execution Environment usage in Samsung KNOX”. In: *Proceedings of the 1st Workshop on System Software for Trusted Execution*. 2016 (cit. on p. 34).

[232] N. Koutroumpouchos, C. Ntantogian, and C. Xenakis. “Building Trust for Smart Connected Devices: The Challenges and Pitfalls of TrustZone”. In: *Sensors* (2021) (cit. on pp. 34, 35).

[233] F. Khalid and A. Masood. “Vulnerability Analysis of Qualcomm Secure Execution Environment (QSEE)”. In: *Computers & Security* (2022) (cit. on pp. 34, 35).

[234] A. Shakevsky, E. Ronen, and A. Wool. “Trust Dies in Darkness: Shedding Light on Samsung’s TrustZone Keymaster Design”. In: *Proc. of USENIX Security*. 2022 (cit. on p. 34).

[235] T. Cooijmans, J. de Ruiter, and E. Poll. “Analysis of Secure Key Storage Solutions on Android”. In: *Proc. of Workshop on Security and Privacy in Smartphones & Mobile Devices*. 2014 (cit. on p. 34).

[236] M. Busch, J. Westphal, and T. Müller. “Unearthing the {TrustedCore}: A Critical Review on {Huawei’s} Trusted Execution Environment”. In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. 2020 (cit. on p. 35).

[237] N. Liu et al. “On the Cost-Effectiveness of TrustZone Defense on Arm Platform”. In: *Proc. of Information Security Applications*. 2020 (cit. on p. 35).

[238] E. Benhani, L. Bossuet, and A. Aubert. “The Security of ARM TrustZone in a FPGA-based SoC”. In: *IEEE Transactions on Computers* (2019) (cit. on p. 35).

[239] W. Li, Y. Xia, and H. Chen. “Research on Arm TrustZone”. In: *GetMobile: Mobile Computing and Communications* (2019) (cit. on p. 35).

[240] X. Tan et al. “Where’s the ”up”?! A Comprehensive (bottom-up) Study on the Security of Arm Cortex-M Systems”. In: *arXiv preprint arXiv:2401.15289* (2024) (cit. on p. 35).

[241] Y. Zhang et al. “SHELTER: Extending Arm CCA with Isolation in User Space”. In: *Proc. of USENIX Security*. 2023 (cit. on p. 35).

[242] P. Nasahl et al. “HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment”. In: *Proc. of Asia CCS*. 2021 (cit. on p. 35).

[243] Y. Jia et al. “HyperEnclave: An Open and Cross-platform Trusted Execution Environment”. In: *Proc. of USENIX ATC*. 2022 (cit. on pp. 35, 103, 112, 116, 121, 122).

[244] S.-K. Han and J. Jang. “MyTEE: Own the Trusted Execution Environment on Embedded Devices.” In: *Proc. of NDSS*. 2023 (cit. on pp. 35, 103, 121, 122).

[245] D. Lee et al. “Keystone: An Open Framework for Architecting Trusted Execution Environments”. In: *Proc. of EuroSys*. 2020 (cit. on pp. 35, 66, 67, 100, 103, 114, 120, 121).

[246] A. Tang, S. Sethumadhavan, and S. Stolfo. “CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management”. In: *USENIX Security Symposium*. USENIX Association, 2017 (cit. on pp. 37, 61, 63, 66).

[247] *Arm Morello Program*. https://www.arm.com/architecture/cpu/morello. 2024 (cit. on p. 38).

[248] T. Van Strydonck et al. “CHERI-TrEE: Flexible Enclaves on Capability Machines”. In: *Proc. of EuroS&P*. 2023 (cit. on p. 38).

[249] A. Fitzek et al. “The ANDIX research OS - ARM TrustZone meets industrial control systems security”. In: *IEEE International Conference on Industrial Informatics*. 2015 (cit. on pp. 43, 48).

[250] N. Asokan et al. “ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018). issn: 0278-0070 (cit. on p. 43).

[251] P. Sparks. *The Route to a Trillion Devices*. White Paper, ARM. 2017 (cit. on p. 43).

[252] Dan Goodin. *Found: New Android malware with never-before-seen spying capabilities*. www.arstechnica.com/information-technology/2018/01/found-new-android-malware-with-never-before-seen-spying-capabilities/. 2018 (cit. on p. 43).

[253] Gal Beniamini. *War of the Worlds - Hijacking the Linux Kernel from QSEE*. www.bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html. 2016 (cit. on pp. 44, 46, 71).

[254] Daniel Komaromy. *Unbox Your Phone*. www.medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c. 2018 (cit. on pp. 44, 47, 57).

[255] Gal Beniamini. *Trust Issues: Exploiting TrustZone TEEs*. www.googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html. 2016 (cit. on pp. 44, 47, 48, 56).

[256] Di Shen. *Attacking your “Trusted Core” Exploiting TrustZone on Android*. www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android.pdf. 2015 (cit. on pp. 44, 47, 54, 55, 57).

[257] N. Zhang et al. “CacheKit: Evading Memory Introspection Using Cache Incoherence”. In: *IEEE European Symposium on Security and Privacy*. 2016 (cit. on pp. 44, 60, 61).

[258] R. Guanciale et al. “Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures”. In: *Proc. of S&P*. 2016 (cit. on pp. 44, 60, 61, 63, 66).

[259] N. Zhang et al. “TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices”. In: *IACR Cryptology ePrint Archive* (2016) (cit. on pp. 44, 60, 61, 63, 66).

[260] H. Cho et al. “Prime+Count: Novel Cross-World Covert Channels on Arm TrustZone”. In: *Proc. of Annual Computer Security Applications Conference*. 2018 (cit. on pp. 44, 60, 61).

[261] N. Jacob et al. “How to Break Secure Boot on FPGA SoCs Through Malicious Hardware”. In: *International Conference on Cryptographic Hardware and Embedded Systems*. Cham: Springer International Publishing, 2017 (cit. on pp. 44, 61, 63, 65).

[262] E. M. Benhani et al. “On the Security Evaluation of the ARM TrustZone Extension in a Heterogeneous SoC”. In: *IEEE International System-on-Chip Conference*. 2017 (cit. on pp. 44, 61, 63, 65, 66).

[263] Gal Beniamini. *Full TrustZone exploit for MSM8974*. http://bits-please.blogspot.com/2015/08/full-trustzone-exploit-for-msm8974.html. 2015 (cit. on pp. 46, 48, 71).

[264] Gal Beniamini. *Exploring Qualcomm's TrustZone implementation*. http://bits-please.blogspot.com/2015/08/exploring-qualcomms-trustzone.html. 2015 (cit. on p. 46).

[265] Gal Beniamini. *Android Linux Kernel Privilege Escalation Vulnerability and Exploit (CVE-2014-4322)*. http://bits-please.blogspot.com/2015/08/android-linux-kernel-privilege.html. 2015 (cit. on p. 46).

[266] Gal Beniamini. *Android Privilege Escalation to Mediaserver from Zero Permissions (CVE-2014-7920 + CVE-2014-7921)*. http://bits-please.blogspot.com/2016/01/android-privilege-escalation-to.html. 2016 (cit. on p. 46).

[267] Gal Beniamini. *Unlocking the Motorola Bootloader*. http://bits-please.blogspot.com/2016/02/unlocking-motorola-bootloader.html. 2016 (cit. on pp. 47, 71).

[268] D. Rosenberg. “QSEE Trustzone Kernel Integer Overflow Vulnerability”. In: *Black Hat conference*. 2014 (cit. on p. 47).

[269] Sean Beaupre. *TRUSTNONE*. http://theroot.ninja/disclosures/TRUSTNONE_1.0-11282015.pdf. 2015 (cit. on p. 47).

[270] Josh Thomas. *A story of Research for PacSec 2014*. www.pacsec.jp/psj14/PSJ2014_Josh_PacSec2014-v1.pdf. 2014 (cit. on p. 47).

[271] David Berard. *Kinibi TEE: Trusted Application Exploitation*. www.synacktiv.com/posts/exploit/kinibi-tee-trusted-application-exploitation.html. 2018 (cit. on pp. 47, 57).

[272] Nick Stephens. *Behind the PWN of a TrustZone*. www.slideshare.net/GeekPwnKeen/nick-stephenshow-does-someone-unlock-your-phone-with-nose. 2016 (cit. on p. 47).

[273] *Common Vulnerabilities and Exposures*. www.cve.mitre.org/about/ (cit. on p. 48).

[274] *Qualcomm Product Security - Security Advisories*. www.qualcomm.com/company/product-security/security-advisories (cit. on p. 48).

[275] *Code Aurora Forum Security Bulletin*. www.codeaurora.org/security-bulletin (cit. on p. 48).

[276] *Nvidia Product Security*. www.nvidia.com/en-us/security/ (cit. on p. 48).

[277] *Huawei Security Advisories*. www.huawei.com/en/psirt/all-bulletins (cit. on p. 48).

[278] *Samsung Mobile Security - Android Security Updates*. www.security.samsungmobile.com/securityUpdate.smsb (cit. on p. 48).

[279] *PS Vita*. http://www.vitahacker.com/ (cit. on p. 48).

[280] *Project Zero*. www.googleprojectzero.blogspot.com/ (cit. on p. 48).

[281] F. Basse. *Amlogic S905 SoC: Bypassing the (Not So) Secure Boot to Dump the BootROM*. www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html. 2016 (cit. on pp. 48, 59).

[282] G. Delugré and I. Arce. *Vulnerabilities in High Assurance Boot of NXP i.MX microprocessors*. www.blog.quarkslab.com/vulnerabilities-in-high-assurance-boot-of-nxp-imx-microprocessors.html. 2016 (cit. on p. 48).

[283] *Common Vulnerability Scoring System v3.0: Specification Document*. www.first.org/cvss/specification-document (cit. on p. 49).

[284] ARM. *Trusted Firmware-A - version 2.0*. www.github.com/ARM-software/arm-trusted-firmware. 2017 (cit. on p. 50).

[285] D. Wheeler. *SLOCCount*. www.dwheeler.com/sloccount (cit. on p. 52).

[286] A. Machiry et al. "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments." In: *Proc. of NDSS*. 2017 (cit. on pp. 57, 63, 64).

[287] ARM. *ARM Trusted Firmware Security Advisory TFV 3*. www.github.com/ARM-software/arm-trusted-firmware/wiki/ARM-Trusted-Firmware-Security-Advisory-TFV-3. 2017 (cit. on p. 58).

[288] J. Wiklander. *arm: sm: [bugfix] save/restore fiq core registers*. www.github.com/OP-TEE/optee_os/commit/f2dec49b7aeacd4cf36cacecc2a62ca925800e7a. 2016 (cit. on p. 58).

[289] J. Forissier. www.github.com/OP-TEE/optee_os/commit/93d3c451da7014193220c3f686c4b6379a1c5095. 2017 (cit. on p. 58).

[290] J. Forissier. *Storage: Protect TA Directory with a Mutex*. www.github.com/OP-TEE/optee_os/commit/b81882b289e70aa571b387f2b54aea853d74b31e. 2016 (cit. on p. 59).

[291] P. Carru. "Attack ARM TrustZone using Rowhammer". In: *GreHack*. 2017 (cit. on pp. 60, 62).

[292] S. Lee et al. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017. isbn: 978-1-931971-40-9 (cit. on p. 62).

[293] Y. Kim et al. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014 (cit. on p. 62).

[294] V. van der Veen et al. "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2016. isbn: 978-1-4503-4139-4 (cit. on p. 62).

[295] J. S. Jang et al. “SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment.” In: *Proc. of NDSS*. 2015 (cit. on pp. 63, 64).

[296] J. Jang and B. B. Kang. “Retrofitting the Partially Privileged Mode for TEE Communication Channel Protection”. In: *IEEE Transactions on Dependable and Secure Computing* (2018). issn: 1545-5971 (cit. on pp. 63, 64, 95–97, 100).

[297] E. Evenchick. “RustZone: Writing Trusted Applications in Rust”. In: (2018) (cit. on pp. 63, 65).

[298] R. R. Collins. *Intel’s System Management Mode*. http://www.rcollins.org/ddj/Jan97/Jan97.html. 1997 (cit. on pp. 66, 67).

[299] V. Costan, I. Lebedev, and S. Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *USENIX Security Symposium*. USENIX Association, 2016 (cit. on pp. 66, 67, 100, 123).

[300] Windows. *Virtualization-Based Security (VBS)*. www.docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs. 2017 (cit. on pp. 66, 67).

[301] Hex-Five. *Hex Five Security Adds MultiZone™ Trusted Execution Environment to the SiFive Software Ecosystem*. www.hex-five.com/2018/08/22/hex-five-adds-multizone-security-to-sifive-software-ecosystem/. 2018 (cit. on pp. 66, 67).

[302] J. V. Bulck et al. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution ”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018. isbn: 978-1-939133-04-5 (cit. on p. 67).

[303] Android. *Android Enterprise Security*. www.android.com/static/2016/pdfs/enterprise/Android_Enterprise_Security_White_Paper_2019.pdf. 2020 (cit. on p. 70).

[304] D. Cerdeira, J. Martins, N. Santos, S. Pinto. *ReZone Source Code and Experimental Results*. www.gitlab.com/ESRGv3/rezone/ (cit. on pp. 73, 90).

[305] Xilinx. *Isolate Security-Critical Applications on Zynq UltraScale+ Devices*. www.xilinx.com/support/documentation/white_papers/wp516-security-apps.pdf. 2020 (cit. on p. 77).

[306] Arm. *System Control Processor Firmware*. www.developer.arm.com/tools-and-software/open-source-software/firmware/scp-firmware. 2021 (cit. on p. 77).

[307] Qualcomm. *FY 2021 3rd Quarter Earnings Release*. www.investor.qualcomm.com/financial-information/historical-financial-results (cit. on p. 77).

[308] Jamie Cui. *Bitcoin Wallet implementation using Trusted Execution Environments*. www.github.com/Jamie-Cui/bitcoin-wallet. 2018 (cit. on pp. 88, 118).

[309] Linaro. *ClearKey OPTEE AOSP drm plugin*. www.github.com/linaro-mmwg/clearkeydrmplugin. 2019 (cit. on p. 88).

[310] Google. *ExoPlayer*. www.github.com/google/ExoPlayer (cit. on p. 88).

[311] Arm. *Realm Management Extension*. www.developer.arm.com/documentation/den0126/latest. 2021 (cit. on pp. 98, 100, 123).

[312] Arm. *Arm Architecture Reference Manual*. www.developer.arm.com/documentation/ddi0487/latest. 2022 (cit. on p. 98).

[313] Anandtech. *Arm Announces Mobile Armv9 CPU Microarchitectures: Cortex-X2, Cortex-A710 & Cortex-A510*. www.anandtech.com/show/16693/arm-announces-mobile-armv9. 2021 (cit. on p. 98).

[314] Arm. *Introducing Arm Confidential Compute Architecture*. www.developer.arm.com/documentation/den0125/latest. 2021 (cit. on p. 98).

[315] Qualcomm. *Snapdragon 820 Mobile Platform*. www.qualcomm.com/products/snapdragon-820-mobile-platform. 2022 (cit. on p. 99).

[316] Qualcomm. *Snapdragon 835 Mobile Platform*. www.qualcomm.com/products/snapdragon-835-mobile-platform. 2022 (cit. on p. 99).

[317] D. Kwon et al. "uXOM: Efficient eXecute-Only Memory on ARM Cortex-M". In: *Proc. of USENIX Security*. 2019 (cit. on p. 99).

[318] H. Janjua et al. "Towards a Standards-Compliant Pure-Software Trusted Execution Environment for Resource-Constrained Embedded Devices". In: *Proc. of SysTEX*. 2019 (cit. on p. 99).

[319] S. Pinto and C. Garlati. "Multi Zone Security for Arm Cortex-M Devices". In: *Embedded World Conference 2020*. 2020 (cit. on p. 99).

[320] M. Grisafi et al. "PISTIS: Trusted Computing Architecture for Low-end Embedded Systems". In: *Proc. of USENIX Security*. 2022 (cit. on p. 99).

[321] P. Maene et al. "Hardware-based Trusted Computing Architectures For Isolation and Attestation". In: *IEEE Transactions on Computers* (2017) (cit. on p. 100).

[322] P. Nasahl et al. "HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment". In: *Proc. of Asia CCS*. 2021 (cit. on pp. 100, 123).

[323] R. Bahmani et al. "CURE: A Security Architecture with CUstomizable and Resilient Enclaves". In: *Proc. of USENIX Security*. 2021 (cit. on pp. 100, 121, 123).

[324] C. Garlati and S. Pinto. "A Clean Slate Approach to Linux Security RISC-V Enclaves". In: *Embedded World Conference*. 2020 (cit. on p. 100).

[325] N. Dautenhahn et al. "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation". In: *Comput. Archit. News* (2015) (cit. on p. 100).

[326] V. Narayanan and A. Burtsev. "The Opportunities and Limitations of Extended Page Table Switching for Fine-Grained Isolation". In: *IEEE Security & Privacy* (2023) (cit. on pp. 104, 115).

[327] J. Gu et al. “A Hardware-Software Co-design for Efficient Intra-Enclave Isolation”. In: *Proc. of USENIX Security*. 2022 (cit. on pp. 106, 115, 116).

[328] A. Vasudevan et al. “Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework”. In: *Proc. of S&P*. 2013 (cit. on p. 110).

[329] F. Zhang et al. “Cloudvisor: Retrofitting Protection Of Virtual Machines In Multi-Tenant Cloud With Nested Virtualization”. In: *Proc. of SOSP*. 2011 (cit. on p. 110).

[330] J. T. Lim et al. “NEVE: Nested Virtualization Extensions for ARM”. In: *Proc. of SOSP*. 2017 (cit. on p. 110).

[331] B. Sá et al. “CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration”. In: *IEEE TVLSI* (2023) (cit. on pp. 112, 118).

[332] L. Valente et al. “Shaheen: An Open, Secure, and Scalable RV64 SoC for Autonomous Nano-UAVs”. In: *Hot Chips* (2023) (cit. on p. 112).

[333] Y. Xu, W. Cui, and M. Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. In: *Proc. of S&P*. 2015 (cit. on pp. 113, 120).

[334] J. Van Bulck, F. Piessens, and R. Strackx. “SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control”. In: *Proc. of SysTEX*. 2017 (cit. on pp. 113, 120).

[335] *RISCV: Initial support of QEMU RISC-V generic virtual platform (virt)*. www.github.com/OP-TEE/optee_os/pull/5734 (cit. on p. 114).

[336] J. Martins and S. Pinto. “Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems”. In: *Proc. of RTAS*. 2023 (cit. on pp. 117, 119).

[337] B. Blazevic et al. “TEEVseL4: Trusted Execution Environment for Virtualized seL4-based Systems”. In: *Proc. of RTCSA*. 2023 (cit. on pp. 121, 122).

[338] S. Zhao et al. “vSGX: Virtualizing SGX Enclaves on AMD SEV”. In: *Proc. of S&P*. 2022 (cit. on pp. 121, 122).

[339] C. Weinhold et al. “Towards Modular Trusted Execution Environments”. In: *Workshop on System Software for Trusted Execution*. 2023 (cit. on p. 123).

[340] M. Armanuzzaman and Z. Zhao. “BYOTee: Towards Building Your Own Trusted Execution Environments Using FPGA”. In: *arXiv preprint arXiv:2203.04214* (2022) (cit. on p. 123).

[341] D. Melotti, M. Rossi-Bellom, and A. Continella. “Reversing and Fuzzing the Google Titan-M Chip”. In: *Proc. of ROOTS*. 2021 (cit. on p. 123).

[342] Y. Cho et al. “Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices”. In: *Proc. of USENIX ATC*. 2016 (cit. on p. 123).

[343] X. Chen et al. “Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems”. In: *SIGPLAN Not.* (2008) (cit. on p. 123).

[344] J. M. McCune et al. “InkTag: Secure Applications on an Untrusted Operating System”. In: *Proc. of ASPLOS*. 2013 (cit. on p. 123).

[345] J. Yang. “Using Hypervisor to Provide Application Data Secrecy on a Per-Page Basis”. In: *Proc. of Virtual Execution Environments (VEE)*. 2008 (cit. on p. 123).

[346] J. McCune et al. “TrustVisor: Efficient TCB Reduction and Attestation”. In: *Proc. of S&P*. 2010 (cit. on p. 123).

[347] Google. *Asylo*. www.github.com/google/asylo. 2022 (cit. on p. 123).

[348] Enarx. *Enarx*. www.github.com/enarx/enarx. 2022 (cit. on p. 123).

[349] C.-C. Tsai, D. E. Porter, and M. Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX”. In: *Proc. of USENIX ATC*. 2017 (cit. on p. 123).

[350] S. Arnautov et al. “SCONE: Secure linux containers with intel SGX”. In: *Proc. of USENIX OSDI*. 2016 (cit. on p. 123).

[351] SOFAEnclave. *SOFAEnclave*. www.github.com/SOFAEnclave/SOFAEnclave. 2022 (cit. on p. 123).

[352] J.-Y. Gu et al. “Unified Enclave Abstraction and Secure Enclave Migration on Heterogeneous Security Architectures”. In: *Journal of Computer Science and Technology* (2022) (cit. on p. 123).