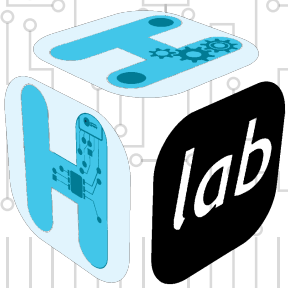
H2LAB-KNOW-TRUSTZONE
08-2026

# TrustZone: a comprehensive guide to its strengths and weaknesses




H$^2$Lab Guide





# Information

Created in 2020, H2Lab is a non-profit organization dedicated to designing, developing, and sharing open-hardware and open-source solutions for experimentation around embedded systems. Its main areas of work are:

- Design of hardware platforms for hardware and software analysis
- Development of open alternatives to proprietary, often opaque tools
- Publication of technical guides and recommendations to promote best practices in security, privacy, and sustainability across the ecosystem of connected devices and embedded systems

Support for researchers, teachers, and students through the provision of tools, educational content, and training.
Academic partnerships to encourage sharing of resources and knowledge.

## Revision history

| Version | Date | Author | Changes |
|---|---|---|---|
| 1.0 | November 2026 | H2Lab | Initial version of the guide |

# Contents

**Information** **i**

**1 Glossary and acronyms** **1**
1.1 Glossary . . . 1
1.2 List of acronyms . . . 2

**2 Purpose of this document** **5**
2.1 Objective . . . 5
2.2 Audience . . . 5
2.3 Scope and methodology . . . 6
2.4 Attacker models . . . 6
2.5 Guide structure . . . 7

**3 About TrustZone** **8**
3.1 History and evolution . . . 8
3.2 The arrival of TrustZone-M . . . 9
3.3 TrustZone and secure boot . . . 9
3.4 A brief perspective . . . 10

**4 TrustZone architectural principles** **12**
4.1 Evolution of processor cores . . . 12
4.1.1 Security state and the NS bit . . . 12
4.1.2 Exception levels and the monitor . . . 12
4.1.3 Interrupts and routing . . . 13
4.1.4 World switches and data exchange . . . 13
4.2 TrustZone and memory . . . 14
4.2.1 A split address space . . . 14
4.2.2 Partitioning memory: the TZASC . . . 14
4.2.3 Points to watch . . . 15
4.3 TrustZone and peripherals . . . 15
4.3.1 The role of the interconnect . . . 15
4.3.2 TZ-aware peripherals . . . 16
4.3.3 The debug case . . . 16

**5 Architectural limits of TrustZone** **17**
5.1 SoC integration: delegated security . . . 17
5.2 Non-TrustZone bus masters . . . 18
5.3 TrustZone and multicore architectures . . . 18
5.4 SMMU and PAMU: extending control to masters . . . 19
5.5 Hardware domain controllers . . . 20

5.6 Mapping a platform: the master–resource matrix . . . 21
5.7 The need for secure boot . . . 21
5.8 Summary of limitations . . . 22

**6 TrustZone and the evolution of attacks** **23**
6.1 Microarchitectural attacks . . . 23
6.1.1 Cache side channels . . . 23
6.1.2 Speculative execution . . . 24
6.1.3 Fault injection through power management . . . 24
6.1.4 Hijacking debug infrastructure . . . 24
6.1.5 DRAM disturbances . . . 25
6.2 Indirect attacks through DMA masters . . . 25
6.3 The secure world's software attack surface . . . 25
6.3.1 A larger-than-expected TCB . . . 25
6.3.2 The semantic gap: the BOOMERANG attack . . . 26
6.3.3 Poorly designed trusted cryptography: the Keymaster case . . . 26
6.4 Assessing the applicability of a published attack . . . 26
6.5 Lessons learned . . . 27

**7 TrustZone-M** **29**
7.1 Overview of TrustZone-M . . . 29
7.2 Differences from TrustZone-A . . . 30
7.3 Memory and peripherals . . . 31
7.4 Security, isolation, and ecosystem . . . 32
7.5 Development and porting pitfalls . . . 32
7.6 Limitations and attack lessons . . . 32
7.6.1 Boundary flaws: stack sealing and VLLDM . . . 33
7.6.2 Microarchitectural side channels without caches: BUSted . . . 33
7.6.3 Platform flaws: the LPC55S69 case . . . 33
7.6.4 Physical attacks . . . 33
7.6.5 Case study: assessing a TrustZone-M microcontroller product . . . 34

**8 Conclusion** **35**
8.1 What TrustZone guarantees, and under what conditions . . . 35
8.2 Sufficient, needs supplementation, or insufficient . . . 35
8.3 Assessment checklist . . . 36
8.4 Outlook . . . 37

**License** **41**



# 1
# Glossary and acronyms

## 1.1 Glossary

**Microarchitectural attack.** An attack that exploits observable behavior of a processor's internal resources (caches, predictors, execution queues, buses) rather than a software flaw, most often through timing measurements.

**Side channel.** An unintended information vector (execution time, power consumption, electromagnetic emissions, contention for a shared resource) that makes it possible to infer a secret without violating logical access controls.

**Chain of trust.** A sequence of software components, each of which verifies the integrity and authenticity of the next before handing over execution, from an immutable root of trust through to the application.

**Security domain.** A set of SoC masters and resources that share the same hardware access rights. TrustZone defines two domains. Domain controllers (xRDC, XMPU, etc.) define additional ones.

**Security state.** The processor's hardware state (*Secure* or *Non-secure*), which determines the security attribute of its transactions and the resources it can access.

**Fault injection.** Deliberately perturbing a circuit's operation (voltage, clock, electromagnetic pulse, laser) to corrupt an instruction or data, such as a security check.

**Interconnect.** An SoC bus fabric connecting masters (cores, DMA engines, accelerators) to slaves (memories, peripherals) and propagating transaction attributes, including the security attribute.

**Bus master.** A hardware component capable of initiating transactions on the interconnect: a compute core, DMA controller, GPU, network or USB controller, or radio coprocessor.

**Secure monitor.** Software running at the highest exception level of a Cortex-A core (EL3), responsible for switching between the normal and secure worlds and preserving their contexts.

**Normal world, secure world.** The two execution environments defined by TrustZone. The normal (*Non-secure*) world hosts the general-purpose OS and applications. The secure (*Secure*) world hosts the TEE and trusted services.

**Bus firewall.** A hardware component that filters transactions to a resource (memory range, peripheral) according to the security attribute, master ID, or domain, and rejects unauthorized accesses.

**TZ-aware peripheral.** A peripheral that samples the security attribute of transactions it receives, and sometimes those it emits, and adapts its behavior accordingly, as opposed to a peripheral protected as a whole by an external filter.

**Root of trust.** An immutable hardware or software element (boot ROM, public key in fuses) whose trust cannot be established by another component and on which the chain of trust relies.

**Secure boot.** A boot process in which each software stage is authenticated (using a cryptographic signature) before execution, ensuring platform integrity at power-on.

**TEE, REE.** *Trusted Execution Environment*: a trusted execution environment hosted in the secure world (secure OS and trusted applications). *Rich Execution Environment*: the general-purpose environment in the normal world (Linux, Android, RTOS).

**Trusted application (TA).** A program running in the TEE that exposes a service (key management, DRM, payment) to the normal world through a controlled interface.

**Veneer (gateway function).** A small code fragment placed in a transition region (NSC in TrustZone-M) that forms the only legitimate entry point from the non-secure world to a secure function.

**World switch.** A transition between the normal and secure worlds: via the EL3 monitor on Cortex-A, or via dedicated branch instructions on Cortex-M.

## 1.2 List of acronyms

**AES.** Advanced Encryption Standard.

**AHB, APB, AXI.** Advanced High-performance Bus, Advanced Peripheral Bus, Advanced eXtensible Interface: bus protocols in the AMBA family.

**AMBA.** Advanced Microcontroller Bus Architecture.

**API.** Application Programming Interface.

**CCA.** (Arm) Confidential Compute Architecture.

**CVE.** Common Vulnerabilities and Exposures.

**DMA.** Direct Memory Access.

**DRM.** Digital Rights Management.

**DVFS.** Dynamic Voltage and Frequency Scaling.

**ECDSA.** Elliptic Curve Digital Signature Algorithm.

**EL.** Exception Level (exception level, EL0 to EL3).

**FIQ, IRQ.** Fast Interrupt Request, Interrupt Request.

**GIC.** Generic Interrupt Controller.

**GPU.** Graphics Processing Unit.

**HSE.** Hardware Security Engine.

**IDAU.** Implementation Defined Attribution Unit.

**IOMMU.** Input/Output Memory Management Unit.

**JTAG, SWD.** Joint Test Action Group, Serial Wire Debug: debug interfaces.

**MMU.** Memory Management Unit.

**MPC, PPC.** Memory Protection Controller, Peripheral Protection Controller.

**MPSoC.** Multi-Processor System-on-Chip.

**MPU.** Memory Protection Unit.

**NS.** Non-secure (TrustZone security attribute).

**NSC.** Non-Secure Callable (region callable from the non-secure world).

**NVIC.** Nested Vectored Interrupt Controller.

**OTP.** One-Time Programmable (fuse memory).

**PAMU.** Peripheral Access Management Unit (NXP QorIQ).

**PSA.** Platform Security Architecture.

**PSCI.** Power State Coordination Interface.

**REE.** Rich Execution Environment.

**ROM.** Read-Only Memory.

**RTOS.** Real-Time Operating System.

**SAU.** Security Attribution Unit.

**SCR.** Secure Configuration Register.

**SMC.** Secure Monitor Call.

**SMMU.** System Memory Management Unit.

**SMP.** Symmetric Multi-Processing.

**SoC.** System-on-Chip.

**SPU.** System Protection Unit (Nordic Semiconductor).

**TA.** Trusted Application.

**TBBR.** Trusted Board Boot Requirements.

**TCB.** Trusted Computing Base.

**TEE.** Trusted Execution Environment.

**TF-A, TF-M.** Trusted Firmware-A, Trusted Firmware-M.

**TLB.** Translation Lookaside Buffer.

**TZASC.** TrustZone Address Space Controller.

**TZPC.** TrustZone Protection Controller.

**xRDC.** Extended Resource Domain Controller (NXP).





# 2
# Purpose of this document

## 2.1 Objective

For twenty years, TrustZone has been Arm's answer to the need for strong isolation between a general-purpose execution environment and sensitive services sharing the same processor. The technology is now present in almost all Cortex-A application processors and, since the Armv8-M architecture, in a growing share of Cortex-M microcontrollers. It is often presented in product specifications as a security feature in its own right, a formulation that fuels confusion: TrustZone is an isolation mechanism, not a security guarantee. Its actual effectiveness depends on the quality of its SoC integration, the presence of complementary mechanisms (secure boot, bus firewalls, domain controllers), the robustness of the software running in the secure world, and the threat model being considered.

This guide aims to give readers the means to assess what TrustZone does and does not provide on a given platform. It describes how the technology works at the core, memory, and peripheral levels, then analyzes its architectural limits, the published attack classes that exploit them, and the complementary mechanisms required for isolation to hold. The Cortex-A variants (called *TrustZone-A* hereafter) and Cortex-M variants (TrustZone-M) are treated separately wherever their mechanisms differ, which is true of most of the architecture.

This guide is a reference for analysis and decision-making. It is neither a TEE development tutorial nor an offensive exploitation manual: published attacks are described in terms of their principles, prerequisites, and impact, at the level needed to decide whether they affect a given product and how to protect against them.

## 2.2 Audience

This document is primarily intended for:

- system architects and embedded engineers who must choose a platform or design a product's security partitioning.
- security teams tasked with assessing an existing platform, building a threat model, or preparing for certification.

- product managers who must choose between TrustZone alone, TrustZone supplemented by a dedicated security component (secure element, HSE), or another isolation approach.

Readers are expected to be generally familiar with embedded systems architecture (buses, MMU or MPU, interrupts, boot chain). TrustZone-specific concepts are defined throughout the text and collected in the glossary in chapter 1.

## 2.3 Scope and methodology

The scope covers TrustZone for the A and M profiles of the Arm architecture, its integration into SoCs and MPSoCs, complementary isolation mechanisms (TZASC, TZPC, SMMU, domain controllers, MPC, and PPC), and publicly documented attack classes up to the time this guide was written. Related technologies (Arm CCA and its *realms*, discrete secure elements, proprietary enclaves) are mentioned only when they shed light on a TrustZone limitation or a design alternative.

This guide focuses on Arm's TrustZone solution. Comparable trusted-isolation concepts in RISC-V architectures are outside its scope for readability and analytical consistency.

The methodology is consistent throughout: first describe the mechanism and its prerequisites, then its observable effect or impact, its applicability to a real product, and finally the available protections and their limitations. Technical claims are based on Arm specifications, vendor documentation, and the original publications of the cited attacks. When public literature is incomplete or behavior depends on an undocumented integration choice, the guide says so explicitly rather than generalizing. A demonstration on one platform does not imply that another product is vulnerable: the conditions to check are given in each case.

## 2.4 Attacker models

The conclusions in this guide depend systematically on the attacker class under consideration: the same platform may be adequate against a remote software attacker and indefensible against a well-equipped laboratory. To avoid vague statements, the guide uses the taxonomy in table 2.1, which is referenced in the attack analysis (chapter 6) and the final decision framework (chapter 8).

| Class | Attacker | Typical capabilities |
|---|---|---|
| A1 | Software, unprivileged | Runs code in the normal world without privileges: malicious application, code reached through a remote vulnerability. Can invoke TEE interfaces and measure time. |
| A2 | Software, privileged | Controls the normal-world kernel: drivers, peripheral and regulator registers, programming of DMA masters accessible to the normal world. This is the nominal threat model TrustZone claims to withstand. |
| A3 | Physical, non-invasive | Has access to the product: external debug interfaces, disassembly, low-cost perturbations (voltage or clock glitches, electromagnetic pulses), physical measurements (power consumption, emissions). |
| A4 | Physical, invasive | Laboratory capabilities: component preparation, microprobing, circuit extraction and modification, imaging. Out of reach of any isolation on standard silicon. |

Table 2.1: Attacker taxonomy used in this guide.

As a first approximation, TrustZone targets classes A1 and A2. Chapter 6 shows that even within this nominal scope, integration and software-quality conditions must be met. Classes A3 and A4 are outside the model: when they are part of a product's threat model, TrustZone must be supplemented with components designed to withstand them.

## 2.5 Guide structure

Chapter 3 traces the history of TrustZone, its relationship with secure boot, and the arrival of TrustZone-M. Chapter 4 describes the architectural principles: core security state, memory partitioning, and propagation of the security attribute to peripherals. Chapter 5 analyzes architectural limits: dependence on SoC integration, non-TrustZone bus masters, multicore architectures, SMMUs and domain controllers, and the need for secure boot. Chapter 6 compares these limits with published attacks, from microarchitectural side channels to TEE software compromises. Chapter 7 covers TrustZone-M in detail, from its address-based attribution model to lessons learned from vulnerabilities published on the first microcontrollers. Chapter 8 synthesizes the material as an assessment framework and decision criteria.

**What this guide is not**



This guide is not a TEE programming manual or an exploitation guide. It does not replace the reference documentation for the SoC being assessed: security controller names, scope, and reset values vary between vendors, and only the component documentation (often subject to a confidentiality agreement) is authoritative for a given product.

# 3
# About TrustZone

## 3.1 History and evolution

Arm introduced TrustZone in 2003 as a security extension to the ARMv6 architecture, first implemented in the ARM1176JZ-S core. Its original motivation came from the mobile-phone market: operators and rights holders wanted handsets able to protect content (DRM), subscription secrets, and later payment transactions, without the cost of a dedicated security processor. The chosen principle was economical: rather than add a second processor, the architecture logically divides the existing processor into two execution environments, a *normal world* for the general-purpose operating system and a *secure world* for sensitive services, with hardware propagating isolation through to memory and peripherals [2].

The ARMv7-A architecture extended the technology across the Cortex-A family, starting with Cortex-A8. During this period, TrustZone became widespread in smartphones, where each manufacturer developed its own secure operating system: QSEE from Qualcomm, Kinibi from Trustonic, TEEGRIS from Samsung, and iTrustee from Huawei. GlobalPlatform standardized the interfaces by specifying the TEE architecture and trusted-application APIs [18].

ARMv8-A (2011) restructured the execution model around four exception levels, EL0 to EL3, and made the secure monitor a full exception level (EL3) [14]. In 2013, Arm published an open-source reference implementation of the EL3 software and boot chain, Trusted Firmware-A (TF-A) [36], which became the de facto standard on Cortex-A SoCs. The open-source TEE OP-TEE, maintained by Linaro, plays a comparable role in the secure world. In 2017, the Armv8.4-A extension added a virtualization level in the secure world (Secure EL2). This is separate from TrustZone's Secure/Non-secure separation mechanism: when implemented and used correctly with a secure hypervisor monitor, Secure EL2 can provide additional isolation between secure partitions. TrustZone alone, including on a multicore system, does not provide this isolation. The secure world remains a single trust domain.

Armv9-A (2021) introduced the Confidential Compute Architecture (CCA) and its *realms* [8]: physical address space is divided not into two but four domains (*Root*, *Secure*, *Realm*, *Non-secure*), with an additional hardware check (*Granule Protection Check*) after the MMU. TrustZone has not been replaced: it remains the foundation on which CCA relies and is still the only mechanism available on the vast majority of production embedded SoCs. Table 3.1 summarizes these milestones.

Two observations guide this historical overview. First, TrustZone is mature, well-supported, and inexpensive: isolation is provided by the core and interconnect already present, without

| Period | Architecture | TrustZone contribution |
|---|---|---|
| 2003–2004 | ARMv6 (ARM1176) | Initial security extensions: NS bit, monitor mode, and world separation. |
| 2005–2011 | ARMv7-A (Cortex-A8 and later) | Extension across the Cortex-A family, widespread deployment in mobile devices. |
| 2011 | ARMv8-A | Exception levels EL0–EL3, monitor at EL3, AArch64; TF-A and OP-TEE ecosystem. |
| 2015–2016 | Armv8-M (Cortex-M23/M33) | TrustZone-M: microcontroller variant, address-based attribution, no software monitor (chapter 7). |
| 2017 | Armv8.4-A | Secure EL2 extension: additional virtualization and isolation between secure partitions. This is not the basic isolation provided by TrustZone. |
| 2021 | Armv9-A (CCA) | Four physical address spaces, *realms*; TrustZone retained as the foundation. |

Table 3.1: Key milestones in the evolution of TrustZone.

significant additional silicon. Second, each successive evolution (Secure EL2, CCA, and, in software, TEE hardening) addresses weaknesses observed in real-world deployments, detailed in chapters 5 and 6: concentration of privileges in the secure world, TEE attack surface, and inadequacy of the two-domain model in complex SoCs.

## 3.2 The arrival of TrustZone-M

Announced in late 2015 with the Armv8-M architecture and then implemented in the Cortex-M23 and Cortex-M33 cores (2016), TrustZone-M brings the two-world principle to microcontrollers [11]. This time the motivation is IoT: low-cost connected devices need to protect identity keys, an update mechanism, and critical functions on components without an MMU and subject to strict real-time and power constraints.

This is not merely a scaled-down version: the mechanism changes in nature. On Cortex-A, security state is a processor state and world switches go through a software monitor. On Cortex-M, security is an address attribute, determined by hardware attribution units (SAU and IDAU), and switching worlds uses simple, controlled branches in a few cycles with no intermediate software [4]. This preserves deterministic interrupt latency, which is essential on microcontrollers, but shifts the security burden to static memory-partition configuration, with specific consequences analyzed in chapter 7.

The first TrustZone-M microcontrollers reached the market between 2018 and 2020: Microchip SAM L11 [26], Nordic nRF9160 [28], NXP LPC55S69 [29], and STMicroelectronics STM32L5 [34]. Most product families with Cortex-M33 cores and later have since followed. The software ecosystem is organized around Trusted Firmware-M (TF-M) [37] and the PSA Certified framework [15].

## 3.3 TrustZone and secure boot

TrustZone and secure boot are distinct but inseparable mechanisms, and confusing them is a recurring source of assessment errors. TrustZone isolates two execution environments at run time. It says nothing about the integrity of the software running in the secure world. Yet the configuration for all isolation — secure translation tables, bus-firewall programming, peripheral

attribution — is set by software at boot. If that software can be replaced or modified, the attacker does not bypass TrustZone: they take control of it and configure it as they wish.

Arm formalizes the expected boot chain in the *Trusted Board Boot Requirements* [5], implemented by TF-A: an immutable ROM (BL1) authenticates a second-stage bootloader (BL2), which in turn authenticates the runtime monitor (BL31), the TEE (BL32), and the normal-world bootloader (BL33). Each check relies on a certificate chain rooted in a public key whose hash is stored in OTP fuses. The root of trust is therefore the ROM plus the fuses, not TrustZone. TrustZone extends at run time the trust established at boot. Figure 3.1 illustrates this chain. Section 5.7 returns to the practical consequences of this dependency, and the H2Lab guide on JTAG/SWD interface security covers the debug aspect of the same issue[1].

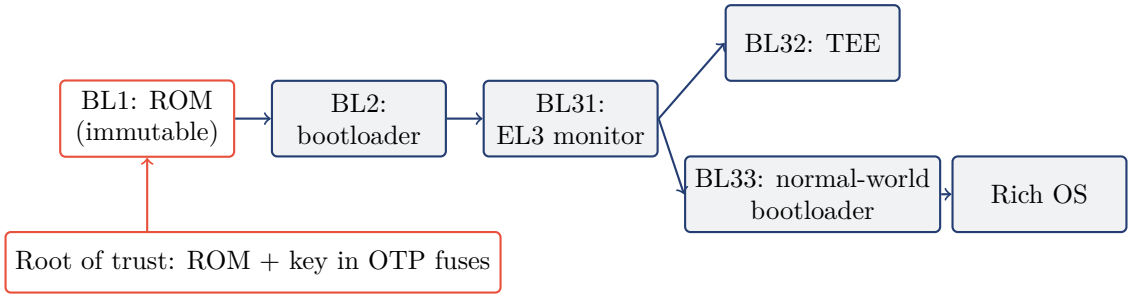



Figure 3.1: Trusted boot chain according to the TBBR model implemented by TF-A. Each stage authenticates the next. TrustZone extends this trust at run time.

**Key point**

Without secure boot, TrustZone protects only against an attacker unable to modify the platform software, which in a realistic threat model means practically no one. Assessment of a TrustZone platform therefore always starts with its boot chain and root of trust.

## 3.4 A brief perspective

Before examining the architecture, it is useful to situate TrustZone among the isolation mechanisms available in an embedded system. An MMU or MPU isolates processes under the control of a single kernel: it does not protect against that kernel itself. Virtualization isolates guest systems under the control of a hypervisor. TrustZone isolates two complete software stacks, including their respective kernels, with an intentional asymmetry: the secure world can observe and control the normal world, but never the reverse. Finally, a secure element or HSE physically isolates secrets on separate silicon or in a dedicated subsystem, with hardware countermeasures (sensors, shielding) that TrustZone does not provide.

These mechanisms are complementary, not competing, and the choice depends on the threat model: TrustZone provides logical access-control protection between two worlds on the same core. By design, it provides no guarantee against microarchitectural side channels (both worlds share the same caches and execution units) or a well-equipped physical attacker. Chapter 6 quantifies

[1] https://blog.h2lab.org/stigs/new-technical-guide-jtag-swd-security/

this using published attacks, while the H2Lab guide to security-component selection[2] offers an approach to choosing between these building blocks.

[2] https://blog.h2lab.org/stigs/new-technical-guide-security-components-selection/

# 4

# TrustZone architectural principles

This chapter describes the three pillars of TrustZone on application processors: the core's security state and world-switch mechanisms (section 4.1), memory partitioning (section 4.2), and propagation of isolation to peripherals (section 4.3). TrustZone-M, whose mechanisms differ substantially, is covered in chapter 7.

## 4.1 Evolution of processor cores

### 4.1.1 Security state and the NS bit

The founding principle of TrustZone is to make a single physical core act as two virtual security processors: at any instant, the core runs either in the *Secure* or the *Non-secure* state. This state is represented by the `NS` bit in the secure configuration register (`SCR_EL3` on Armv8-A), which is accessible only at the highest privilege level [14]. Security state is not limited to the core: every transaction emitted on the bus carries a security attribute derived from this state, making isolation visible end-to-end from the pipeline to the memory controller. A non-secure transaction that reaches a resource reserved for the secure world is rejected by hardware, usually with a bus error.

For the two worlds to coexist without corrupting one another, some system registers are banked: each world has its own MMU configuration registers, translation-table base registers, and exception vector. The secure world therefore sees a complete processor, with its own address translations and exception handling, overlaid on the processor seen by the normal world. General-purpose registers are not banked: saving and clearing them during world switches is a software responsibility, a point whose importance will recur throughout this guide.

### 4.1.2 Exception levels and the monitor

On Armv8-A, the execution model has four exception levels: EL0 (applications), EL1 (kernels), EL2 (hypervisors), and EL3, the highest privilege level, which hosts the secure monitor. EL0 and EL1 exist in both worlds. EL2 originally existed only in the normal world, until Secure EL2 was introduced in Armv8.4-A. EL3 structurally belongs to the secure world and controls the `NS` bit. Figure 4.1 illustrates this organization.

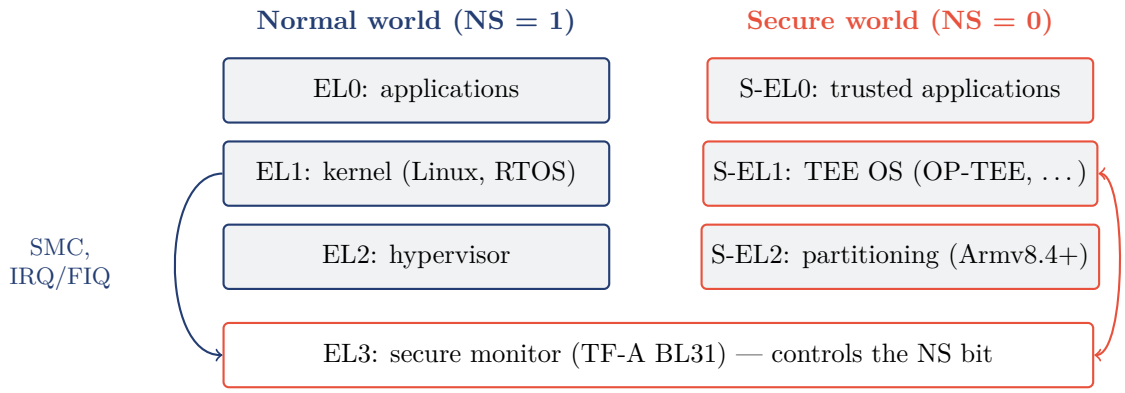



Figure 4.1: The two worlds and exception levels on Armv8-A. The EL3 monitor is the only transition point between worlds.

The monitor is the pivot of the architecture. Every world switch passes through it: it is reached by the `SMC` instruction (*Secure Monitor Call*) issued from EL1 or EL2, or by an interrupt configured to be routed to EL3. Its role is intentionally minimal: save the outgoing world's context (general-purpose registers and non-banked system registers), switch the `NS` bit, restore the incoming world's context, and return control to it. This minimality is a security requirement as well as a performance one: the monitor runs at the highest privilege, and every line of its code belongs to the platform's trusted computing base.

The SMC calling convention (SMCCC) standardizes the interface between worlds [12]: each service is identified by a function number, and arguments and return values are passed in general-purpose registers. The convention supports both fast, non-preemptible calls and long calls that may be interrupted. Arm's standard services use it, foremost PSCI [9], which delegates core power management to the monitor. This is appropriate, since powering a core down and waking it up directly affect the platform's security state (section 5.3).

### 4.1.3 Interrupts and routing

An interrupt may require a world switch: a secure peripheral must be serviced by the secure world even if the core was executing the normal world, and vice versa. The Generic Interrupt Controller (GIC) is therefore TZ-aware: each interrupt line is assigned to a group, secure (Group 0, historically signaled by FIQ) or non-secure (Group 1, signaled by IRQ). The normal world cannot observe or reconfigure secure-group interrupts, and secure interrupts can be assigned higher priority by design, ensuring that the secure world can regain control. The conventional routing scheme — FIQ to the secure world, IRQ to the normal world — is an integration convention, not an architectural requirement. The actual configuration must be checked during an assessment.

### 4.1.4 World switches and data exchange

A typical service call (for example, a TEE signature operation) follows this path: the normal-world application invokes a kernel driver, which prepares the arguments and issues an `SMC`. The monitor switches to the secure world. The TEE OS schedules the relevant trusted application. The result returns along the reverse path. Each switch has a cost, ranging from a few hundred cycles to a few microseconds depending on the SoC and what the monitor must save. This cost limits the reasonable granularity of secure services: TrustZone suits occasional services (key management, attestation, update verification), but not high-throughput stream encryption if

every block requires a switch. In the normal world, the call amounts to loading the function ID and arguments into registers and executing the `SMC` instruction; the monitor handles the rest:

```
    movw r0, #0x0004     /* function ID */
    movt r0, #0x3200     /* TEE service range */
    ldr  r1, =shm_base   /* shared buffer (NS) */
    smc  #0              /* to the EL3 monitor */
    /* return: r0 = status, r1-r3 = results */
```

Large data does not travel in registers. Instead, it is placed in shared memory, that is, a non-secure region accessible to both worlds — the secure world can also access non-secure memory. This asymmetry is the foundation of the model, but shared memory is also one of the most delicate parts of the interface: its contents remain under the potential attacker's control while the secure world examines them. Secure code must therefore copy data before validating it (to prevent *time-of-check/time-of-use* issues), validate that received pointers really refer to non-secure memory (otherwise it may become a write oracle into its own memory or the normal-world kernel, as exploited by BOOMERANG, section 6.3), and clear sensitive remnants from registers and caches before returning control.

# 4.2 TrustZone and memory

### 4.2.1 A split address space

From the perspective of the memory system, a transaction's physical address is augmented with its security attribute: it is as though the SoC had two parallel physical address spaces, one secure and one non-secure. Caches and TLBs store this attribute with each line or entry: a cache line loaded by a secure transaction is not served to a non-secure request, and caches do not need to be flushed on a world switch. This marking, essential for performance, has a direct security consequence: the two worlds physically share the same caches and contend for the same entries. Isolation is functional, not temporal — chapter 6 shows how side-channel attacks exploit this.

Each world translates addresses through its own set of tables: the MMU uses the translation registers of the current world. For each entry, the secure world's tables include an `NS` bit indicating whether the resulting access is issued as secure or non-secure. The secure world can therefore map and access non-secure memory (needed for data exchange), while the normal world always issues non-secure transactions, regardless of its tables. This asymmetry is structural and does not depend on any software configuration in the normal world.

### 4.2.2 Partitioning memory: the TZASC

The core emits attributes. The memory must still enforce them. Standard DDR memory knows nothing about TrustZone: filtering is performed upstream by a dedicated controller placed on the memory-controller path, the TZASC (*TrustZone Address Space Controller*), whose Arm reference implementation is the TZC-400 [3]. The TZASC divides DDR into regions and defines the read and write permissions for secure and non-secure transactions in each region. It can also refine filtering by master ID (*NSAID*), allowing, for example, a video decoder to read a protected region without opening it to the entire normal world. Violating transactions are rejected and may





raise an interrupt, which should be routed to the secure world and handled. A silent rejection would deprive the platform of a valuable compromise signal.

On-chip memories follow the same principle using other mechanisms: on-chip SRAM is partitioned by SoC-specific controllers or fixed divisions, and ROM and internal flash use comparable units. These controllers' configuration registers are themselves secure slaves and generally have lock bits that prevent further changes until the next reset. Correct integration requires programming and locking them early in the boot process.

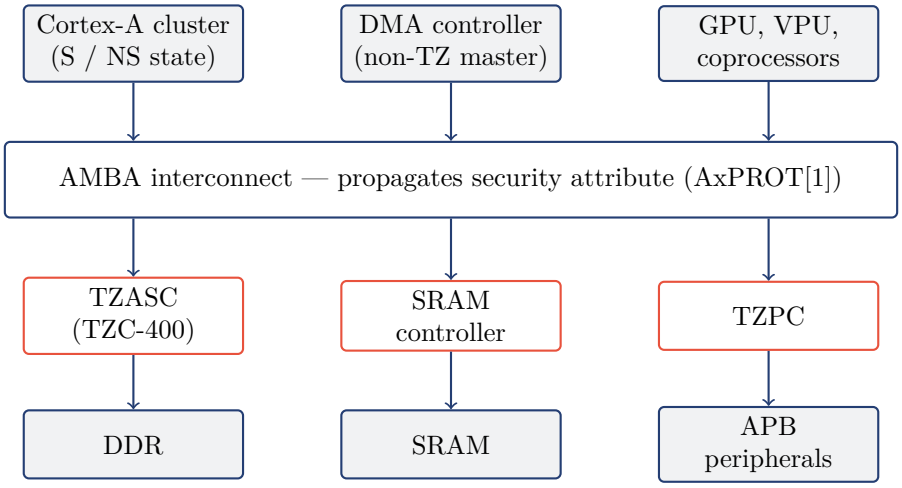



Figure 4.2: Propagation of the security attribute in an SoC. Firewalls (in red) filter transactions as close to the resources as possible. Non-TZ masters raise the question of which attribute they emit (chapter 5).

### 4.2.3 Points to watch

Memory partitioning is where the gap between the architecture and its integration is most apparent. Three areas account for most assessment errors. First, coverage: every physical access path to memory must pass through a filter. If the same DDR is accessible through a second, unfiltered port (an accelerator or display controller, for example), the TZASC on the primary port protects nothing. Second, configuration consistency: overlapping regions, overly coarse granularity that forces a secure region to be widened, and address aliases that expose the same physical memory through two addresses when only one is filtered. Third, locking: a correct configuration that can be changed by a compromised normal world is worthless. Lock bits must be set before control is first handed outside the trusted boot chain. Work on programmable SoCs has shown that a single interconnect configuration error can collapse the isolation of the entire system [16].

## 4.3 TrustZone and peripherals

### 4.3.1 The role of the interconnect

The security attribute of a transaction is carried across the SoC by the interconnect: on AXI, it is the `AxPROT[1]` signal [7]; on AHB, it is `HNONSEC`. The interconnect itself participates in enforcement: a fabric such as NIC-400 can be synthesized with security filtering on each slave port, rejecting non-secure transactions to slaves designated as secure. Peripheral protection is

therefore distributed among the interconnect, dedicated protection controllers, and the peripherals themselves — a distribution that varies from one SoC to another and is described precisely only in the component documentation.

The TZPC (*TrustZone Protection Controller*) complements this setup for simple peripherals: it provides configuration bits, programmable by the secure world, that dynamically assign each peripheral (or group of peripherals) on an APB bus to one of the two worlds. A peripheral assigned to the secure world becomes invisible to the normal world: all non-secure reads and writes are rejected. This assignment can change at run time — a touchscreen controller can be loaned to the secure world while a passcode is entered, then returned to the normal world.

### 4.3.2 TZ-aware peripherals

A peripheral is *TZ-aware* if it samples the security attribute of incoming transactions itself and adapts its behavior, rather than simply being hidden from one world. The GIC is the canonical example: it presents both worlds with different views of the same hardware, each limited to its own interrupt group. Cryptographic accelerators are another example: a TZ-aware AES engine can hold keys loaded by the secure world in slots inaccessible to non-secure requests, while remaining available to the normal world with other keys. Real-time clocks, system counters, and fuse controllers follow the same principle: some of their registers are visible only to the secure world.

A bus master can also be TZ-aware: a DMA controller designed for TrustZone emits, for each channel, transactions carrying the attribute of the world that programmed the channel. This may seem like a minor detail, but it is one of the main distinctions between robust TrustZone integration and a superficial one. A master without this capability must be constrained by other means, which is exactly where the limitations analyzed in the next chapter begin.

### 4.3.3 The debug case

Debug and trace interfaces cross the worlds by design: a debugger able to halt the core and read its registers in the secure state defeats all isolation. The architecture provides separate authorization signals for invasive and non-invasive debug of the secure world (`SPIDEN`, `SPNIDEN`), controlled by the component lifecycle (fuses, debug authentication). Proper management — secure debug disabled in production, re-enabling it subject to authentication — is the responsibility of the integrator and product manufacturer. Section 6.1 shows what happens when this is neglected, and the H2Lab JTAG/SWD guide details debug-authentication architectures.

# 5

# Architectural limits of TrustZone

TrustZone defines a model; it does not guarantee that the model is enforced. This chapter analyzes limits arising from the architecture itself and its integration into real SoCs: dependence on the integrator's work (section 5.1), bus masters outside the model (section 5.2), multicore architectures (section 5.3), mechanisms that extend control to masters — SMMUs and domain controllers (sections 5.4 and 5.5) — and dependence on secure boot (section 5.7). These are not hidden defects: they define the scope within which TrustZone delivers on its promises, and each corresponds to concrete attacks presented in chapter 6.

## 5.1 SoC integration: delegated security

Arm does not sell chips: TrustZone is delivered as architectural properties of the cores and infrastructure IP (interconnect, TZC, GIC) that each vendor assembles, extends, and configures. A platform's effective security therefore depends on decisions that the architecture leaves entirely open: which slaves are filtered and at what granularity, which masters can emit secure transactions, the reset values of protection controllers, which registers have lock bits, and which violations raise an interrupt. Two SoCs with the same Cortex-A cores can provide completely different levels of isolation.

This delegation has three practical consequences for an assessor. First, an assessment cannot rely on the Arm architecture alone: it requires the SoC security documentation, which is often incomplete in public manuals and partly restricted to confidential appendices. The absence of verifiable documentation is itself an assessment result. Second, the configuration-error surface is broad: research on programmable SoCs has shown that a security signal left at its default value in the interconnect configuration, or a peripheral omitted from a controller's scope, is enough to open a complete bypass [16]. Finally, the vendor's reference software (boot chain, adapted monitor, drivers) is part of the assessment scope: a correctly wired platform shipped with permissive software firewall settings is just as exposed as one that is incorrectly wired.

**Assessment point**



For each sensitive SoC resource (memory, peripheral, fuses, debug), identify the component that enforces the security attribute, its reset value, the software that configures it, and whether it is locked. Any resource whose protection cannot be traced this way must be considered unprotected.

## 5.2 Non-TrustZone bus masters

A modern SoC contains dozens of bus masters besides its cores: general-purpose DMA controllers, GPUs, video and display processors, USB, Ethernet and storage controllers, modems and radio coprocessors, and image-processing engines. The TrustZone model assumes that every transaction carries a relevant security attribute. Yet most of these masters are IP designed independently of TrustZone and cannot manage two security contexts. The integrator must choose between wiring the master's attribute to a fixed value and making it configurable by the secure world.

A master hardwired as non-secure is the safest case: it is confined to the normal world, firewalls keep it away from secure resources, and the only consequence is functional — the secure world cannot delegate transfers to that master. The dangerous case is a master with a configurable attribute, or worse, one hardwired as secure for functional convenience: any software able to program it effectively inherits its bus privileges. A DMA controller that emits secure transactions but can be programmed from the normal world is a read/write oracle into secure memory, which completely nullifies isolation (section 6.2). The corresponding integration rule is simple: a master's programming registers must be at least as well protected as the transactions it can emit. Exhaustively checking this, master by master, is one of the most valuable exercises in a platform assessment.

The problem extends beyond explicit DMA controllers: any coprocessor running its own firmware (modem, GPU, power-management controller) is a programmable master. If its transactions can be secure, its firmware — rarely audited and sometimes updated through separate channels — is effectively part of the trusted computing base. Domain controllers (section 5.5) and SMMUs (section 5.4) are structural responses to this problem.

## 5.3 TrustZone and multicore architectures

On an SMP platform, each core has its own security state: at a given moment, some cores may execute the normal world while others execute the secure world. The architecture supports this — transaction attributes and cache marking remain correct, including for coherence traffic [7] — but concurrency increases the software-error surface. The monitor and TEE must be reentrant or correctly serialize their critical paths. World-switch windows, during which a core handles both contexts, must be immune to activity on other cores. Global secure state (firewall configuration, TEE tables) must also be protected against concurrent access. Reference implementations (TF-A, OP-TEE) address these issues, but ports and proprietary monitors must be assessed against this criterion.





**Fundamental limit: one secure domain**



TrustZone distinguishes two states, Secure and Non-secure, not multiple independent secure domains. When two separate cores each run a TEE OS in the Secure state, their physical separation creates no security boundary between them: they belong to the same secure world and can access the Secure resources that the hardware integration makes available to them. TrustZone alone therefore cannot, in any circumstances, isolate two TEE OSes from one another, even if each is assigned to a different physical core. Presenting this configuration as isolation provided by TrustZone is a fundamental misunderstanding of its model. A separate, explicitly configured isolation mechanism is required (for example, Secure EL2 with a partitioning monitor, or hardware domain controllers covering the relevant cores, memories, and peripherals), and that mechanism must then be assessed in its own right.

Power management is the second point of friction: powering down, waking, or migrating a core affects the platform's security state, because a waking core executes code at a resume address that must be protected and its caches and registers must be returned to a safe state. This is the purpose of PSCI [9]: the normal world requests, the monitor performs. Any power management interface that bypasses EL3 (a clock or voltage control register accessible to the normal world) is an integration flaw — and the CLKSCREW attack (section 6.1.3) shows how it can become a fault-injection tool.

Finally, heterogeneous MPSoCs — Cortex-A clusters accompanied by Cortex-M or R cores, DSPs, and security enclaves — raise a scope question that is often misunderstood: TrustZone on the A cluster governs only the A cluster. From the interconnect's perspective, a companion Cortex-M is just another master. Its membership in a security domain is determined by SoC controllers (section 5.5), not by the security state of the A cores. Assessing "TrustZone security" in an MPSoC without mapping every master and its domain is like assessing the lock on a door in a wall full of holes.

## 5.4 SMMU and PAMU: extending control to masters

The SMMU (*System MMU*) [10] is Arm's response to the DMA-master problem: placed in the path of one or more masters, it translates their addresses and filters their accesses as an MMU does for a process. Each transaction stream has an ID (*StreamID*) and is associated with a translation context. The SMMU's configuration interface is itself partitioned between the secure and normal worlds, allowing the secure world to impose access windows on a master that the normal world cannot widen. NXP's PAMU serves a comparable role on QorIQ platforms, using per-master access windows.

The SMMU turns the non-TrustZone-master problem into a configuration problem, a real improvement but not an automatic solution. Its effectiveness depends on three conditions that are rarely all met. First, coverage: only masters actually placed behind an SMMU are constrained. On many SoCs only some masters are covered, and the documentation does not always make this clear. Second, configuration granularity: windows that are too broad and permanently open to

simplify drivers or preserve performance leave most of the risk intact. The Thunderclap family of attacks demonstrated this on IOMMUs in general-purpose systems, where oversized transfer windows exposed kernel structures to malicious peripherals [25]. Finally, cost: each SMMU TLB miss adds a page-table walk to the DMA path, which in practice encourages broad static configurations — precisely those that provide the least protection.

## 5.5 Hardware domain controllers

TrustZone's two-world model is too limited for MPSoCs: between "the normal world" and "the secure world", where should the real-time core that controls the machine go, or the modem with third-party firmware, or the multimedia subsystem? Vendors have responded with hardware domain controllers that generalize the mechanism: each SoC master is assigned to a domain (4, 8, or 16, depending on the implementation), each transaction carries its master's domain ID, and resource controllers (for memory and peripherals) grant or deny access by domain in addition to the TrustZone attribute. NXP calls these RDC and then xRDC on the i.MX and S32 families; Xilinx uses XMPU and XPPU on Zynq UltraScale+; Texas Instruments uses *firewalls* on the Sitara and Jacinto families. The TZC-400 itself, with its NSAID filtering, is an embryonic form of this approach [3]. Figure 5.1 illustrates the principle on a simplified MPSoC.

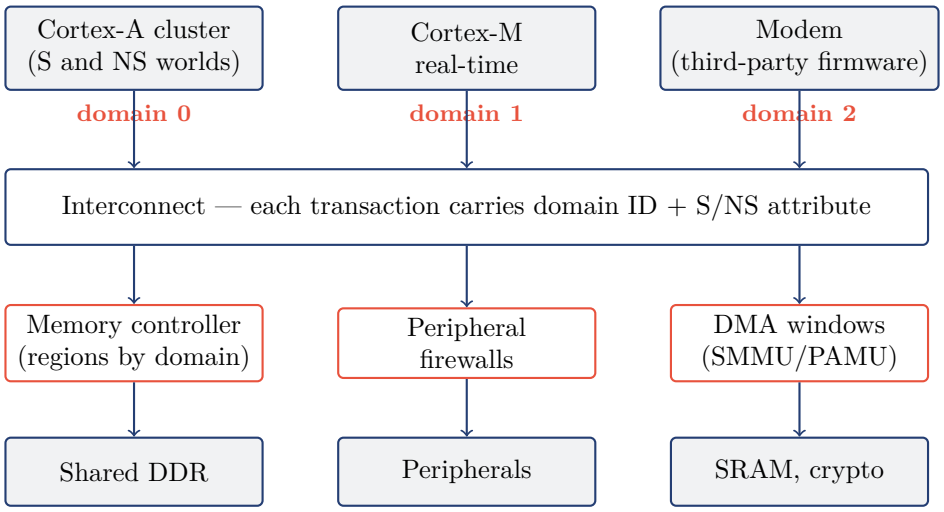



Figure 5.1: Domain controllers in a heterogeneous MPSoC. The secure-world software assigns masters to domains and sets each domain's rights to each resource at boot, then locks the configuration. The TrustZone attribute remains a separate control dimension, added to the domain.

These controllers are essential for serious isolation in an MPSoC: they confine the modem outside the application-core domain, or protect a safety core from application cores. But once again, they shift security toward integration and configuration: their models, granularities, and defaults are proprietary and not portable between vendors. Their configuration is at least as complex as that of TrustZone firewalls and must be locked at boot using the same discipline. The interaction between domain and TrustZone attribute (both dimensions coexist) is a common source of blind spots: a resource may be correctly filtered by domain but not by attribute, or vice versa. Assessing an MPSoC requires the complete master-by-resource matrix, including both dimensions.

## 5.6 Mapping a platform: the master–resource matrix

The preceding sections converge on one exercise, central enough to formalize: build a matrix for the platform under assessment, crossing each bus master with each sensitive resource, and compare it with the partitioning intended by the product architecture. The process has three steps. First, inventory all masters (cores in each of their states, DMA, accelerators, firmware-running coprocessors, debug interfaces acting as masters) and resources (memory regions by use, peripherals, fuses, and the protection controllers' own registers). Second, for each pair, determine the effective hardware decision — allowed, denied, or configuration-dependent — and the component that enforces it, using the SoC documentation. Finally, compare the result with the intended policy and classify each discrepancy. Table 5.1 sketches such a matrix for a simplified platform.

| **Master ↓ / Resource →** | **Secure DDR** | **Normal DDR** | **Crypto engine** | **Fuses** |
|---|---|---|---|---|
| Cortex-A, secure state | A | A | A | A (partial) |
| Cortex-A, normal state | R (TZASC) | A | A (NS keys only) | R (TZPC) |
| General-purpose DMA | **?** | A | **?** | R |
| GPU | R (NSAID) | A (window) | R | R |
| Modem (third-party firmware) | **?** | A (SMMU window) | R | R |

Table 5.1: Example master–resource matrix (A: allowed, R: denied by the indicated component, **?**: undocumented, to be established). The "?" cells are the main result of the exercise: each is an unqualified risk.

Experience shows that the value of this exercise lies less in the cells that can be filled in than in those the documentation leaves blank: each "?" represents either a documentation gap to resolve with the vendor, a test to run on target, or a risk to accept explicitly. The matrix then serves as a regression reference: any boot-software change that modifies a firewall configuration must be checked against the policy it implements.

## 5.7 The need for secure boot

Section 3.3 established the principle. Here we spell out its consequences as a limitation. The entire isolation configuration — firewalls, domain controllers, peripheral attribution, TEE tables — is the work of boot software. TrustZone is therefore only as robust as the chain that takes the platform from reset to that locked configuration. Three links concentrate the risk. First, the root of trust: boot ROM and fuse keys, whose vulnerabilities cannot be fixed. A ROM flaw permanently compromises every unit produced (section 7.6 gives a concrete case). Next, the intermediate stages: each bootloader must verify the next without logic or cryptographic errors and resist downgrades to vulnerable older versions, which requires anti-rollback counters in fuses. Finally, the transition window: between reset and firewall locking, the platform passes through a transient state in which protections are not yet in place. Interfaces active during this window (debug, recovery modes, booting from external media) must be disabled or authenticated.

The converse is also true: secure boot without TrustZone (or equivalent execution isolation)

guarantees software integrity at launch but protects nothing afterwards. The two mechanisms cover different phases of system operation, and the platform's security is that of the weaker one.

## 5.8 Summary of limitations

Table 5.2 summarizes the analyzed limitations, their consequences, and the measures that must be verified on the platform.

| **Limitation** | **Consequence if unaddressed** | **Measure to verify** |
|---|---|---|
| Two Secure TEE OSes on separate cores | No isolation between them from TrustZone alone; possible compromise of the other TEE and its Secure resources | Separate partitioning mechanism (Secure EL2/SPMC or hardware domain controller), with configuration and access policy verified |
| Security delegated to integration | Superficial isolation despite compliant cores | SoC security documentation, firewall mapping, reset values, locks |
| Non-TrustZone masters | Read/write access to secure memory via DMA programmable from the normal world | Attribute emitted by each master; protection of programming registers; SMMU or per-master firewalls |
| Multicore and power | Race windows in the monitor; fault injection via DVFS | Proven SMP monitor; power management confined to EL3 (PSCI) |
| Heterogeneous MPSoCs | Companion cores and coprocessors outside TrustZone's scope | Domain controllers (xRDC, XMPU…) covering all masters, locked configuration |
| No microarchitectural guarantee | Leakage through caches, speculation, contention (chap. 6) | Constant-time implementations, resource partitioning, dedicated analysis |
| Boot dependency | Monitor or TEE replacement before locking | Complete TBBR chain, anti-rollback, debug and recovery-mode management |

Table 5.2: Summary of TrustZone's architectural limitations.

# 6
# TrustZone and the evolution of attacks

TrustZone implements logical access control over transactions in an SoC. Attacks published over the past fifteen years do not contradict that control: they bypass it by operating in layers it does not govern — shared microarchitecture, circuit physics, and secure-world software interfaces. This chapter presents them by class, with the principle, prerequisites, published demonstrations, and available protections for each. Attacks specific to TrustZone-M are covered with it in chapter 7.

## 6.1 Microarchitectural attacks

### 6.1.1 Cache side channels

The two worlds share the same caches (section 4.2): NS marking ensures that a secure cache line is never *read* by the normal world, but does not make its *presence* undetectable. Cache occupancy can be measured: by filling a set of lines and timing its own accesses, an attacker in the normal world can observe which cache locations the secure world used in the meantime (the so-called *Prime+Probe* technique). If secure code performs secret-dependent memory accesses — AES substitution tables, RSA exponent windows, branches depending on nonce bits — this occupancy trace is enough to reconstruct the secret.

The ARMageddon research established in 2016 that the cache-attack toolkit developed on x86 also applies to Arm processors in mobile devices [21]. That same year, TruSpy demonstrated recovery of an AES key running in the secure world from the normal world, even from an unprivileged application [39]. In 2019, Keegan Ryan's *Cachegrab* attack demonstrated the product-level impact of this class: the cache trace of Qualcomm's hardware keystore ECDSA implementation (QSEE) leaked a few nonce bits per signature, enough to reconstruct the keystore's P-256 private key using lattice methods in an Android keystore "hardware-backed" [32]. The prerequisite is modest — run code in the normal world and trigger operations in the target service — and the target is TrustZone's central use case: protecting keys.

Protections belong in secure software, not TrustZone: constant-time implementations without secret-dependent memory accesses, blinding of nonce-based operations, and, where the SoC permits, cache partitioning. A platform assessment must therefore examine the quality of the TEE's cryptographic implementations — information that is rarely public, which argues for

auditable or certified TEEs. This class of attacks is analyzed in detail in the H2Lab guide to microarchitectural attacks[1].

### 6.1.2 Speculative execution

The Spectre and Meltdown attack families [20, 22] showed in 2018 that speculative execution can move data across privilege boundaries, with leakage then occurring through a cache side channel. TrustZone boundaries are affected just like other boundaries on out-of-order cores: a speculative gadget in monitor or TEE code may be triggered by the normal world (branch predictors are shared resources) and leak secure memory. Arm published lists of affected cores by variant and corresponding countermeasures, integrated into TF-A and kernels: speculation barriers and predictor sanitization on world switches. For an assessor, there are two checks: is the core in use among those affected, and does the firmware version include the corresponding mitigations? These have a cost, since predictor sanitization makes each world switch more expensive.

### 6.1.3 Fault injection through power management

CLKSCREW introduced a distinctive class in 2017: fault injection without physical access [35]. Frequency and voltage regulators (DVFS) in mobile SoCs are controlled by software, often without hardware safeguards against unstable operating points. From the normal-world kernel, the authors imposed out-of-range voltage-frequency pairs for a few microseconds, specifically targeting secure-world execution on another core. The induced faults enabled them to extract an AES key from the TEE using differential fault analysis, then to make an unsigned trusted application pass by corrupting RSA verification, on a commercial device (Nexus 6). VoltJockey extended the voltage-based demonstration to other platforms [30].

The prerequisite — normal-world kernel privileges — is exactly what TrustZone claims to protect against. These attacks therefore break the nominal model, not a marginal use case. The root cause is power management designed without security in mind: DVFS registers are effectively shared between worlds because their physical effects do not know about the NS bit. Protections are a matter of integration: hardware-bounded operating ranges, regulation delegated to EL3 or to a dedicated controller beyond the normal world's reach, and glitch detectors. During an assessment, the question "who can write the clock and voltage registers?" must have a documented answer.

### 6.1.4 Hijacking debug infrastructure

The Nailgun study showed in 2019 that a core can use the bus-accessible (*memory-mapped*) CoreSight debug infrastructure to take control of another core's debug facilities — including inspection of its secure execution — when secure-debug authorization signals are left open. This was the case on several commercial SoCs and development boards, and the attack was carried through to extracting a fingerprint from a commercial phone's TEE [27]. The lesson goes beyond this example: "sideways" paths to core state (inter-processor debug, trace, performance monitors) are part of the security perimeter and must be governed by the component lifecycle as rigorously as the external JTAG port. Shared performance counters require the same scrutiny: they provide precise timers to microarchitectural attackers.

[1] https://blog.h2lab.org/stigs/new-technical-analysis-microarchitectural-attacks/

### 6.1.5 DRAM disturbances

Security marking stops at the boundaries of physics. Rowhammer — bit flips in DRAM rows adjacent to a repeatedly accessed row — has been demonstrated on Arm mobile platforms, gaining root privileges from an Android application with no permissions [38]. A TZASC filters transaction addresses. It cannot prevent electrical disturbances from affecting cells in an adjacent secure region. Applicability to a given TrustZone target depends on the DRAM technology, controller protections (adaptive refresh, ECC), and memory layout. In the absence of a direct public demonstration against a TEE, the risk should be classified as theoretical but credible on platforms with vulnerable DRAM and addressed through memory selection rather than software.

## 6.2 Indirect attacks through DMA masters

Section 5.2 described the limitation. Public demonstrations confirm that it is exploitable. On general-purpose systems, the Thunderclap family showed that a malicious or compromised peripheral communicating over DMA can exploit windows left open by the IOMMU to read and corrupt system memory [25]. The issue transfers directly to embedded systems, with two aggravating factors. First, many embedded SoCs have no SMMU at all, so masters are confined only by static firewalls whose configuration is rarely audited. Second, the masters at issue are not user-pluggable cards but internal coprocessors (modem, Wi-Fi, GPU) running proprietary firmware, some remotely accessible, whose compromise has been documented repeatedly in industry.

A typical attack is indirect: first compromise a coprocessor through its own attack surface (radio stack, network interface), then use its bus privileges to reach secure-world memory or the normal-world kernel. An assessment must therefore treat each programmable master as a potential attacker and verify what the firewall matrix actually allows it to reach — an exercise that overlaps with the mapping required in section 5.5. Protections are those in the previous chapter: a finely configured SMMU, locked per-master firewalls, and a default refusal for any master that emits secure transactions if it can be programmed outside the secure world.

## 6.3 The secure world's software attack surface

### 6.3.1 A larger-than-expected TCB

In theory, TrustZone reduces the trusted computing base to a monitor and a small secure OS. In practice, commercial TEEs have accumulated services — key management, DRM, payments, biometrics, attestation — and their TCBs amount to hundreds of thousands of lines of C running mostly at high privilege. Cerdeira et al.'s systematic study catalogued and analyzed 207 published vulnerabilities in major TEEs (Qualcomm, Trustonic, Samsung, Nvidia, Huawei): memory overflows in trusted applications, insufficient validation of parameters received from the normal world, and persistent architectural weaknesses — lack of ASLR and basic memory protections in several secure OSes, trusted applications that could be revoked but effectively reactivated by downgrading their version [17]. Dynamic analysis campaigns using emulation

(PARTEMU) confirmed the finding by discovering exploitable flaws in trusted applications deployed at scale [19].

The point deserves to be stated plainly: the TrustZone boundary holds, but the software behind it is ordinary software, with ordinary software defect rates — and often fewer opportunities for auditing, monitoring, and updates than in the normal world. Adding a service to the TEE does not automatically make the product more secure: it moves code into a more privileged and less observable environment.

### 6.3.2 The semantic gap: the BOOMERANG attack

The boundary itself creates a class of flaws that would not exist without it. A trusted application receives requests from the normal world containing pointers to shared memory (section 4.1.4). The secure world can write anywhere in non-secure memory: if it does not ensure that received pointers remain within the calling application's legitimate region, an unprivileged application can make it write to the normal-world kernel's memory. The TEE becomes a *confused deputy* that elevates the privileges of its own attacker. The BOOMERANG attack demonstrated this pattern simultaneously on several commercial TEEs. Its cause was a semantic gap: the secure world does not know the internal boundaries (processes, kernel) of the normal world and cannot enforce them without explicit cooperation [24]. Protections include having the normal-world kernel validate buffers before transmission and strictly bounding the accessible ranges on the TEE side. These protections are now present in commercial TEEs in general, but the pattern remains a design trap for any custom secure service.

### 6.3.3 Poorly designed trusted cryptography: the Keymaster case

Analysis of Samsung's Keymaster (USENIX Security 2022) illustrates one final lesson: the label "hardware-protected" says nothing about cryptographic design. Keystore keys, sealed by the trusted application, were encrypted with AES-GCM using an initialization vector controlled by the normal world on some generations. Reusing an IV allowed recovery of key material, and a compatibility mechanism allowed newer keys to be rolled back to the vulnerable format (a downgrade attack), leading to a demonstrated bypass of FIDO2 authentication [33]. No TrustZone boundary was crossed: the attack stayed entirely within the model's rules. Isolation is no substitute for cryptographic review or version management for sealed formats.

## 6.4 Assessing the applicability of a published attack

Attack publications regularly prompt hasty conclusions in both directions: "TrustZone is broken" or, conversely, "that only affects the platform in the paper." Neither is an assessment. A useful analysis of a publication follows five questions.

First, what is the nature of the result: a primitive (a few leaked bits, a fault exploitable in a lab, a constrained write) or a complete exploit chain on a real product? A demonstrated primitive is a serious signal, but the cost of turning it into an actual compromise may remain high. Publications rarely quantify this. Second, what are the prerequisites, compared with the classes in table 2.1: required normal-world privileges, physical access, co-residence on a core, number of measurements, and repeatability? An attack requiring millions of measurements against a service

whose throughput is limited by the product changes status. Third, how platform-dependent is it: exact core and revision (Arm errata and CVEs are indexed by revision), presence of the exploited shared resource, and the integration choice at fault? A result obtained on a development board with open debug does not automatically transfer to a locked-down product. Fourth, what is the state of countermeasures: is there a firmware fix, build option, or silicon revision, and does the assessed platform include it? Finally, does the attack generalize by revealing a class of flaw (interface validation, unpartitioned shared resource) for which other instances should be sought on the platform beyond the demonstrated case?

This questionnaire turns a publication into decision inputs: either the platform is outside the demonstrated scope and the risk is qualified with its uncertainties, or it is within scope and the publication provides the prerequisite, impact, and usually the countermeasure to verify.

## 6.5 Lessons learned

Table 6.1 summarizes the attack classes, their prerequisites, and the corresponding protections.

| **Class** | **Published examples** | **Prerequisites** | **Protections** |
|---|---|---|---|
| Cache side channels | TruSpy, ARMageddon, Cachegrab/QSEE [39, 21, 32] | Unprivileged execution in the normal world; non-constant-time secure code | Constant-time implementations; cache partitioning; TEE audit |
| Speculation | Spectre and variants [20] | Affected speculative core; firmware without mitigations | Arm mitigations in TF-A/kernel; core selection |
| DVFS faults | CLKSCREW, VoltJockey [35, 30] | Normal-world kernel privileges; accessible regulators | DVFS confined to EL3; hardware bounds; detectors |
| Side-channel debug | Nailgun [27] | Secure debug not locked; kernel access | Debug lifecycle; authentication; SPIDEN/SPNIDEN audit |
| DRAM disturbance | Drammer [38] | Susceptible DRAM; controlled access patterns | Memory choice (TRR, ECC); qualification |
| DMA masters | Thunderclap (transferable) [25] | Compromised or programmable master; permissive firewalls | Fine-grained SMMU; locked per-master firewalls |
| TEE software | SoK 207 vulns, BOOMERANG, PARTEMU, Keymaster [17, 24, 19, 33] | Access to the TEE interface from the normal world | Minimal TCB; parameter validation; anti-downgrade; crypto review |

Table 6.1: Attack classes against TrustZone-A: prerequisites and protections.

Three lessons shape a product assessment. First, the most damaging attacks did not violate TrustZone access control: they exploited what the model does not cover (microarchitecture, physics, debug) or the software it hosts. Assessing "TrustZone" therefore makes sense only as part of assessing the entire platform. Second, the prerequisites for published attacks are mostly realistic — unprivileged code or a compromised kernel in the normal world, precisely TrustZone's nominal threat model. Third, secure-software quality dominates residual risk: with the same hardware integration, a minimal, audited, up-to-date TEE changes the assessment outcome. A bloated and opaque TEE condemns it.

**Key point**



TrustZone draws an access-control boundary. Published attacks go over it (TEE software), under it (microarchitecture, physics), or around it (DMA masters, debug). A threat model that assumes TrustZone protects against all three without platform-by-platform verification is invalid.

# 7
# TrustZone-M

## 7.1 Overview of TrustZone-M

TrustZone-M is the security extension to the Armv8-M architecture, optionally implemented in the Cortex-M23, M33, M35P, M55, and M85 cores [11]. It reuses the two-world terminology but changes the mechanism: on a microcontroller, security is an attribute of the *address*, not a global state controlled by a monitor. The physical address space is unique (there is no MMU) and partitioned into secure, non-secure, and secure-callable (NSC) regions. The core's security state at any instant follows from the address of the code currently executing. Two hardware units determine this partition: the IDAU (*Implementation Defined Attribution Unit*), fixed by the microcontroller manufacturer and wired into the silicon, and the SAU (*Security Attribution Unit*), programmable by secure software across a limited number of regions (typically eight). The effective attribute for an address is the more restrictive of the two [4].

Each world has its own stacks (banked `MSP` and `PSP` pointers, with stack-limit registers), its own optional MPU, its own vector table, and two privilege levels. The model therefore provides four contexts (secure or non-secure, privileged or unprivileged). The NVIC assigns each interrupt to one of the worlds (the `NVIC_ITNS` register), and hardware automatically saves and masks registers when a non-secure exception occurs during secure code, so that residual state is not exposed.

World switching is direct and implemented in hardware. A call from the non-secure world to the secure world must land in an NSC region on an `SG` (*Secure Gateway*) instruction: any branch to a secure address outside these entry points raises a fault. Exposed secure functions are therefore preceded by veneers generated by the compilation toolchain (CMSE extension), which provide an exhaustive list of entry points. In the other direction, secure code calls the non-secure world using `BLXNS` and `BXNS`; hardware masks registers and pushes a special return address (`FNC_RETURN`) on the secure side. There is no monitor and no system call: the transition takes a few cycles and preserves deterministic latency, including for interrupts, which can preempt directly into the other world. A secure entry point is thus represented by a two-instruction veneer generated by the toolchain:

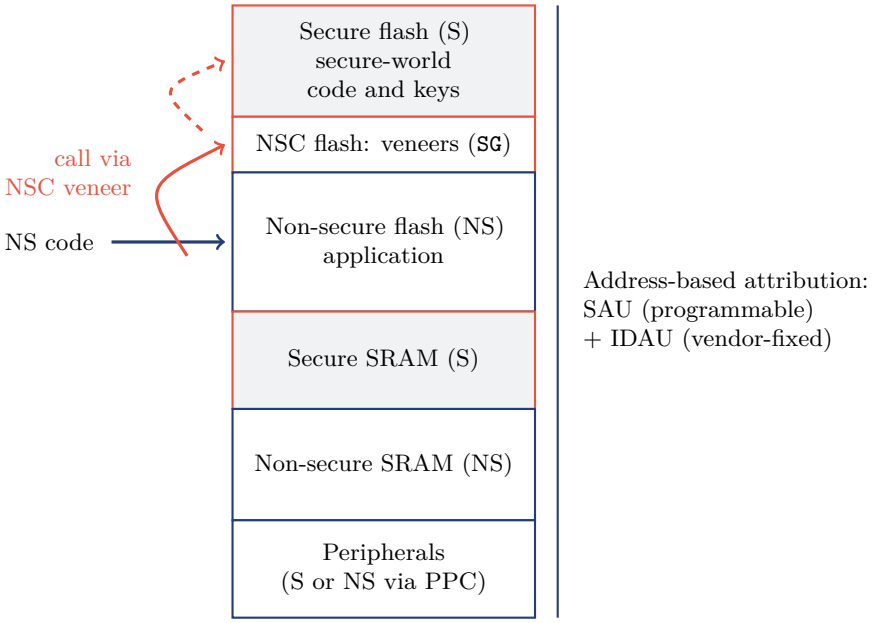



Figure 7.1: Typical partitioning of a TrustZone-M microcontroller. The non-secure world can enter secure code only through veneers in the NSC region, whose first instruction is `SG`.

```
    .global tz_sign
tz_sign:                        /* in the NSC region        */
    sg                          /* sole entry point         */

    b.w   __acle_se_tz_sign     /* to the S region          */
```

# 7.2 Differences from TrustZone-A

Table 7.1 summarizes the structural differences between the two variants.

| | **TrustZone-A (Cortex-A)** | **TrustZone-M (Cortex-M)** |
|---|---|---|
| Security attribution | Core state (NS bit in `SCR_EL3`) | Address attribute (SAU + IDAU) |
| World switch | Exception to EL3 monitor (`SMC`, interrupts) | Controlled branches (`SG`, `BLXNS`/`BXNS`), no intermediate software |
| Switching cost | Hundreds of cycles (software context save) | A few cycles, deterministic latency |
| Memory translation | Per-world MMU, separate tables | No MMU; per-world MPU |
| Privilege levels | EL0–EL3, Secure EL2 possible | Privileged/unprivileged in each world |
| Typical software stack | TF-A (EL3) + TEE (OP-TEE…) + rich OS | TF-M (or equivalent) + RTOS or bare metal |
| Interrupts | GIC, secure/non-secure groups | NVIC, per-line assignment (`ITNS`), preemptible priorities (`PRIS`) |
| Partitioning | Dynamic (tables, reprogrammable TZASC) | Essentially static, fixed at boot |

Table 7.1: Comparison of TrustZone-A and TrustZone-M mechanisms.

Two consequences deserve emphasis. The first is positive: removing the monitor eliminates

a critical software component, and the mostly static address-attribution model is amenable to analysis. The transition surface is reduced to the list of veneers, which can be checked when building the binary. The second is less favorable: the security boundary is now a *function-call* boundary, potentially crossed thousands of times per second, with arguments passed in registers and pointers. Every secure entry function faces the same requirements as a kernel system call: validate that pointers belong to the non-secure world (using `TT` instructions and dedicated CMSE routines), control reentrancy, and avoid leaking state in registers on return. The compiler helps (register clearing, veneers), but semantic validation of arguments remains the developer's responsibility. The BOOMERANG class of flaws (section 6.3) transfers directly, at a higher call frequency.

## 7.3 Memory and peripherals

The SAU and IDAU attribute only core accesses. As on Cortex-A, isolation must be extended through the SoC. TrustZone-M microcontrollers therefore include memory protection controllers (MPCs) that partition flash and SRAM into blocks assignable to either world, and peripheral protection controllers (PPCs) that assign each peripheral, with the AHB5 bus propagating the security attribute (`HNONSEC`). Consistency between the SAU/IDAU map and MPC/PPC configuration is the responsibility of secure initialization software: a region declared secure at the core but left open in the controller is accessible to every non-secure master — a typical TrustZone-M porting error.

Implementations vary substantially between vendors, down to granularity and terminology: GTZC (MPCBB for SRAM blocks, TZSC, TZIC for illegal events) from STMicroelectronics [34]; a *Secure AHB Controller* with per-master and per-slave rules and per-master security levels on NXP LPC55 [29]; a unified SPU (memory, peripherals, GPIO pins) from Nordic [28]; and flash attribution by rows with a reduced scope on the SAM L11 Cortex-M23, which lacks an SAU [26]. Non-core masters — especially DMA controllers, but also radios and accelerators — raise the same question as on Cortex-A: their transaction attribute and the protection of their programming registers must be checked master by master. Most microcontroller DMA controllers are configurable per channel, with an attribute inherited from the world that programmed them. Finally, most platforms allow individual I/O pins to be assigned to a world, making it possible to reserve a button, LED, or bus to an external secure element for the secure world. Table 7.2 summarizes these variants.

| Family | SoC isolation mechanisms | Notable points |
|---|---|---|
| STMicroelectronics STM32L5/U5 [34] | GTZC: MPCBB (SRAM in blocks), TZSC (peripherals), TZIC (illegal events) | Dedicated violation interrupt; flash partitioning by *watermark* |
| NXP LPC55S6x [29] | Secure AHB matrix, MPC/PPC, per-master security levels | Extensive ROM services; CVE-2021-31532 lessons (section 7.6) |
| Nordic nRF9160, nRF5340 [28] | Unified SPU: flash, RAM, peripherals, GPIO pins | Consistent model; individual pin assignment |
| Microchip SAM L10/L11 [26] | Cortex-M23 without SAU: IDAU partition fixed by fuses | Flash granularity by row; reduced core scope |

Table 7.2: TrustZone-M isolation mechanisms from four vendors. Names and granularities are not portable between families.

## 7.4 Security, isolation, and ecosystem

The typical use of TrustZone-M has converged on a simple partition: the secure world hosts a root of execution trust — secure boot, key storage, cryptography, attestation, updates — and the non-secure world hosts the application, including the RTOS. The reference implementation is Trusted Firmware-M [37], which provides these services behind PSA APIs, with three increasing levels of isolation (from simple separation of the two worlds to mutual isolation between service partitions using the secure MPU). Boot typically relies on ROM or an MCUboot-style BL2, with signature verification and anti-rollback counters. The PSA Certified framework assigns assurance levels evaluated by third-party laboratories [15]. The most common level (PSA Certified Level 2) evaluates the target against "scalable" software attacks, explicitly not physical attacks.

This architecture makes good use of the mechanism: the secure TCB remains small (tens of thousands of lines), there are few, well-typed entry points, and the application can be compromised without compromising identity keys or the ability to update — precisely the recovery scenario sought for a connected device. Departing from this model — multiplying secure services or running protocol stacks in the secure world — reproduces the TCB inflation seen in Cortex-A TEEs, without the memory protections of an MMU.

## 7.5 Development and porting pitfalls

Because the TrustZone-M boundary is a function-call boundary, its robustness depends on development details that an assessment of secure firmware must explicitly check. The first group concerns entry points: every exposed function (with the `cmse_nonsecure_entry` attribute) must check that received pointers refer to memory accessible to the non-secure world with the caller's permissions — the purpose of CMSE address-checking routines based on the `TT` instruction — and copy data before validating it, since the caller can modify shared memory during processing. The second group concerns exits: calls from the secure world to non-secure code (function pointers marked `cmse_nonsecure_call`, callbacks) must use compiler-generated sequences that mask registers and use `BLXNS`. Any hand-built direct call is suspicious. Callback targets must be treated as hostile values: the Ret2NS class (section 7.6) specifically penalizes missing checks at this point.

The third group concerns configuration: compile all secure-world code with the CMSE option (`-mcmse`) and applicable errata workarounds; seal secure stacks; lock the SAU and controllers after initialization; install a `SecureFault` handler that logs and handles violations instead of hiding them; and prevent the non-secure world from modifying resources on which the secure world depends (clocks for secure peripherals, power to SRAM banks). Arm documents these points, and TF-M implements them [4, 37]. Verifying them in a custom secure world is an essential review step. Most can be checked with tools against the binary (veneer list, compiler options, sealing values).

## 7.6 Limitations and attack lessons

TrustZone-M is younger than its predecessor, but there is already substantial published experience, revealing three classes of weaknesses: flaws in the transition architecture itself, microarchitectural

side channels that remain despite simple cores, and platform implementation flaws.

### 7.6.1 Boundary flaws: stack sealing and VLLDM

Two Arm security advisories illustrate how sensitive transitions are. The first (CVE-2020-16273) concerns secure stack sealing: because the secure stack pointer can, in some sequences, be influenced by values from the non-secure world, secure software that does not place a seal value at the top of its stacks may be made to pop attacker-controlled data [6]. The countermeasure — sealing every secure stack with a reserved value — is software-based, documented, and included in reference implementations. Its absence from custom secure firmware is exploitable. The second (CVE-2021-35465) affects lazy floating- point context saving: on some Cortex-M33, M35P, and M55 revisions, a sequence involving the `VLLDM` instruction allowed the non-secure world to read floating-point registers used by the secure world [13]. The fix is by silicon revision, with a compiler workaround: the assessor must check the core revision and secure-firmware compiler options. Both cases teach the same lesson: the hardware boundary, implemented "in a few cycles," relies on fine-grained conventions that software and its toolchain must respect.

Academic research has generalized the point: the Ret2NS study showed that insufficiently validated returns from the secure to the non-secure world (`BXNS`, `BLXNS`) allow non-secure control flow to be redirected to chosen targets — in particular, to elevate privileges in the non-secure world by making the return land in a privileged context — and proposed low-cost address sanitization [23]. Here too, hardware checks that the transition is legal, not that it is semantically correct.

### 7.6.2 Microarchitectural side channels without caches: BUSted

The argument that microcontrollers without caches or speculative execution are immune to microarchitectural side channels no longer holds. The BUSted attack (IEEE S&P 2024) demonstrated a timing channel that exploits contention on the microcontroller AHB interconnect. It can be measured from the non-secure world and used to infer secure-world access patterns on real TrustZone-M targets [31]. The channel's bandwidth is lower than a cache channel's, but the assessment conclusion is the same as in chapter 6: secure code handling secrets must be constant-time, including on Cortex-M, and this must be part of secure-firmware review.

### 7.6.3 Platform flaws: the LPC55S69 case

The NXP LPC55S69 microcontroller (Cortex-M33, TrustZone-M) provided a textbook case of implementation flaws. In 2021, Oxide Computer's analysis of its ROM revealed an undocumented hardware block for ROM patching (*ROM patcher*), accessible from the non-secure, unprivileged world, that could change the behavior of ROM APIs at run time — including APIs used for secure boot and TrustZone configuration (CVE-2021-31532) [1]. No weakness in the Armv8-M architecture was involved: the SoC's secure perimeter omitted a critical slave, which was itself undocumented. This case brings together several lessons from this guide: effective security is determined by integration (section 5.1); undocumented blocks are part of the attack surface; opacity in proprietary ROM prevents auditing; and a ROM vulnerability cannot be fixed on silicon already deployed.

### 7.6.4 Physical attacks

TrustZone-M microcontrollers are deployed in physically accessible devices, where attackers can use increasingly affordable fault-injection tools (voltage or clock glitches, electromagnetic pulses).

The isolation configuration — SAU, MPC, and PPC registers, and secure-boot checks — is a natural target: causing a comparison or configuration write to fail is enough in some scenarios to weaken partitioning. The literature and industrial assessments regularly document secure boot bypasses using fault injection on microcontrollers across architectures. Resistance is component-specific: glitch detectors, redundant critical checks in ROM, multiple reads of configuration fuses. This information is rarely public. Certifications targeting hardware attacks (PSA Certified Level 3, high AVA_VAN-level evaluations) may therefore be the only available indicator, and their absence should be interpreted as no guarantee. TrustZone-M, like TrustZone-A, was not designed to protect against a physical attacker.

### 7.6.5 Case study: assessing a TrustZone-M microcontroller product

To make the method concrete, consider a typical product: a connected device built on a Cortex-M33 microcontroller. Its secure world (TF-M) holds an identity key and provides signed updates, while the non-secure world runs the application and network stack. The threat model includes a remote attacker able to compromise the application and a local attacker without laboratory equipment. The assessment then follows these questions, in order:

1. **Root of trust.** Are the ROM and boot configuration documented and free of known CVEs (the LPC55S69 precedent justifies the question)? Are public keys and anti-rollback counters held in fuses and locked in production?

2. **Partitioning.** Are the SAU/IDAU memory map and MPC/PPC configuration consistent, locked after initialization, and do violations raise a handled secure interrupt?

3. **Masters.** For each DMA engine and coprocessor: what attribute does it emit, who can program it, and what can it reach? The answer must be documented, not assumed.

4. **Call boundary.** Do secure entry functions validate their pointers (CMSE routines), are secure stacks sealed, and do the core revision and compiler options cover Arm CVEs?

5. **Secrets.** Are cryptographic operations in the secure world constant-time (BUSted makes this relevant even without caches)? Are keys hardware-bound?

6. **Debug and lifecycle.** Is debug disabled or authenticated in production, for both worlds? Does re-enabling it destroy secrets?

7. **Physical resistance.** If the threat model includes a well-equipped attacker, what fault-injection countermeasures does the component document, and what certification attests to them? Otherwise, high-value secrets should be moved to a secure element.

If questions 1 through 6 receive positive, verifiable answers, TrustZone-M properly supports the scenario "application compromised, identity and updates preserved," its characteristic use case. Question 7 marks what it cannot provide on its own.

# 8

# Conclusion

## 8.1 What TrustZone guarantees, and under what conditions

At the end of this analysis, the scope of TrustZone can be stated precisely. At marginal hardware cost, the technology provides access-control isolation between two complete software stacks sharing the same cores: a compromised normal world, including its kernel, cannot read or modify secure-world resources *provided* four conditions are met. First, a secure boot chain rooted in sound ROM and fuses brings the platform to a locked isolation configuration (sections 3.3 and 5.7). Second, SoC integration is complete: every memory, peripheral, and bus master is covered by a filter whose configuration is known and locked, and programmable masters are constrained by SMMUs, firewalls, or domain controllers (chapter 5). Third, secure-world software is minimal, validates its boundaries, and runs in constant time when handling secrets, since it concentrates most of the demonstrated residual risk (chapter 6). Fourth, the threat model excludes — or covers by other means — a well-equipped microarchitectural attacker and a physical attacker, against whom TrustZone has never claimed to protect.

None of these conditions is exotic, but none is automatic: each corresponds to published failures on commercial products, sometimes prominent ones. That is why this guide exists. The useful question is not "does the platform have TrustZone?" but "does the platform meet the conditions under which TrustZone delivers on its promises?"

## 8.2 Sufficient, needs supplementation, or insufficient

The final decision depends on the threat model and the value of the assets. Three typical situations frame the trade-off.

TrustZone (or TrustZone-M) is **sufficient** when the dominant attacker is remote and software-based, the assets to protect are device identity, update capability, and keys whose compromise has bounded impact, and the platform meets the four conditions above. Recovery — a compromised application, an intact root of execution trust, and a fix deployed through a signed update — is

the characteristic service provided, on both Cortex-A and Cortex-M.

TrustZone **needs supplementation** when some assets justify protection against side channels or a moderately equipped physical attacker: high-value or long-lived keys, fleet-wide shared secrets, or certification requirements. The supplement is a secure element or HSE for operations on those keys, with TrustZone retaining its role in execution isolation and orchestration. This hybrid architecture is now state of the art for demanding products and is developed in the H2Lab guide to selecting security components.

TrustZone alone is **insufficient** — and its presence should not influence the decision — against a well-equipped physical attacker with time and access to the component, when disclosure of secrets would be critical across a fleet, or on a platform whose integration cannot be documented or verified. In the latter case, the problem is not the technology but the impossibility of assessing it: from an assurance perspective, unverifiable isolation is no better than no isolation.

## 8.3 Assessment checklist

**TrustZone platform assessment checklist**



1. Boot chain: ROM with no known CVEs, signatures verified at every stage, anti-rollback in fuses, controlled reset window.
2. Documentation: SoC security map available and sufficient to trace the protection of every resource.
3. Memory firewalls: all access ports covered, suitable granularity, locked configuration, violations detected and handled.
4. Bus masters: emitted attribute, protected programming registers, SMMU or per-master firewalls, coprocessor firmware within scope.
5. MPSoC: domain controllers covering all masters; master × resource matrix verified along both dimensions (domain and attribute).
6. Power management: DVFS and core control confined to the secure world (PSCI) or bounded in hardware.
7. Debug: secure access disabled or authenticated in production; sideways paths (inter-core debug, trace) included in the analysis.
8. Secure software: minimal, up-to-date TCB; parameter validation at boundaries; constant-time cryptography; anti-downgrade mechanisms.
9. TrustZone-M-specific: SAU/IDAU/MPC/PPC consistency, sealed stacks, core revisions and compiler options that cover published CVEs.
10. Threat-model fit: high-value assets entrusted to a component resistant to physical attacks if the model requires it.

## 8.4 Outlook

Ongoing developments do not invalidate this framework; they extend it. For application processors, Secure EL2 and Armv9's CCA architecture [8] address the concentration of privileges in the secure world by fragmenting it and adding hardware-controlled domains. Platforms that benefit from these mechanisms remain subject to the same integration and boot requirements. For microcontrollers, consolidation around TF-M and PSA Certified is gradually raising the quality baseline for secure worlds, while research — from BUSted to Ret2NS — continues to document the mechanism's blind spots, informing future assessments. One constant will remain: TrustZone is an architectural tool whose value is determined by integration, verified through assessment, and lost when assumptions remain implicit.

# Bibliography


[1] Laura Abbott and Rick Altherr. Exploiting undocumented hardware blocks in the LPC55S69. Oxide Computer Company, https://oxide.computer/blog/exploiting-undocumented-hardware-blocks-in-the-lpc55s69, 2021. CVE-2021-31532.

[2] ARM Limited. *ARM Security Technology: Building a Secure System using TrustZone Technology*. ARM Limited, 2009. PRD29-GENC-009492C.

[3] Arm Limited. *ARM CoreLink TZC-400 TrustZone Address Space Controller: Technical Reference Manual*. Arm Limited, 2013. DDI 0504.

[4] Arm Limited. *TrustZone Technology for the Armv8-M Architecture*. Arm Limited, 2017. Document 100690, version 2.1.

[5] Arm Limited. *Trusted Board Boot Requirements CLIENT (TBBR-CLIENT) Armv8-A*. Arm Limited, 2018. DEN 0006.

[6] Arm Limited. Armv8-M stack sealing vulnerability (CVE-2020-16273). Arm Security Center, avis de sécurité, 2020.

[7] Arm Limited. *AMBA AXI and ACE Protocol Specification*. Arm Limited, 2021. IHI 0022.

[8] Arm Limited. *Arm Confidential Compute Architecture (CCA) Security Model*. Arm Limited, 2021. DEN 0096.

[9] Arm Limited. *Arm Power State Coordination Interface (PSCI)*. Arm Limited, 2021. DEN 0022.

[10] Arm Limited. *Arm System Memory Management Unit Architecture Specification, SMMU architecture version 3*. Arm Limited, 2021. IHI 0070.

[11] Arm Limited. *Armv8-M Architecture Reference Manual*. Arm Limited, 2021. DDI 0553.

[12] Arm Limited. *SMC Calling Convention*. Arm Limited, 2021. DEN 0028.

[13] Arm Limited. VLLDM instruction security vulnerability (CVE-2021-35465). Arm Security Center, avis de sécurité, 2021.

[14] Arm Limited. *Arm Architecture Reference Manual for A-profile architecture*. Arm Limited, 2023. DDI 0487.

[15] Arm Limited and PSA Certified. PSA Certified: Security framework and certification scheme. https://www.psacertified.org/, 2021.






[16] El Mehdi Benhani, Lilian Bossuet, and Alain Aubert. The security of ARM TrustZone in a FPGA-based SoC. *IEEE Transactions on Computers*, 68(8):1238–1248, 2019.

[17] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[18] GlobalPlatform. *TEE System Architecture*. GlobalPlatform, 2022. GPD_SPE_009.

[19] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: Enabling dynamic analysis of real-world TrustZone software using emulation. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[20] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[21] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[22] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[23] Zheyuan Ma, Xi Tan, Lukasz Ziarek, Ning Zhang, Hongxin Hu, and Ziming Zhao. Return-to-non-secure vulnerabilities on ARM Cortex-M TrustZone: Attack and defense. In *Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC)*, 2023.

[24] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. BOOMERANG: Exploiting the semantic gap in trusted execution environments. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[25] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[26] Microchip Technology. *SAM L10/L11 Family Data Sheet*. Microchip Technology, 2020. DS60001513.

[27] Zhenyu Ning and Fengwei Zhang. Understanding the security of ARM debugging features. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[28] Nordic Semiconductor. *nRF9160 Product Specification*. Nordic Semiconductor, 2021.

[29] NXP Semiconductors. *UM11126: LPC55S6x/LPC55S2x/LPC552x User Manual*. NXP Semiconductors, 2021.

[30] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[31] Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto. BUSted!!! microarchitectural side-channel attacks on the MCU bus interconnect. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2024.

[32] Keegan Ryan. Hardware-backed heist: Extracting ECDSA keys from Qualcomm's TrustZone. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[33] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust dies in darkness: Shedding light on Samsung's TrustZone keymaster design. In *Proceedings of the 31st USENIX Security Symposium*, 2022.

[34] STMicroelectronics. *RM0438: STM32L552xx and STM32L562xx Advanced Arm-based 32-bit MCUs Reference Manual*. STMicroelectronics, 2021.

[35] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[36] TrustedFirmware.org. Trusted Firmware-A documentation. https://trustedfirmware-a.readthedocs.io/, 2024.

[37] TrustedFirmware.org. Trusted Firmware-M documentation. https://trustedfirmware-m.readthedocs.io/, 2024.

[38] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[39] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. TruSpy: Cache side-channel information leakage from the secure world on ARM devices. Cryptology ePrint Archive, Report 2016/980, 2016.

# License

This document is distributed under the **Creative Commons CC BY-NC-ND 4.0** license (Attribution – NonCommercial – NoDerivatives).
Full license text: https://creativecommons.org/licenses/by-nc-nd/4.0/deed.en