ReZone: Disarming TrustZone with TEE Privilege Reduction

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

[†]*Centro ALGORITMI, Universidade do Minho,* [‡]*INESC-ID / Instituto Superior Técnico, Universidade de Lisboa*

{david.cerdeira, jose.martins, sandro.pinto}@dei.uminho.pt, nuno.m.santos@tecnico.ulisboa.pt

Abstract

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

1 Introduction

Arm TrustZone [42, 45] is a technology embedded into Arm processors shipped in billions of mobile phones and embedded devices. Vendors and Original Equipment Manufacturers (OEM) rely on TrustZone for deploying Trusted Execution Environments (TEEs), which protect the execution of sensitive programs named Trusted Applications (TAs). Some TAs implement kernel-level services of the operating system (OS), e.g., for user authentication or file disk encryption [3]. Other TAs provide shared user-level functionality, e.g., DRM media decoders [37] or online banking services [57]. TEEs themselves consist of a trusted software stack offering API and runtime support for hosting TAs. TEEs such as Qualcomm's QSEE and OP-TEE protect the confidentiality and integrity of TAs' memory state, thereby ensuring it cannot be inspected or tampered with by a potentially compromised OS.

To protect the TAs, TEEs leverage mechanisms provided by TrustZone. In Armv7-A/Armv8-A, this technology provides two execution environments named *normal world* and *secure world*[1]. The normal world runs a rich software stack, named Rich Execution Environment (REE), comprised of full-blown OS and applications; it is generally considered untrusted. The secure world runs a smaller software stack consisting of TEE and TAs. TrustZone enforces system-wide isolation between worlds and provides controlled entry points for world switching whenever the REE invokes TAs' services.

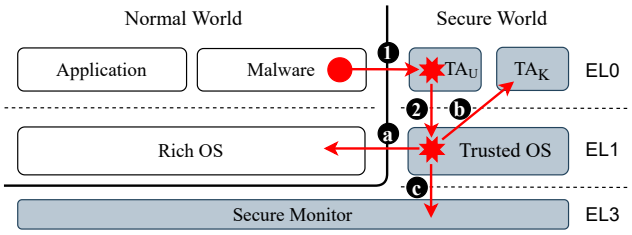

Figure 1: Privilege escalation attack in a TrustZone-assisted TEE: (1) hijack a user-level TA, e.g., exploiting vulnerability in the Widevine TA (CVE-2015-6639) [19], (2) hijack the trusted OS, e.g., exploiting vulnerability in syscall interface of Qualcomm's QSEOS (CVE-2016-2431) [20]. Once in control of the trusted OS, other attacks can be launched: (a) control the rich OS, e.g., backdoor in the Linux kernel [22], (b) control a kernel-level TA, e.g., to extract full disk encryption keys of Android's KeyMaster service [18], or (c) control the secure monitor, e.g., to unbrick the device bootloader [21].

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

[1] As per Arm's recent documentation [6], *secure world* can be referred to as *trusted world*; we adopt the previous and more familiar terminology [45].

control of the system, including the REE in the normal world (a), all the TAs (b), and the secure monitor (c).

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

Unfortunately, numerous devices face serious risks of being compromised by attacks of this nature. In the past, security researchers have demonstrated how these attacks can be mounted by targeting the trusted OS through a chain of vulnerability exploits as described in the caption of Figure 1. Notably, recent studies [11] have shown that commercial TEE systems have plenty and serious vulnerabilities, most of them affecting user-level TAs and trusted OS. This suggests that the current attack surface for TEE systems is seriously enlarged.

Despite the abundant research on TrustZone security [11, 45], there is a lack of practical system defenses if the trusted OS is hijacked. TEE systems like Sanctuary [10] aim to reduce the attack surface of the trusted OS by relocating TAs onto user-level enclaves in the normal world. However, due to portability reasons, vendors and OEMs continue deploying TAs in commercial TEEs, therefore exposing many devices to potential attacks. Another approach is to confine the TEE stack inside a secure world sandbox, e.g., by leveraging same privilege isolation [32, 35] or hardware virtualization [56]. However, these approaches do not solve the fundamental problem of the excess of privileges granted to S.EL1 (or S.EL2).

This paper presents ReZone, a new security architecture that can effectively counter ongoing privilege escalation attacks by reducing the privileges of a potentially compromised trusted OS. We build our solution specifically for restricting the privileges of S.EL1 on Armv8-A platforms featuring the protection modes displayed in Figure 1, and then discuss how our techniques can be employed for reducing the excess of S.EL2 privileges in Armv8.4-A and ensuing architectures. With ReZone, a typical monolithic TEE can be restructured and partitioned into one or multiple *zones*, which consist of sandboxed domains inside the secure world. Zones have access only to private memory regions, and provide an execution environment for untrusted $S.EL_{1/0}$ code, i.e., trusted OS and TAs. ReZone restricts the memory access privileges of the code running inside a zone, preventing it from arbitrarily accessing memory allocated for: (a) the normal world REE,

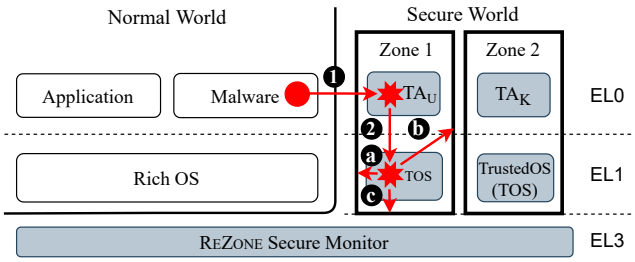

Figure 2: TEE privilege reduction with ReZone: an attacker cannot escalate privileges from the compromised trusted OS running in zone 1 to (a) the rich OS, (b) the kernel-level TA_K in zone 2, or (c) the secure monitor.

(b) other zones, and (c) the secure monitor. Figure 2 depicts a setup featuring two zones – 1 and 2 – each of them runs a trusted OS instance as in a library OS hosting local user- or kernel-level TAs. An attacker hijacking the trusted OS in zone 1 (as explained in Figure 1), will not be able to further escalate its privileges beyond its sandbox. Additional zones can be created, potentially hosting different trusted OS software.

A central novelty of ReZone's design is that we leverage TrustZone-agnostic hardware primitives available in COTS SoCs to restrict the privileges of S.EL1 (trusted OS) code. Existing systems such as Sanctuary [10] have already leveraged the TrustZone Address Space Controller (TZASC) for restricting access permissions to various physical memory regions in the normal world. However, S.EL1 code can still revert these permissions, rendering the TZASC insufficient for zone isolation. Instead, we use a platform resource controller such as the Resource Domain Controller (RDC) existing in i.MX8MQ and i.MX8QM SoCs, which allows specific bus masters to control memory access permissions. In ReZone, we harness this feature along with a low-end microcontroller also available in the SoC to control these permissions, effectively creating private memory regions for each zone. RDC and microcontroller can then be used as building blocks for restricting the memory access privileges of a (potentially compromised) trusted OS, and enforcing strong zone isolation.

Based on this insight, we have built ReZone. Our system includes multiple non-trivial optimizations and fine-grained mechanisms to guarantee system-wide security (e.g., avoiding cache-based attacks based on cached content available for EL3 and S.EL1 code). To make our design generally applicable to various hardware platforms, we specify the general properties of SoCs that satisfy ReZone's requirements. We have fully implemented a ReZone prototype for the i.MX 8MQuad EVK, and integrated it with Android OS as REE and with OP-TEE as TEE. Our code is open sourced [15].

We evaluated ReZone using microbenchmarks and real-world applications. Our evaluation shows that despite the existence of non-negligible performance overhead in microbenchmark programs, ReZone does not sensibly degrade the performance of applications such as crypto wallets or DRM

media decoders. The DRM application, which requires frequent transitions between the normal world and the zone, preserves its ability to replay video at high frame rates. We have surveyed 80 CVE vulnerability reports and estimate that REZONE can thwart 86.84% of potential privilege escalation attacks arising from exploiting these bugs.

2 Background and Motivation

2.1 Standard TrustZone Mechanisms

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

TrustZone extends the memory system by means of an additional bit called *NS bit* that accompanies the address of memory and peripherals (see Figure 3). This bit creates independent physical address spaces for normal world and secure world. Software running in the normal world can only make *non-secure accesses* to physical memory because the core always tags the bus NS bit to 1 in any memory transaction generated by the normal world. Software running in the secure world usually makes only *secure accesses* (i.e., NS=0), but can also make non-secure accesses (NS=1) for specific memory mappings using the NS flags in its page table entries. The processor maintains separate virtual address spaces for the secure and non-secure states. Trusted OS (or the secure monitor) can map virtual addresses to non-secure physical addresses by setting to 1 the NS bit in its page table entries.

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

2.2 The Excess of Privileges of the Trusted OS

The TrustZone mechanisms described above were designed under the assumption that the S.EL1 (trusted OS) and EL3 (secure monitor) are trusted. However, if the trusted OS is hijacked, the attacks described in Figure 1 can be trivially mounted. For illustration purposes, suppose that the physical memory addresses PA_{OS}, PA_{TK}, and PA_{SM} are allocated to the rich OS, TA_K, and secure monitor, respectively. The attacker

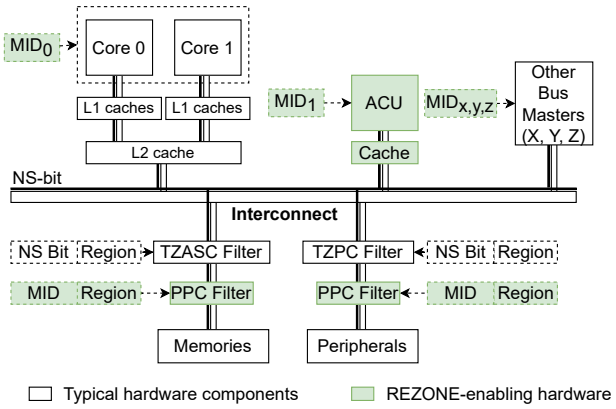

Figure 3: Hardware platform featuring TrustZone: REZONE-enabling hardware components are highlighted, i.e., ACU and PPC. PPC is configured by the bus master MID_1 (ACU). These configurations determine which access permissions are granted to a given MID for a specific physical memory region.

could mount said attacks by creating virtual address mappings in the S.EL1 page tables as indicated below, and use the virtual addresses VA_i to access (read/write) the memory pages of the rich OS, TA_K, or secure monitor. Notation $VA \rightarrow \langle PA, NS \rangle$ denotes a page table entry translating virtual address VA to the physical address PA and associated NS bit value:
$VA_{OS} \rightarrow \langle PA_{OS}, 1 \rangle$, $VA_{TK} \rightarrow \langle PA_{TK}, 0 \rangle$, $VA_{SM} \rightarrow \langle PA_{SM}, 0 \rangle$

Simply put, *the secure monitor cannot prevent the trusted OS from mapping these memory regions arbitrarily into the S.EL1 address space*. Given that the trusted OS does not require this level of control to sustain its typical operations, this means that S.EL1 is endowed with excessive privileges that can be abused. If the trusted OS is compromised, the attacker can leverage these privileges to take over every part of the system, including the monitor, TZASC, and normal-world components. For this reason, we argue that reducing the memory access privileges of S.EL1 code can significantly lower the impact of vulnerability exploits targeting the trusted OS. In this paper, we concentrate primarily on devising a solution to this problem. Then, in §8.2 and §8.3, we explain that Arm's architecture releases starting from Armv8.4 did not entirely solve the problem of over-privileged protection modes in the secure world, and discuss how our techniques can potentially be applied to these more recent architectures.

2.3 Platform Model for Restricting S.EL1

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

Vendor	SoC Platform	PPC	ACU	RZ?	Sources
NXP	iMX8MQ	RDC	Cortex-M4	Yes	Ref. manual (https://www.nxp.com/webapp/Download?colCode=IMX8MDQLQRM)
	iMX8QM	xRDC	SCU	Yes	Ref. manual (https://www.nxp.com/webapp/Download?colCode=IMX8QMRM)
Xilinx	UltraScale+ MPSoC	XMPU, XPPU	PMU	Yes	Ref. manual (https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf)
Nvidia	Tegra X1 and X2, Xavier	SMMU	BPMP	Yes	Ref. manual (https://developer.nvidia.com/embedded/downloads)
Socionext	SC2A11	SMMU	SCP	Yes	TF-A V2.6 (https://github.com/ARM-software/arm-trusted-firmware) and U-boot v2022.01 (https://github.com/u-boot/u-boot)
Qualcomm	Snapdragon 845, 855, 865, 888, 8gen1	SMMU, XPU*	SPU	Yes	Linux 5.16 for SD855 (https://gitlab.com/sm8150-mainline/linux), CVE-2020-11252, CVE-2017-18311, [48]. *We lack the data to support that 845 and 8gen1 contain an XPU, though earlier platforms of the same segment feature one.
Broadcom	Stingray	SMMU	SCP	Yes	TF-A (Same as Socionext), Linux 5.16 (https://github.com/torvalds/linux)
Samsung	Exynos 990	Periph. MMUs	iSE	N/A	S20 and S21 Linux kernel source code (https://opensource.samsung.com/main)
	Exynos 2100	Periph. MMUs	eSE	N/A	
Mediatek	Dimensity 1200	✗	✗	No	OnePlus Nord 2 Linux kernel source (https://github.com/OnePlusOSS/android_kernel_oneplus_mt6893.git)
HiSilicon	Kirin 9000	✗	PMU	No	Huawei Mate 40 Pro Linux (https://consumer.huawei.com/en/opensource/)

Table 1: Availability of PPC/ACU hardware primitives and applicability of REZONE on COTS platforms.

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

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

1. Platform Partition Controller (PPC). Consists of a memory and peripheral protection controller that can restrict physical accesses to a given resource, i.e., memory or peripheral (see Figure 3). On the i.MX8MQ SoC, the PPC is implemented by the RDC peripheral. We characterize the PPC by having four main properties. It must be capable of: (a) intercepting all physical requests regardless of both the NS bit signal injected into the system bus and the configurations of the TZASC, (b) setting fine-grained access control attributes (e.g., RO, RW) for configurable memory regions; (c) differentiating between multiple bus masters, and (d) protecting memory-mapped input/output (MMIO) regions, including the PPC's memory-mapped registers. Importantly, (c) and (d) help to ensure that the PPC's configurations cannot be arbitrarily changed by (untrusted) S.EL1 code executed by the processor. Intuitively, to provide this guarantee in the setup illustrated in Figure 3, the PPC would not allow the bus master identified with MID_0 – i.e., the processor cluster – to arbitrarily change the PPC configurations. Instead, as explained next, it is the bus master identified with MID_1 that will have that privilege.

2. Auxiliary Control Unit (ACU). ACUs are small microcontrollers like Cortex-M4 added into the SoC for housekeeping purposes. For example, Figure 3, the ACU is identified by MID_1. The ACU coordinates the configuration of memory access permissions in the PPC, ensuring that the main processor cores cannot tamper with the PPC permissions.

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

2.4 Hardware Support on COTS Platforms

The primitives presented above are readily available on i.MX8MQ and i.MX8QM SoCs, the first of which we use for building REZONE. In i.MX8MQ, PPC and ACU are implemented, respectively, by RDC and Cortex-M4 hardware. To further assess if REZONE can be applied at a large scale, although secure boot is already commonly available, we need to determine if COTS platforms include hardware components that can play the role of PPC and ACU. To this end, we analyzed 17 popular SoCs from nine different vendors. We based our study on several sources. In some cases, we consulted publicly available SoC reference manuals. In others, we inspected the source code of the Linux kernel, bootloader, and secure monitor looking for PPC/ACU driver support. CVEs were also useful for identifying a PPC-like mechanism on Qualcomm SoCs. Our main findings are presented in Table 1.

PPC hardware candidates. Apart from the RDC, we identified several other components that can be used as PPC: SMMU, XPU, and XMPU in conjunction with XPPU. The Arm SMMU is an IOMMU architecture that can translate an input address to an output address based on address mapping and memory attributes (i.e., translation tables). The main requirement for an IOMMU to be used as a PPC is the possibility of performing address translation functions for multiple

masters, in particular the CPU. The Arm SMMU architecture, in its different versions, i.e., Arm SMMUv1, SMMUv2, SMMUv3, can fully provide such a feature. All of our studied platforms by Nvidia, Qualcomm, and Broadcom include a full-fledged SMMU. However, some custom SMMU implementations are designed to only interpose accesses of specific bus masters, e.g., media processing hardware. Such is the case of Mediatek 1200 and Kirin 9000. Samsung Exynos 900/2100 include dedicated SMMUs for specific peripherals. However, due to the lack of available information, we cannot ascertain that these SMMUs can also filter CPU accesses. The XPU [48] is a Qualcomm hardware component that entirely fills the PPC requirements. The same is true for Xilinx's XMPU and XPPU, which restrict accesses to memory and peripherals respectively [53]. In general, we observe that recent platforms are more likely to feature PPC-like mechanisms[2].

ACU hardware candidates. On the analyzed SoCs, we identified various ACU candidates. Qualcomm, Samsung, and NXP's i.MX8QM include security co-processors that can be used for this purpose: Secure Processing Unit (SPU) [47], integrated/embedded Secure Element (iSE and eSE), and System Controller Unit (SCU). All other surveyed SoCs incorporate programmable microcontrollers usually reserved for platform management purposes, e.g., power management. With one exception, these microcontrollers can be used as a full-fledged ACU: Mediatek's SoC has a System Control Processor (SCP), but the SCP interface is based on MMIO registers and not a mailbox interface as the other analyzed ACUs. Therefore, this SoC is likely too restrictive for ReZone. In summary, our findings are consistent with Arm's reports [52], which point to a clear trend in COTS SoCs to include companion processing units for power management and security operations.

Platform support for ReZone. In our study, we found that 13 SoCs have suitable PPC and ACU hardware, which make the deployment of ReZone possible on devices featuring these platforms. In contrast, the SoCs from Mediatek and HiSilicon lack at least one of the required PPC or ACU mechanisms to accommodate ReZone. In the case of Samsung's SoCs, we miss important information to issue a final judgment. In general, our study shows that our defense mechanisms can further be ported to platforms beyond the i.MX8 SoC family, which we use for our implementation. All NXP's i.MX8, Xilinx's Zynq, and Qualcomm's Snapdragon SoCs deployments span a wide range of edge computing settings (e.g. automotive) [46]. It is interesting to note that most of the analyzed Qualcomm platforms are compatible with ReZone, which means that our system can potentially be deployed on many commodity mobile devices.

[2] At a first glance, it may seem that TZASC fits all the PPC requirements enumerated in §2.3. However, a TZASC only protects memory regions. Therefore, it misses the PPC requirement (d). Additionally, only TZASC-400 can differentiate between bus masters, a highly specific hardware component not widely deployed. In fact, looking at the latest TF-A release [4] only two platforms support TZASC-400: NXP's lx2160a and ls1028a. To control access to peripherals, the TZPC is also needed. However, available documentation [5] shows that the TZPC cannot distinguish between multiple bus masters or prevent accesses to itself, missing PPC requirements (c) and (d).

3 Design Goals and Threat Model

We aim to design a security architecture that leverages the primitives presented above to create secure world sandboxes. We refer to these sandboxes as *secure world zones*, or simply *zones*. A zone is meant to host a partitioned TEE stack running at $S.EL_{1/0}$, i.e., trusted OS and TA software (see Figure 2). Since multiple zones z_i can co-exist within the secure world, we use the superscript notation $S.EL^i_{1/1}$ to identify zone i. More concretely, we strive to attain four subgoals:

1. Reduce the privileges of the trusted OS. As depicted in Figure 2, zones must thwart the escalation of privileges of an attacker from the trusted OS to other memory regions of the system. To this end, a zone z_i is expected to enforce three security properties: (P1) protection of the normal world, i.e., software running at $NS.EL_{1/0}$, (P2) protection of the secure-monitor, i.e., EL3 code, and (P3) protection of co-located zones, i.e., software running at $S.EL^j_{1/0}$, where $i \neq j$.

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

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

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

Threat model. As for the threat model, the attacker's main goal is to subvert the security properties of zones (i.e., P1-3 shown above in subgoal 1). We assume the adversary has already compromised the trusted OS of a given zone z_i, and aims to escape it (see Figure 2). Running in $S.EL1^i$, the attacker may attempt to access (read/write) external memory addresses outside the z_i's private memory (§2.2). These access requests may target i) the physical address space of the victim at $NS.EL_{1/0}$, EL3, or $S.EL^j_{1/0}$, or ii) the PPC or memory regions used by the ACU. In the first case, the objective is to override the PPC's memory-mapped registers containing the permissions that forbid access to the victim's memory regions. In the second case, the idea is to subvert the ACU's behavior, e.g., causing the ACU to hand over the control of the PPC to the processor core executing the attacker's code. This

would indirectly allow the attacker to disable the PPC's restrictions. The attacker may also leverage the effect of caches to read or write cached memory cells for which S.EL1^i does not currently have permissions.

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

4 Design

Figure 4 represents the architecture of REZONE. Hardware-wise, our system relies on a typical TrustZone-enabled platform. For controlling memory access permissions, in addition to a TZASC controller, REZONE relies on a PPC hardware component. The PPC is dynamically configured to block secure world accesses from the processor based on the processor's bus master ID (MID_0). The PPC can be reconfigured only by a bus master, predefined at bootstrapping time. In REZONE, this bus master is the ACU (MID_1). ACU and processor can communicate with each other efficiently using a message queue (MQ) implemented by a hardware peripheral.

Software-wise, REZONE comprises the *secure monitor* and the *gatekeeper*. The former consists of standard secure monitor software (e.g., implemented by Arm Trusted Firmware) augmented with a REZONE-specific sub-component named *trampoline*. The secure monitor (and trampoline) run on the main processor core and the gatekeeper on the ACU; the PPC protects their private memory regions, which store security-sensitive context information. Taken together, trampoline and gatekeeper manage the execution of zones in the system. They ensure that each zone can access only a private physical memory address space assigned to the zone, and take care of all context-switching tasks involving zone entering and exiting operations. These operations occur when an REE application makes a call to a zone's guest TA (*zone entry*), and the TA returns the results of the call (*zone exit*). REE and zone can share data through a shared memory region. REZONE's software components are shipped with the platform firmware. When the system bootstraps, the firmware configures the memory layout and statically creates one or multiple zones indicating the composition of their respective software stacks, i.e., trusted OS and TAs. Next, we explain our design decisions and how the system works. We refer the reader to Figure 5.

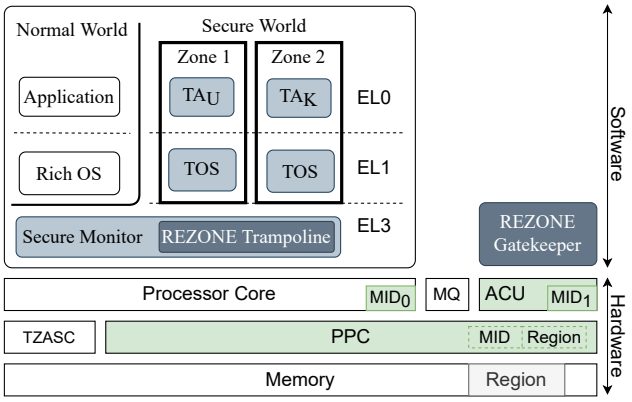

Figure 4: REZONE architecture: Its software components are colored in dark shade and consist of secure monitor (which includes the trampoline) and gatekeeper. Secure monitor runs on the main processor cores; gatekeeper runs on the ACU. REZONE software controls memory accesses via the PPC.

4.1 Memory Partitioning and Permissions

The starting point in REZONE is to partition the physical memory layout into regions, allocating these regions to different protection domains of the system, and specifying regions' memory access permissions accordingly. In doing this, we are primarily concerned about restricting secure world accesses when the processor runs zone code, i.e., S.EL$_{1/0}$. The final memory layout and access permissions are depicted in Figure 5. We progressively explain these regions and permissions, beginning with a vanilla TrustZone-enabled platform, and then extending it with REZONE's protections.

Just like in a typical TrustZone setup, in REZONE, the normal world includes regions for the REE OS (M_{REE}) and shared memory (M_{SH}). The secure world has independent regions for the TEE software. The TZASC is then configured to prevent the normal world software from accessing secure world regions. The first line in the permissions table (see Figure 5) shows how the normal world has only read/write access permissions to normal world memory regions.

Then, we further separate the secure world memory into independent regions for the secure monitor and for each zone *i*, e.g., zone 1 (M_{Z1}). We need to ensure that the secure monitor software (EL3) has full access permissions, but the code running at S.EL$_{1/0}$ is confined to (i) the zone's private memory M_{Z1} and (ii) the REE shared memory for TA calls. To this end, we leverage the PPC to restrict secure world accesses. However, using the PPC for this purpose is not trivial. Given that TrustZone does not mark memory accesses with the privilege level from which they originate (being tagged per world only) the PPC cannot distinguish secure accesses coming from EL3 or from S.EL$_{1/0}$. As a result, PPC's permissions need to be dynamically changed by the monitor when entering a zone (i.e., reprogramming the PPC to remove RW access permissions to M_{REE} and to secure world memory as per line three in

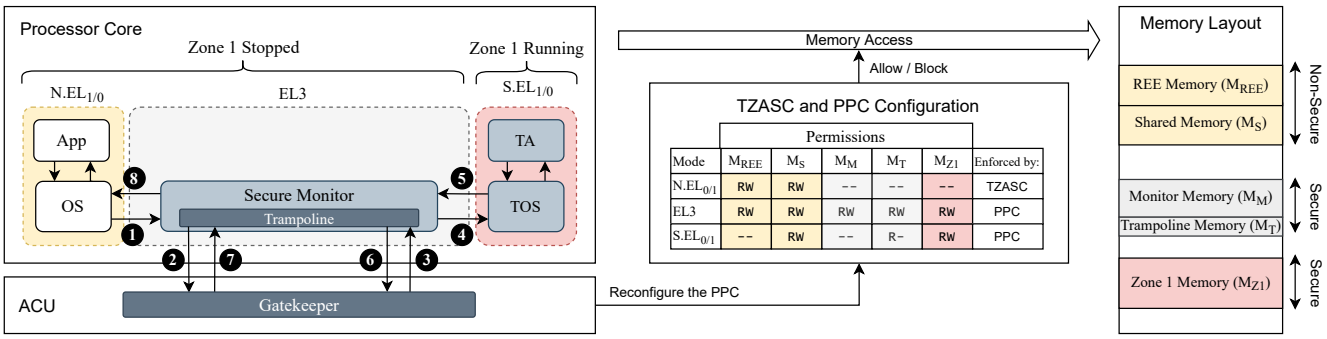

Figure 5: REZONE execution model, memory access permissions, and memory layout: REZONE ensures that the code running inside zone 1 – i.e., at $S.EL_{1/0}$ – can only access zone 1's private memory region (M_{Z1}). This protection is achieved by reconfiguring the PPC to restrict the processor's access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

the permissions table) and when leaving a zone (i.e., reverting to the permissions in line two of the table).

However, a difficulty arises when exiting a zone. If the S.EL1 code running inside a zone is wholly prevented from accessing the secure monitor memory, this protection will still be in place by the time the zone's trusted OS traps into EL3; consequently, the secure monitor will be blocked by PCC resulting in memory access aborts and a system crash. To solve this problem, we create a dedicated secure memory region containing code and data that are necessary only to facilitate this transition securely. This code is part of the trampoline, which still belongs to the secure monitor because it runs in EL3. We then segregate a region of the trampoline's memory (M_T) from the rest of the secure monitor memory (M_M) because these regions need to be configured with different memory access permissions. M_M is always denied access from $S.EL_{1/0}$. In contrast, M_T is marked read-only by the PPC, therefore, allowing the processor to both execute trampoline code from M_T and load context data also from M_T while still running in S.EL1. The trampoline code will then engage the ACU to remove the PPC memory access restrictions to M_M, thus allowing the secure monitor to freely run at S.EL3 and restore the normal world execution. These reasons justify the PPC permissions in the last two rows of the table in Figure 5.

4.2 Securing the PPC Memory Permissions

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

REZONE also needs to defend against a potential attack. Processor and ACU communicate via a message queue unit (MQ in Figure 4) which is agnostic to the current exception level of the processor. As a result, an attacker running at S.EL1 in a zone could craft a request to disable the memory access restrictions enforced by the PPC. To thwart this threat, the gatekeeper must foretell that this request is malicious and therefore refuse it. To this end, trampoline in EL3 and gatekeeper share a secret token that is initialized at boot time and copied to their private address space. This token is then used by the trampoline for authenticating PPC reconfiguration requests sent to the gatekeeper. Since the secret token is not accessible at S.EL1, an attacker can no longer spoof legitimate requests, making the system robust against these attacks.

4.3 Preventing Cross-zone Interference

The mechanisms discussed so far secure all memory transactions observed outside the processor cluster according to the permissions matrix in Figure 5. These mechanisms require additional measures to safeguard against potential attacks within the processor itself. We address two concrete concerns.

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

Second, the PPC cannot differentiate between individual cores within a cluster, because the PPC observes requests as being from a single bus master: the cluster's shared cache. As a result, REZONE's memory restrictions enforced by the PPC are collectively applied across all cores of the same cluster. However, blocking access to the normal world memory when

a core enters a zone will also prevent other co-located cluster cores from accessing normal world memory, which may lead to crashes if they are currently executing REE code. To prevent this problem, upon zone entry, REZONE must halt other co-located cluster cores running in the normal world until the core exits the zone. Despite the non-negligible performance overheads potentially introduced by these measures, our experiments show that these overheads can be tolerated due to the infrequent TEE invocation patterns and short TA execution times inside the secure world (see §6), essentially trading some performance degradation for increased TEE security.

4.4 Zone Entry and Exit Workflows

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

A zone entry begins when the normal-world OS issues a request through the invocation of the `smc` instruction (1), causing the execution to be handed over to the secure monitor. If the request needs to be handled by the trusted OS, REZONE will first differentiate to which zone the requests to that particular TA have been statically assigned. Then, it forwards the request to the respective zone, i.e., zone 1 in Figure 5. For this, the secure monitor starts by notifying all other cores in that cluster. The other cores will then enter an idle state. The trampoline first flushes the core's and shared cache content and then continues by issuing a request (2) to the gatekeeper to configure the PPC to block access of the application core. To authenticate the request as coming from the secure monitor processor mode, the gatekeeper expects to receive the secret token, which was handed to the monitor during boot. After a successful reply from the gatekeeper (3), the trampoline will then (4) jump to the trusted OS.

A zone exit occurs once the trusted OS returns execution to the secure monitor (5). At this point, the trampoline issues a new request to the gatekeeper (6) to unblock secure memory accesses from the core. The gatekeeper ensures this by reconfiguring the PPC to disable the zone's protection policies, i.e., granting `RW` for all memory regions. The secure monitor will then (7) resume execution, notify other cluster cores to, finally, return to the normal world context (8) executing under traditional TrustZone restrictions enforced by the TZASC.

5 Implementation

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

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

Based on the security domain scheme provided by this PPC, we use the following insight to implement the memory partitioning and access permissions described in §4.1. Since the PPC is itself a platform resource, we need to reserve a domain (e.g., DID_1) with full read/write permissions to the PPC's registers, and associate the ACU as the single bus master to DID_1. Doing that, we guarantee no other bus master in the system will be able to tamper with PPC's security permissions. If DID_1 can be specified on the system prior to any tampering from an adversary, we ensure that the ACU remains in full and exclusive control of the PPC. To achieve this, we can rely on the secure boot to run a customized initialization routine (i.e., gatekeeper code) that will fully initialize the ACU before starting the zones' trusted OSes.

Our implementation configures two security domains when booting the system: DID_0, and DID_1. DID_0 contains all memory (and other) resources exposed to the cluster. DID_1 mainly contains the PPC and the gatekeeper's private memory region. The processor is associated with DID_0, and the ACU to DID_1. As a result, only the ACU can reconfigure the PPC as per the access permission matrix depicted in Figure 5 (last two rows). In the i.MX 8M family, Cortex-A clusters are assigned a unique master ID. This means that DID_0's permissions apply indistinguishably to every Cortex-A53 core of the cluster.

We need to establish a secure channel between the cluster and ACU. Upon zone entry/exit events, they exchange message requests (reconfiguration) via the platform's MQ (see Figure 4). In the i.MX8MQ SoC, the MQ is composed of MU_A and MU_B. The PPC is configured to assign MU_A exclusively to DID_0, and MU_B to DID_1, ensuring that only the processor and ACU can communicate with each other through the local MU interfaces. To authenticate the requests sent by the processor, trampoline and gatekeeper share a secret token. To preserve the secrecy of the token, the trampoline stores it in the `TPIDR_EL3` 64-bit register, which is exclusively available to EL3 code (not even S.EL1 code can read this register) leaving the attacker a 1 in 2^{64} chance of guessing its value correctly. The token is randomly generated at boot time.

Given that the ACU runs at a clock speed 5.6× slower than the cluster, we implemented an optimization to offload workload from the gatekeeper (ACU) to the trampoline (cluster).

We observed that the reconfiguration of the PPC's permissions DDI_0 resources become slower as the number of resources grows. To shift most work to the trampoline, the gatekeeper temporarily associates the PPC to DDI_0, allowing the trampoline to update PPC's permissions on the gatekeeper's behalf. Once finished, gatekeeper removes PPC from DDI_0, reacquiring its former condition of sole guardian of the PPC.

5.1 Cross-core Synchronization

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

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

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

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

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

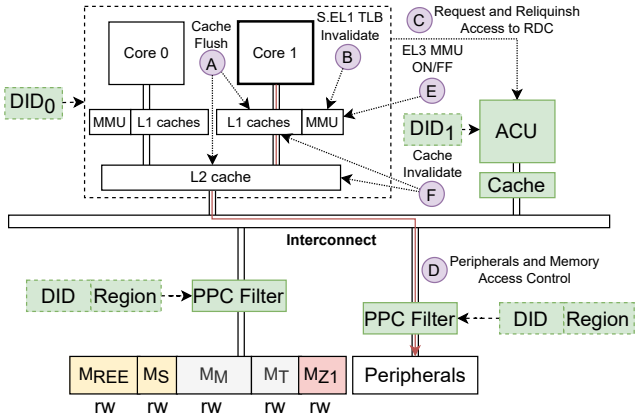

Figure 6: Context-switch between EL3-S.EL1 requires microarchitectural maintenance operations (A, B, E). Entering a zone (EL3→S.EL1) requires A, B, C, D and E. Exiting a zone (S.EL1→EL3) requires C, D, E and F.

5.2 Microarchitectural Maintenance

Triggered by a zone entry/exit event, context switching between EL3-S.EL1 requires the trampoline to coordinate not only cross-core synchronization, but also perform important cleanup tasks (see Figure 6) and PPC reconfiguration operations. We now focus on microarchitectural maintenance that needs to be carried out, and then in §5.3 we describe how the trampoline dynamically reconfigures the PPC.

When entering a zone (EL3→S.EL1), the trampoline running on the core needs to ensure that the trusted OS (after the switch is finished) cannot access resources outside of the zone. After taking all measures to prevent interference from other cores (§5.1), the trampoline needs to invalidate the L1 instruction and data caches and the shared L2 cache (A). Unless this operation is performed, data or instructions left in the cached will be accessible to the trusted OS, thus violating zone isolation. Also, right before jumping into the trusted OS, the trampoline disables SMP coherency. Recall that at this point, all other cores in the cluster are suspended. Each core has data and instruction in its own L1 caches. With SMP coherency enabled, a request to access cached data would cause the coherency protocol to fetch the data from another core's L1, resulting in a cache hit; thus, data could be accessed by the present core, violating the zone's isolation properties.

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

belongs to another zone, thus causing unexpected faults. As an optimization, we perform TLB maintenance only when switching between different trusted OSes, and it does not affect TLB entries of the REE OS. With all microarchitectural maintenance done, the trampoline will restore the general-purpose register state and exit to the trusted OS.

5.3 Dynamic PPC Reconfiguration

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

The reconfiguration of PPC's permissions is executed by the trampoline in EL3 after performing micro-architectural maintenance, as doing so before would prevent normal world and monitor memory from being written back to main memory. To perform the reconfiguration, the trampoline must first obtain access to the PPC. This step is achieved by sending a request to the ACU with the secret token to authenticate the trampoline (step C in Figure 6). The ACU will then allow the application cores to access the PPC. At this point, only the trampoline can access the PPC. The trampoline can then perform the necessary PPC configurations (step D) as per the permissions matrix presented in Figure 5. After the PPC reconfiguration is concluded, the trampoline sends another request to the ACU. This time the ACU configures the PPC to prevent any accesses to the PPC by any bus master other than the ACU itself. Once the ACU replies to the trampoline, control can be safely handled over to the trusted OS.

5.4 Handling Exceptions at S.EL1 Exits

In contrast to zone entries, exiting a zone from S.EL1 to EL3 precludes steps A and B (see Figure 6). Flushing caches (A) is not necessary, since we selectively invalidate monitor memory in step F (explained below), thus preventing malicious cache manipulation to gain code execution privileges. TLB maintenance (B) is not required because S.EL1 TLB entries only affect S.EL1. However, we need special precautions in dealing with exceptions from S.EL1. Two problems may arise:

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

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

To solve both problems, we disable the MMU for EL3 code (E in Figure 6). We do this in the last steps of exiting from EL3 to S.EL1 when entering a zone. Note that this measure only applies to the MMU for EL3 code: we do not disable the MMU for S.EL1 code. This solves problem 1, as, from that point on, memory accesses at EL3 will not be subject to MMU translation; thus, no page table accesses are needed, and the exception handler can then be executed. Disabling the MMU will also disable L1 and L2 caches for EL3 address spaces, and this will partially solve problem 2 as the exception handling code will always be fetched from main memory. This memory is protected read-only by the PPC, hence no modifications will be possible; however, this solution does not entirely solve the whole problem. Note that during the switch from EL3 to S.EL1, after flushing all caches, the trampoline still executes with caches enabled. This will inevitably leave some traces of trampoline data and instructions in the caches, which a malicious trusted OS can try to leverage for an attack by modifying them locally while running in S.EL1 as described above. To clean up potentially tampered cache content and avoid these attacks, we invalidate on-cache trampoline-related memory when the exception handler in EL3 is executed on an S.EL1 exit (F in Figure 6). After this cache management operation completes, the exception can be handled securely.

5.5 Software Implementation

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

We also changed the TF-A's context management routines and data structures to support two zones, each running a

trusted OS instance based on OP-TEE (v3.7.0). To run OP-TEE inside a zone, no intrusive modifications were necessary; we have disabled only the support for dynamic shared memory. We create the second OP-TEE instance in a different address space and updated the values of SMC IDs to be uniquely attributed to each OP-TEE instance. We created two full-blown stack builds featuring different REE: one running the Linux kernel (5.4.24) and the second the Android OS (AOSP 10.0). It was not necessary to modify REE code to support REZONE; however, since we added support to run multiple trusted OSes, we added drivers to interface with both of them. We also duplicated the OP-TEE Linux kernel driver and modified the SMC_IDs used to make calls to the second OP-TEE instance. There are two tee-supplicant services, one for each OP-TEE.

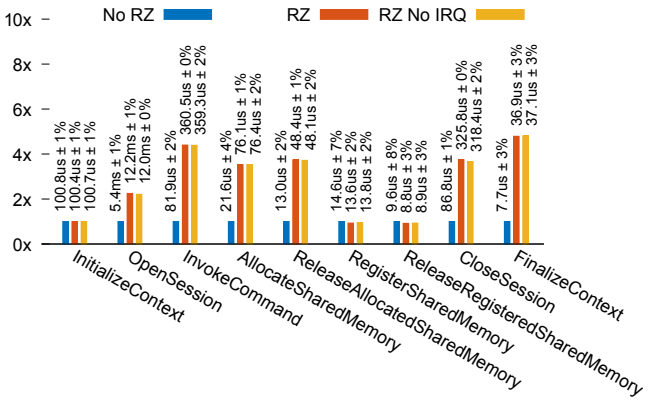

Figure 7: REZONE overhead, time, and normalized standard deviation (coefficient of variance) of GlobalPlatform API.

6 Performance Evaluation

We evaluated REZONE using the i.MX8 platform and software described in §5. On the i.MX8 platform, all cores, i.e., the Cortex-A53 APU cluster (4x) and the Cortex-M4, were running. Our performance evaluation covers three vectors:

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

2. Real-world applications. We evaluated two TAs: a Bitcoin wallet and a DRM player service. The open-source Bitcoin wallet TA [29] provides services, e.g., for creation and deletion of a master key, access the mnemonic for a master key, sign a transaction, and get the address of the wallet. The DRM player service is implemented with the open-source OP-TEE Clearkey plugin [36] and the media player ExoPlayer [24]. This service approximates to a security level 2 (L2) DRM compatible setup [41], where video content is split into subsamples and decrypted within the TEE, but processed in the normal world. The L2 DRM setup represents the worst-case scenario for performance since the process is split among the two worlds. We measured the (i) execution time for decrypting one subsample for three different resolutions and frame rates and (ii) the time elapsed between decoded content.

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

Methodology. To measure world switch time, we used the AArch64 generic timer, a 64-bit cycle counter running at 8.33MHz. For the GlobalPlatform API and xtest suite, we used the Linux time API with the CLOCK_REALTIME parameter enabled. For the Bitcoin wallet, we used the Linux time utility. For DRM service, we used (i) the Linux time API for the subsamples decryption and (ii) the Java System.nanoTime for the processing. Excepting world switch measurements, for all other experiments, we compare three system configurations: (i) standard OP-TEE deployment (No RZ); (ii) REZONE deployment (RZ); and (iii) REZONE deployment with no trusted OS preemption from the REE (RZ no IRQ). We collected 1000 samples and present the average of values within the 95^{th} percentile, which removes the effect of occasional Linux interference. For microbenchmarks, we flush the caches at the end of each test. This reduces the hot-cache effect of the previous iteration and approximates a realistic TEE setup.

6.1 Microbenchmarks

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

World switch time. The world switch time increased from an average of 5.7±0.33% to 72.5±1.7% microseconds, i.e., 13×.

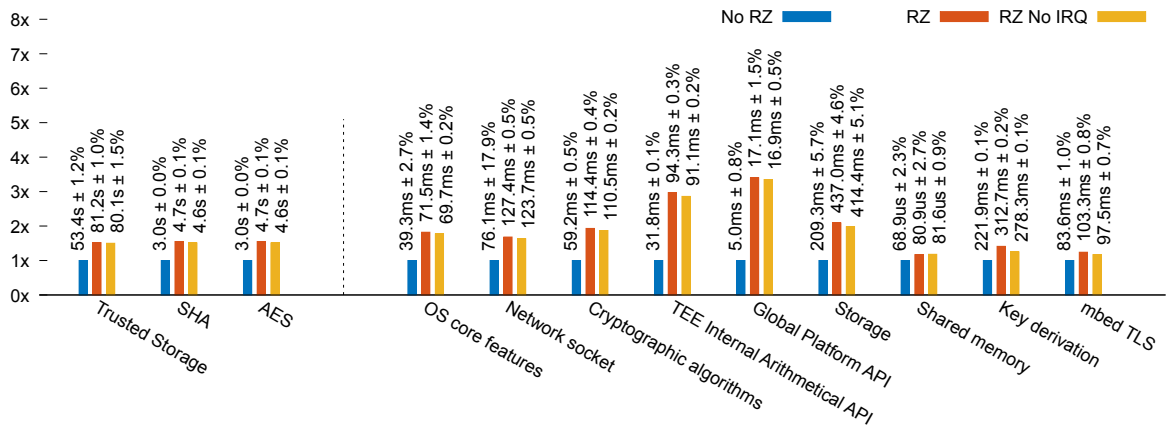

Figure 8: Geometric mean of REZONE overhead for xtest. Benchmarks (left) and unit tests (right).

This increase is the cumulative penalty of (i) additional REZONE trampoline and gatekeeper execution time, (ii) cluster-microcontroller requests, (iii) penalty of running a portion of the trampoline with caches disabled, and (iv) time of cache maintenance operations. Despite the high overhead of raw world switches, they often translate into relatively small slowdowns in real-world TEE usage scenarios (see §6.2).

GlobalPlatform API. Figure 7 presents our main results. The geometric mean of the normalized performance overhead for the nine API benchmarks is 2.327×, i.e., a significant decrease compared to the 13× overhead from the world switch. We can observe that the three APIs (i.e., *OpenSession*, *InvokeCommand*, and *CloseSession*) that explicitly call the secure world have a larger penalty, ranging from 2.25× to 4.79×. The *InvokeCommand* and *CloseSession* APIs are the ones that take less time to execute, and thus the ones with the bigger overhead, due to the over-penalty of cache-related operations (see xtest evaluation). Third, from the remaining six APIs, two have almost no penalty. This fact is related to the OP-TEE configuration which disables dynamic shared memory. With this option, *Allocate* and *Register* API implementations are identical. While calling these APIs, there is no switch to the secure world. Therefore, after the execution of *Allocate* and *ReleaseAllocatedSharedMemory*, code and data are already cached for the *Register* and *ReleaseRegisteredSharedMemory* calls. Lastly, we can observe that the system configuration without preemption (RZ No IRQ) slightly decreases the overhead, reducing the geometric mean from 2.327× to 2.325×.

xtest suite. Figure 8 summarizes our results, represented by the geometric mean of all tests for the same group. Unit tests are represented in nine groups on the left, while benchmarks are represented in three groups on the right. A detailed view of the results per test/benchmark can also be found in the supplementary material published online [15]. The group of tests that has a bigger impact is the TEE Internal API and Global Platform API, i.e., 2.97× and 3.40×, respectively. Note that this group of tests are the ones with smaller execution times and simpler secure world operations. On the other hand, tests such as mbed TLS and key derivation perform fewer but longer operations in the secure world. Since the execution time of the operations performed in the secure world is large, the performance overhead is relatively lower when compared to the TEE API groups. To complete, we tested a potential application-level optimization. Specifically, we modified one particular unit test (4007 symmetric) which consists of performing different symmetric encryption schemes (e.g., AES, DES) for incremental key sizes (e.g., 64-, 128-bit). The standard implementation has a total of 3884 secure world calls per run. We modified the implementation to perform all iterations in a single TA call. By simply batching all operations and processing them in a one-time operation, we reduced the number of calls to the secure world to 96 and were able to decrease the performance overhead from 3.65× to 1.09×.

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

6.2 Real-world Applications

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

In the case of the Bitcoin wallet, the geometric mean of the performance overhead is 1.49× for the vanilla REZONE implementation and 1.46× for a non-preemptible configuration (Figure 9). The most impacted wallet services are related to simple one-time operations. For instance, the absolute exe-

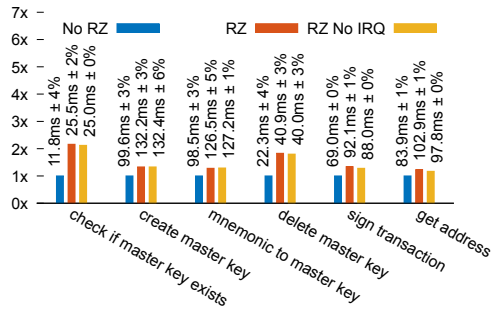

Figure 9: Relative, absolute execution time, and normalized standard deviation (over 100 runs) for Bitcoin wallet.

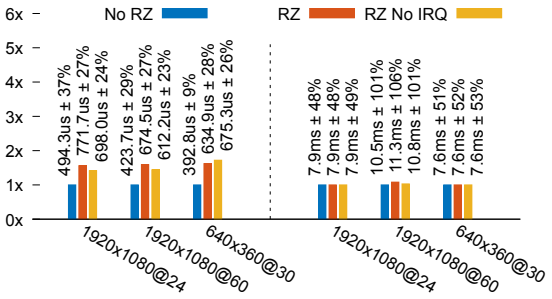

Figure 10: Performance overhead, execution time, and normalized standard deviation (over N runs) for DRM subsample decryption (left) and processing (right). N=1000, 1000, 400, for 1080@24, 1080@60, 360@30 setups, respectively.

cution time of the operation checking if a master key exists is considerably smaller (11.8ms for the standard OP-TEE deployment) than the other services, and thus the penalty is higher. Notwithstanding, the results clearly demonstrate that the impact of REZONE in longer time-consuming operations, e.g., Bitcoin sign transaction (the one likely to be executed more frequently) is low (1.34×) and one order of magnitude smaller than the overhead observed in the raw world switch.

Regarding the DRM player, Figure 10 plots our results. Overall, the geometric mean of the normalized performance overhead for the DRM subsample decryption is 1.59×. For instance, Figure 10 (left) shows that the performance penalty of individual subsample decryption for 1920x1080@60 added by REZONE (red bar) in comparison to the bare TrustZone setup (blue bar) is 1.59×. This operation is performed every time the secure world is invoked to decrypt a single subsample and return to normal world. These per-subsample overheads tend to be dwarfed when we consider the total processing time that it takes to render a video (Figure 10, right). In this case, computing the time overheads for all configurations gives time increases of 0.2%, 0.4%, and 7.7%, respectively, for 24, 30, and 60 frames per second. The higher the frame, the higher the overhead due to the larger number of trusted OS calls and the smaller duration of secure operations (smaller subsample size). Second, the REZONE configuration without preemption (RZ No IRQ) slightly improves the performance

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

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

Table 2: Impact of REZONE on the PCMark performance score assessing the REE. On the left, the impact of decreasing the interval between calls for one core. On the right, the effect of issuing TA calls (1000 ms intervals) with multiple cores.

of the decryption process, reducing the geometric mean of the normalized performance overhead to 1.52×. This benefit is also noticeable in the processing of the decoded frames and the rendering of the final image (Figure 10, right). Third, although there is a slowdown in decryption, this penalty is in the order of 100 μseconds. Hence, it is not noticeable in the playback of the video and does not impact the user experience.

6.3 Impact on the REE Performance

We observed that the performance impact from REZONE on the REE is low, even when stressing secure world calls and the number of parallel calls from concurrent CPUs. We configured the REE to run PCMark (in the quad-core configuration), while concurrently running a toy TA at different intervals (10, 100, and 1000 ms). TA performs a simple addition and returns to normal world. We compared with a setup running vanilla OP-TEE and without calling the TA while running PCMark. Table 2, on the left, presents the results. For reasonable intervals between the toy TA calls (1000 and 100), the performance penalty is low, 0.74% to 4.86%, respectively. When the interval is 10 ms (similar to a DRM workload), the impact is reasonable (12.97%) and in the order of magnitude as the ones presented in §6.2. To understand the impact on REE for having multiple CPUs issuing concurrent TA calls, we fixed the interval rate of calling the toy TA in 1000ms. Table 2, on the right, shows that the impact is low and proportional to the number of concurrent cores, achieving the largest penalty (3.73%) when the four cores are issuing concurrent TA calls.

Memory usage. Given that REZONE needs to reserve memory for the secure monitor and for allocating zones where trusted OS stacks will be deployed, less memory becomes available to the REE. In our experiments, the secure monitor requires 192KB of memory, and the default OP-TEE configuration that we use for initializing a single zone requires only 32MB. We used this OP-TEE configuration for all our tests, including running the Bitcoin wallet and DRM player (see §6.2). Considering that our platform features 4GB of RAM, this means that the memory footprint of REZONE for hosting a single trusted OS is residual. Naturally, the amount of reserved memory can be increased in order to accommodate i) single trusted OS setups where TAs have higher memory demands, or ii) multiple trusted OS stacks enclosed inside independent zones.

7 Security Evaluation

7.1 Theoretical Security Analysis

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

A1. Direct memory mapping violations. As described in §2.2, an attacker controlling the trusted OS running in S.EL1 of a given zone, may attempt to directly map unauthorized memory regions pertaining to monitor, other zones, or normal world. We prevent these attacks using a PPC to restrict access to memory resources depending on whether a zone is executing (see Figure 5). To protect the normal world, we further need to overcome some limitations of the PPC (see §4.3) by having only one active core in the cluster while executing a zone, and performing cross-core synchronization (see §5.1).

A2. PPC hijacking. The adversary may attempt to circumvent these restrictions by controlling the PPC and disabling memory protections through reconfiguration of PPC's permissions. To prevent this attack, we ensure that only the trampoline can access the PPC by requiring the trampoline to authenticate itself before using the PPC (see §5.3). This authentication involves a shared secret token established at boot time. A malicious zone can submit a request to unlock the PPC; however, an attacker has a 1 in 2^{64} chance of guessing the token correctly.

A3. Unintended cache data leakage. Instead of trying to compromise the PPC through trampoline impersonation, the attacker may leverage cached data to gain access to the secret token. To mitigate this attack, we make sure that the token is only accessible to EL3. This is guaranteed by: i) flushing the token from the caches, ii) having the PPC deny access to the token, and iii) storing the secret token in an exclusive EL3 register while a zone is executing, to allow the trampoline to authenticate itself after a zone exits. The adversary may also leverage cached data to access other memory content from monitor, other zones, or normal world. We perform cache flushes thus evicting sensitive cached data. Memory accesses will then result in direct accesses to main memory, which the PPC can control and block in case of permission violations.

A4. Cache code injection. The attacker can try to bypass the protections described above by tampering with the trampoline at runtime. Although direct access to monitor or gatekeeper memory is not possible, he may try to inject code into the trampoline by leveraging the fact that the PPC does not perform access control at cache level (see §5.4). To prevent this attack, we disable the MMU for EL3 before entering a zone, thus preventing the trampoline from fetching maliciously manipulated code, and data from being fetched upon a zone exit, allowing the trampoline to sanitize data before continuing.

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

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

Table 3: Mitigation analysis of selected CVEs with critical CVSS scores: CVEs in scope (✓), out of scope (—), insufficient information (?).

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

7.2 CVE Mitigation Analysis

Lastly, beyond our theoretical security assessment, we aim to study how deploying REZONE in real-world systems could help mitigate privilege escalation attacks resulting from the exploitation of vulnerabilities located in TEEs. As part of this study, we leverage the database of TEE-related CVEs from our previous work [11]. We then analyzed 80 CVEs reported as critical and grouped them according to the affected component: trusted OS (TO), trusted application (TA), cryptographic implementation (CI), hardware issue (HW), and bootloader (BL). CVEs that do not clearly identify the vulnerable component (i.e., trusted OS or TA) are labeled as TO/TA. We excluded four CVEs that lack enough information about the affected component. Table 3 summarizes our analysis.

Mitigation of attacks in scope. In total, we identified 66 CVEs that are in scope. These refer to vulnerabilities affecting the trusted OS or trusted applications that have the potential to be successfully exploited and may lead to privilege escalation attacks as indicated in Figure 1. In all these cases, REZONE can help counter successful exploits of these bugs by sandboxing the trusted OS inside a zone and preventing privilege escalation into normal world, secure monitor, or other TAs. In some cases, the attacker may be originally driven by a slightly different goal. For instance, CVE-2018-

		Security (Can defend against)			Scalability (Does not depend on)			Programming (Unmod. code)		Performance (Sources of overhead)			TCB (Subcomponents and total size)			
Enclave Type	System	TOS→NW (P1)	TOS→Mon (P2)	TOS_1→TOS_2 (P3)	PPC	ACU	Non-COTS Features	Runtime	API	S2-TLB	Micro. Arch. Maintanance	Trap & Emul.	Monitor	Trusted OS	Other SLOC	Total kSLOC
NW	PrivateZone [30]	—	—	—	○	●	●	○	○	○	○	●	TF-A$^+$	OPTEE	-	259.9
	OSP [12]	—	—	—	○	●	●	○	○	○	○	●	TF-A$^+$	OPTEE	OSP Hyp	255.5
	vTZ [28]	—	—	—	○	●	●	●	●	○	●	○	Custom	-	-	2.0
	Sanctuary [10]	—	—	—	●	●	○	○	○	●	○	●	TF-A$^+$	OPTEE$^+$	TA 2x	256.0
	TrustICE [55]	—	—	—	●	●	●	○	○	●	○	●	TF-A$^+$	OPTEE	-	255.0
SW	TEEv [35]	◐	◐	◐	●	●	●	○	●	●	●	○	TF-A$^+$	-	TEE-Visor	28.8
	PrOS [32]	◐	◐	◐	○	●	●	○	●	●	●	○	TF-A$^+$	-	PVisor	27.6
	ReZone	●	●	●	○	○	●	●	●	●	○	●	TF-A$^+$	-	gatekeeper	30.0

Table 4: Analysis of ReZone and similar systems: ● indicates that the system fares positively, ○ that it fares negatively.

5210 describes a vulnerability that allows an attacker to gain TEE privilege execution, which leads to retrieving device unlocking information at the TA level. This can then be used to obtain the device unlocking code. We consider this vulnerability sandboxed because an attacker cannot affect other system components.

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

8 ReZone in Perspective

8.1 ReZone and Related Systems

In this section, we put ReZone in perspective with closely related research on TrustZone-aware TEE systems. We surveyed seven representative projects that have been designed to improve the security of TEEs along different (and sometimes complementary) dimensions. Next, we provide an overview of these systems and then compare them with ReZone according to five distinct criteria. Table 4 guides our discussion.

Related TrustZone-aware TEE systems. One class of solutions aims to provide normal world enclaves using virtualization. Notable examples include vTZ [28], OSP [12], PrivateZone [30] which leverage normal world's virtualization extensions (NS.EL2) to create isolated environments in normal world. While vTZ virtualizes the full TrustZone hardware (enabling the execution of full-fledged trusted OSes in the normal world), OSP and PrivateZone provide a custom runtime environment per TA. Other systems like Sanctuary [10] and TrustICE [55] provide normal world enclaves using the TZASC. They rely on monitor and trusted OS to dynamically program the TZASC and create normal world enclaves isolated from the REE. A third class of systems aims to create software-enforced secure world enclaves. For instance, TEEv [35] and PrOS [32] aim to support multiple TEE stacks on Arm platforms where S.EL2 is not available. These solutions rely on a "hypervisor" running in S.EL1 (TEEv) or EL3 (PrOS) and perform binary instrumentation of the trusted OSes running in S.EL1 to guarantee isolation. In our work, we introduce a fourth category of systems aimed at creating hardware-enforced secure world enclaves. By relying on PPC/ACU hardware, ReZone creates secure world enclaves (i.e., zones) to effectively restrict S.EL1 privileges.

Security. To better understand the security properties offered by ReZone, we consider the threat model defined in §3. Assuming that an attacker manages to hijack the trusted OS running in S.EL1, we intend to analyze if the studied systems can prevent further privilege escalation attacks to normal world (P1), secure monitor (P2), or other trusted OSes (P3). For systems that create normal world enclaves, given that the trusted OS hosted by an enclave runs in NS.EL1 (not S.EL1), these attacks are out of scope. Focusing on systems designed to create secure world enclaves, TEEv and PrOS rely on same privilege isolation techniques (binary instrumentation) to shield trusted OS instances running in S.EL1. Although these systems can offer protection against said attacks, they require substantial manual engineering effort or the employment of binary instrumentation tools for removing privileged memory instructions from the trusted OS. In contrast, ReZone offers similar protections without the need to re-engineer existing trusted OSes, being able to secure unmodified TEE stacks.

Scalability. By scalability we mean to identify important deployability barriers on real-world platforms. We base our anal-

ysis on assessing system dependencies on specific hardware components, namely: PPC, ACU, or hardware unavailable on COTS platforms. The fewer these dependencies, the better a solution can scale. Most systems, including REZONE, depend on some PPC-like component, e.g., SMMU, to restrict accesses from system bus masters. REZONE also depends on an ACU. Sanctuary precludes PPC or ACU, but relies on the unrealistic assumption that the core ID is propagated to the bus, i.e., this solution is not practical for real-world platforms.

Programming overheads. TrustZone-aware TEE systems may require modifications to standard TEE APIs (e.g., Global Platform) and/or runtime components, i.e., trusted OS and normal world software. Some systems require modifications of trusted OS [10,35] or normal world hypervisor [28], or implementation of custom drivers [10,12,30,32,55]. Conversely, REZONE requires no modifications to TEE APIs or runtime, being able to function with existing TEE/REE stacks.

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

Trusted computing base. Different systems depend on various software components to function properly and enforce the security properties for which they were originally designed. In Table 4, we identify the components that pertain to the TCB of each system and indicate their respective sizes. We collect this information from the original papers and added reference sizes for TF-A (24,607 SLOC) and OP-TEE (230,094 SLOC). SLOC values were computed using the SLOCCount tool. Modifications to software are marked with "+". In REZONE, the TCB covers a TF-A version that includes the trampoline (1,523 SLOC) and the gatekeeper (3,822 SLOC). REZONE features a TCB size of about 30 kSLOC that approximates the TCB size of systems that achieve comparable security goals, i.e., TEEv and PrOS. Given that REZONE's gatekeeper leverages readily available board support package (BSP) code, we foresee that an optimized assembly implementation would reduce gatekeeper's size to just a few hundred SLOC.

8.2 REZONE and Armv8.4 S.EL2

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

However, despite the introduction of S.EL2, a fundamental violation of the principle of the least privilege still persists: *the secure hypervisor has unlimited privileges to arbitrarily map memory regions into the S.EL2 address space* – including memory pertaining to monitor and normal world. As a result, if an attacker manages to exploit a bug in the secure hypervisor and run code at S.EL2, Armv8.4 offers no defense mechanism that can further prevent the attacker from violating all properties P1-3 and controlling the entire system. Hijacking the secure hypervisor in such a way is not unrealistic given its relative complexity. For instance, we computed the TCB size of Hafnium [56], Arm's reference implementation for the secure hypervisor, and it features over 20 KSLoC. REZONE can still be leveraged in this setting to isolate the secure hypervisor thus fully guaranteeing P1 and P2. From a technical point of view, repurposing REZONE should not require major challenges. REZONE's trampoline can even be simplified as it only needs to shield a single S.EL2 component (akin to a single zone) and interpose context-switch events.

8.3 REZONE and Armv9 CCA

In Q2 2021, Arm introduced the Confidential Computing Architecture (CCA) whereby Armv9 CPUs are augmented with a so-called Real Management Extension (RME) [51]. RME allows ordinary software developers to instantiate a new type of isolation environments named *realms* where they can run application code. To support realms, RME extends the TrustZone architecture with two additional worlds: *root world* and *realm world*. The root world is exclusively dedicated to EL3 and it hosts the secure monitor. The realm world corresponds to a second independent instance of the secure world which is now dedicated to hosting realms. Realm world, secure world, and normal world can run code in EL0, EL1, and EL2 protection modes. In the realm world, EL2 can house a *realm manager* – equivalent to secure hypervisor – which in turn can instantiate multiple realms – i.e., confidential VMs.

Compared to Armv8, CCA introduces important security improvements. For one, the root world prevents access to EL3 memory from any other world. This architectural change effectively solves the excess of privileges of Armv8's S.EL2 or S.EL1 by preventing a compromised secure hypervisor

or trusted OS from hijacking the secure monitor, therefore enforcing P2. Similar to Armv8, CCA can also guarantee P3 given that S.EL2 can virtualize S.EL1.

Notwithstanding, *the normal world can arbitrarily be accessed by S.EL2 in secure or realm worlds*. This excess of privileges opens the door for P1 violations due to a compromised realm manager or secure hypervisor. Given CCA's early stage, it is premature to make a definitive assessment of this technology. Nevertheless, our preliminary analysis leads us to infer that REZONE may be sided with RME to further protect the normal world. Moreover, given that *RME is an optional feature starting only from Armv9.2* [8], it is not yet clear how widely OEMs will adopt RME. As of early 2022, the only announced SoCs with Armv9 CPUs feature Armv9.0 [2], which omit RME. Without RME, CPUs lack root world protection for EL3 [7] thus having the same security limitations of Armv8.4 CPUs, which may justify deploying REZONE.

8.4 Discussion

REZONE limitations and optimizations. REZONE, has two central limitations. First (a), since the core ID is not propagated to the bus, only a single TA can simultaneously run per cluster (see §4.3), which may limit TA concurrency and occasionally increase TA response time. Second (b), frequent invocation of (small duration) TAs may lead to non-negligible performance overheads due to numerous world-switch calls. To improve (a), we propose to enforce a fair-cluster scheduling policy, which balances TA execution across clusters. Since multi-core platforms typically provide multi-cluster configurations (e.g., Qualcomm 8 series SoCs since 2016 [49, 50]), this optimization allows concurrently running as many TAs as the available clusters. To improve (b), we suggest batching requests from the rich OS and serving each batch through a single secure world call. This approach reduces the number of world transitions, and consequently the number and impact of REZONE's microarchitectural maintenance operations.

Effects of REZONE's appropriation of PPC/ACU. We deduce that REZONE can leverage a platform's PPC and ACU without jeopardizing the utilization of these components for their original purposes and legacy use cases. In particular, PPC/ACU can be shared, e.g., using a spare security domain in the PPC or enabling the co-existence of the gatekeeper code with other critical or non-critical functionalities already embedded in the ACU. Numerous works in the literature provide strong isolation for software running in microcontrollers [25, 31, 33, 42, 43]. Modern low-end TEEs such as MultiZone [43] leverage minimal hardware primitives ready-available in almost all Arm Cortex-M microcontrollers to provide high-performing, robust, and lightweight enclaves.

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

Using Arm SMMU as REZONE's PPC. To leverage an SMMU as PPC, we envision minor changes in the execution flow while entering (EL3→S.EL1) and exiting (S.EL1→EL3) a zone. Trampoline's dynamic access control configuration code (see §5.3) must explicitly create page tables that enforce the required access permissions for each zone, instead of configuring permissions through memory-mapped registers. The trampoline must also invalidate the SMMU TLB to ensure the configured page tables enter in effect. The ACU must also prevent the CPU from accessing the SMMU or the page tables otherwise a zone could undo the established access control policy and fully compromise the system.

9 Related Work

Several security-oriented hardware architectures provide underlying primitives for building TEE systems [11, 38, 45], namely CPU extensions [13, 14, 45], separate co-processors [39, 47], dedicated security chips [26], and secure virtualization [1, 51]. Recently, academia has been focused on exploring RISC-V to propose novel TEE hardware architectures [9, 40]. In our work, we leverage COTS hardware primitives to provide augmented isolation within the TEE.

As for hardware-enforced TEE systems, Komodo [17] implements a small TEE monitor which provides sealed storage and remote attestation per the SGX specification. Lightweight secure world runtimes [27, 54] aim at shielding security-critical applications from untrusted OSes. TrustICE [55] and Sanctuary [10] leverage the TZASC to create enclaves within the normal world. LTZVisor [44] and Voilá [42] have leveraged TrustZone hardware to virtualize system resources for a dual OS system configuration while addressing real-time guarantees. REZONE leverages hardware orthogonal to TrustZone to provide containerization for multiple trusted OSes, while providing the same availability and real-time guarantees as existing TrustZone-assisted TEEs. CaSe [58] and SecTEE [59] allow TAs to run entirely from the cache and on-chip memory, respectively. Keystone [34] and MultiZone [23] leverage standard hardware primitives from the RISC-V ISA, i.e. physical memory protection (PMP), to isolate individual enclaves.

vTZ [28], OSP [12], and PrivateZone [30] leverage the vir-

tualization extensions available in the normal world (NS.EL2) to create isolated environments. TEEv [35] and PrOS [32] use same privilege isolation [16] to secure a minimalist hypervisor from trusted OSes, due to the lack of secure virtualization support prior to Armv8.4-A. Our solution does not rely on any hardware virtualization support and thus can be used in combination with existing hypervisors and legacy systems.

10 Conclusion

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

Acknowledgments

We thank our shepherd Aastha Mehta and the anonymous reviewers for their comments and suggestions. This work was supported by national funds through Centro ALGORITMI / Universidade do Minho, Instituto Superior Técnico / Universidade de Lisboa, and FCT under project UIDB/50021/2020 and UIDB/00319/2020. David Cerdeira was supported by FCT grant SFRH/BD/146231/2019.

References

[1] AMD. AMD Secure Encrypted Virtualization. https://developer.amd.com/sev/, 2019.

[2] Anandtech. Arm Announces Mobile Armv9 CPU Microarchitectures: Cortex-X2, Cortex-A710 & Cortex-A510. https://www.anandtech.com/show/16693/arm-announces-mobile-armv9, 2021.

[3] Android. Android Enterprise Security. https://www.android.com/static/2016/pdfs/enterprise/Android_Enterprise_Security_White_Paper_2019.pdf, 2020.

[4] Arm. Trusted Firmware-A. https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git.

[5] Arm. PrimeCell Infrastructure AMBA 3 TrustZone Protection Controller (BP147). https://developer.arm.com/documentation/dto0015/a/, 2004.

[6] Arm. TrustZone for Armv8-A. https://documentation-service.arm.com/static/602167b6873dd96c4deaf49b, 2019.

[7] Arm. Introducing Arm Confidential Compute Architecture. https://developer.arm.com/documentation/den0125/latest, 2021.

[8] Arm. Arm Architecture Reference Manual. https://developer.arm.com/documentation/ddi0487/latest, 2022.

[9] R. Bahmani, F. Brasser, P. Jauernig Dessouky, M. Klimmek, A. Sadeghi, and E. Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *Proc. of USENIX Security*, 2021.

[10] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *Proc. of NDSS*, 2019.

[11] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. SoK Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *Proc. of S&P*, 2020.

[12] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek. Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices. In *Proc. of USENIX ATC*, 2016.

[13] V. Costan and S. Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016.

[14] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proc. of USENIX Security*, 2016.

[15] D. Cerdeira, J. Martins, N. Santos, S. Pinto. ReZone Source Code and Experimental Results. https://gitlab.com/ESRGv3/rezone/.

[16] N. Dautenhahn, T. Kasampalis, Will Dietz, J. Criswell, and V. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. *Comput. Archit. News*, 2015.

[17] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proc. of SOSP*, 2017.

[18] G. Beniamini. Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption. https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html, 2016.

[19] G. Beniamini. QSEE privilege escalation vulnerability and exploit. http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html, 2016.

[20] G. Beniamini. TrustZone Kernel Privilege Escalation (CVE-2016-2431). http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html, 2016.

[21] G. Beniamini. Unlocking the Motorola Bootloader. http://bits-please.blogspot.com/2016/02/unlocking-motorola-bootloader.html, 2016.

[22] G. Beniamini. War of the Worlds - Hijacking the Linux Kernel from QSEE. https://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html, 2016.

[23] C. Garlati and S. Pinto. A Clean Slate Approach to Linux Security RISC-V Enclaves. In *Embedded World Conference*, 2020.

[24] Google. ExoPlayer. https://github.com/google/ExoPlayer.

[25] M. Grisafi, M. Ammar, M. Roveri, and B. Crispo. PISTIS: Trusted computing architecture for low-end embedded systems. In *Proc. of USENIX Security*, 2022.

[26] Trusted Computing Group. TPM. https://trustedcomputinggroup.org/work-groups/trusted-platform-module/, 2018.

[27] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone. *Proc. of MobiSys*, 2017.

[28] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vTZ: Virtualizing ARM TrustZone. In *Proc. of USENIX Security*, 2017.

[29] Jamie Cui. Bitcoin Wallet implementation using Trusted Execution Environments. https://github.com/Jamie-Cui/bitcoin-wallet, 2018.

[30] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. Kang. PrivateZone: Providing a Private Execution Environment Using ARM TrustZone. *TDSC*, 2018.

[31] H. Janjua, M. Ammar, B. Crispo, and D. Hughes. Towards a Standards-Compliant Pure-Software Trusted Execution Environment for Resource-Constrained Embedded Devices. In *Proc. of SysTEX*, 2019.

[32] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek. PrOS: Lightweight Privatized Secure OSes in ARM TrustZone. *TMC*, 2019.

[33] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek. uXOM: Efficient eXecute-Only Memory on ARM Cortex-M. In *Proc. of USENIX Security*, 2019.

[34] D. Lee, D. Kohlbrenner, S. Shinde, Krste Asanović, and D. Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proc. of EuroSys*, 2020.

[35] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang. TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms. In *Proc. of VEE*, 2019.

[36] Linaro. ClearKey OPTEE AOSP drm plugin. https://github.com/linaro-mmwg/clearkeydrmplugin, 2019.

[37] M. Lu. TrustZone, TEE and Trusted Video Path Implementation Considerations. https://www.arm.com/files/event/Developer_Track_6_TrustZone_TEEs_and_Trusted_Video_Path_implementation_considerations.pdf, 2018.

[38] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers*, 2018.

[39] T. Mandt, M. Solnik, and D. Wang. Demystifying the secure enclave processor. *Black Hat Las Vegas*, 2016.

[40] P. Nasahl, R. Schilling, M. Werner, and S. Mangard. HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment. In *Proc. of Asia CCS*, 2021.

[41] PallyCon. About Google Widevine DRM. https://pallycon.com/google-widevine-drm/, 2019.

[42] S. Pinto, H. Araújo, D. Oliveira, J. Martins, and A. Tavares. Virtualization on TrustZone-enabled Microcontrollers? Voilà! In *Proc. of RTAS*, 2019.

[43] S. Pinto and C. Garlati. Multi Zone Security for Arm Cortex-M Devices. In *Embedded World Conference 2020*, 2020.

[44] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. LTZvisor: Trustzone Is The Key. In *Proc. of ECRTS*, 2017.

[45] S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.*, 2019.

[46] Qualcomm. FY 2021 3rd Quarter Earnings Release. https://investor.qualcomm.com/financial-information/historical-financial-results.

[47] Qualcomm. Qualcomm Secure Processing Unit SPU230. http://216.117.4.138/files/epfiles/1045b_pdf.pdf, 2019.

[48] Qualcomm. An Introduction to Access Control on Qualcomm Snapdragon Platforms. https://www.qualcomm.com/media/documents/files/an-introduction-to-access-control-on-qualcomm-snapdragon-platforms.pdf, 2020.

[49] Qualcomm. Snapdragon 820 Mobile Platform. https://www.qualcomm.com/products/snapdragon-820-mobile-platform, 2022.

[50] Qualcomm. Snapdragon 835 Mobile Platform. https://www.qualcomm.com/products/snapdragon-835-mobile-platform, 2022.

[51] Arm. Realm Management Extension. https://developer.arm.com/documentation/den0126/latest, 2021.

[52] Arm. System Control Processor Firmware. https://developer.arm.com/tools-and-software/open-source-software/firmware/scp-firmware, 2021.

[53] Xilinx. Isolate Security-Critical Applications on Zynq UltraScale+ Devices. https://www.xilinx.com/support/documentation/white_papers/wp516-security-apps.pdf, 2020.

[54] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. *Comput. Archit. News*, 2014.

[55] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Proc. of DSN*, 2015.

[56] TrustedFirmware. Hafnium. https://www.trustedfirmware.org/projects/hafnium/, 2021.

[57] Trustonic. Trustonic Application Protection Delivers Comprehensive Security for Mobile Financial Services. https://www.trustonic.com/markets/financial-services/, 2018.

[58] N. Zhang, K. Sun, W. Lou, and Y. T. Hou. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *Proc. of S&P*, 2016.

[59] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng. SecTEE: A Software-Based Approach to Secure Enclave Architecture Using TEE. In *Proc. of CSS*, 2019.