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

David Cerdeira
Centro Algoritmi
Universidade do Minho
david.cerdeira@dei.uminho.pt

Nuno Santos
INESC-ID / Instituto Superior Técnico
Universidade de Lisboa
nuno.santos@inesc-id.pt

Pedro Fonseca
Department of Computer Science
Purdue University
pfonseca@purdue.edu

Sandro Pinto
Centro Algoritmi
Universidade do Minho
sandro.pinto@dei.uminho.pt

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

***Index Terms*—TEE, TrustZone, Security Vulnerabilities, Arm**

I. Introduction

Trusted Execution Environments (TEE) are a key security mechanism to protect the integrity and confidentiality of applications. By leveraging dedicated hardware, TEEs enable the execution of security-sensitive applications inside protected domains isolated from the platform's operating system (OS). Arm TrustZone [1] has become the de facto hardware technology to implement TEEs in mobile environments and has been employed in industrial control systems [2], servers [3], and low-end devices [4]. In the future, where trillions of TrustZone-enabled IoT devices are expected worldwide [5], TEEs can provide secure environments for data processing at the edge.

TrustZone-assisted TEEs are generally assumed to be more secure than modern OSes due to the hardware-based separation enforced by TrustZone technology and their smaller Trusted Computing Base (TCB), which is several orders of magnitude smaller than standard OSes'. For this reason, TEEs have become widely adopted for securing mobile devices against malware [6–10]. For instance, Android platforms incorporate TrustZone-assisted TEEs to secure application-specific operations involving, e.g., user authentication [11], online banking [12], or DRM [13]. Unfortunately, some of these systems have been exploited over the past years, which casts doubt on the real security guarantees that existing commercial TEEs can effectively provide.

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

One first observation is that TEE systems have a long history of *critical implementation bugs*. Numerous bugs have been (and continue to be) found inside TEE applications – named Trusted Applications (TAs) – and inside the trusted kernel responsible for managing the TEE runtime. Many bugs involve classic input validation errors, such as buffer overflows. As shown by multiple attacks, these bugs can be leveraged to hijack Android's Linux kernel or to entirely compromise the TEE kernel of devices featuring TEEs by Qualcomm [14, 15], Trustonic [16, 17], or Huawei [18].

Second, exploiting vulnerable TAs is facilitated by the *numerous architectural deficiencies* of TrustZone-assisted TEE systems. For instance, the memory protection mechanisms commonly found in modern OSes, e.g., ASLR or page guards, are almost absent or ill-implemented in most analyzed systems. TEE systems also tend to expose a large attack surface, including dangerous TEE kernel system calls that can be invoked by TAs. For example, on Qualcomm's TEE, any TA can map in memory regions of the host OS. As a result, by hijacking a vulnerable TA, e.g., leveraging a buffer overflow, an attacker can easily control Android [15].

Third, *important hardware properties are overlooked* in most TrustZone systems at the architectural and microarchitectural levels, which can compromise the security of the TEE. Some vulnerabilities are caused by unexpected behavior of trusted hardware components due to microarchitectural side-channels (e.g., in caches) [19–23]. Others are caused by components that can be leveraged to exfiltrate sensitive data from TEE-restricted

DOI 10.1109/SP40000.2020.00061

memory, for instance via reconfigurable hardware (FPGAs) embedded into the modern SoCs [24, 25].

Although many of these problems remain difficult to solve for software systems in general, we observe that the defense mechanisms currently implemented in the studied TEEs lag considerably behind the state-of-the-art defenses incorporated into commodity mainstream OSes and proposed by the research community. We argue that, by adopting up-to-date defenses, commercial TrustZone-assisted TEEs could be made significantly more secure and capable of countering many prevailing vulnerabilities. We present a collection of relevant defense techniques according to their suitability to address specific kinds of issues: architectural, implementation or hardware issues.

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

II. Background and Motivation

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

A. Trusted Execution Environment and Arm TrustZone

A TEE provides an isolated environment for secure processing of sensitive data, without the need to rely on the integrity of the OS. TEEs aim at guaranteeing the secure execution of programs, known as TAs or *trustlets*. TEE systems rely on trusted hardware, such as Arm TrustZone [26], which has been supplied on Arm application processors (Cortex-A) since 2004 [27] and it was recently re-engineered for the new generation of Arm microcontrollers (Cortex-M) [28]. Our work focuses primarily on the Cortex-A TrustZone implementation, which is widely used on mobile devices.

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

B. Software Architecture of TrustZone-assisted TEE

The typical software architecture of a TrustZone-assisted TEE runs the untrusted OS inside NW – named Rich Execution Environment (REE) – and the TEE software components run in the SW (see Figure 1). Inside SW, the *trusted OS* runs in supervisor mode (protection ring EL1) and provides runtime support for sustaining the lifecycle of TAs, which run in user mode (protection ring EL0). The core of the trusted OS is the trusted kernel, which provides the basic OS primitives for scheduling and managing TAs. The trusted OS additionally implements device drivers for accessing trusted peripherals, handles cross-world requests through the world switching SMC instruction and shared memory, and implements shared libraries (e.g., cryptographic) and TEE primitives, namely remote attestation, trusted I/O, and secure storage.

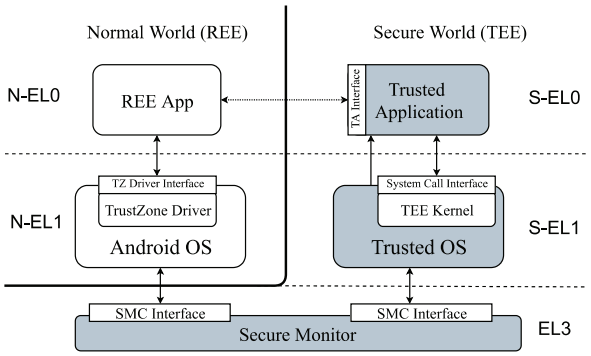

Figure 1. Software architecture of a TrustZone-assisted TEE system.

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

C. Attacking TEE-enabled Devices

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

Compromising the TEE kernel: Targeting Qualcomm TEE (QSEE), the TEE system developed by Qualcomm, Gal Beniamini showed how to hijack the TEE kernel from an unprivileged user-level NW application in two different ways. One way requires escalating privileges into the Linux kernel

ID	Ref	Year	Description	Component	Vulnerabilities	Impact
E1	[29]	2015	Input validation weakness can be used as a zero-write primitive anywhere on memory QSEOS's virtual memory to obtain arbitrary code execution in trusted OS. Requires root privileges in Linux kernel.	SW Monitor	[30]	Full control of TZ kernel
E2	[31]	2015	Exploits bug in the TrustZone Linux driver, which allows an attacker to obtain root privileges and thus launch the E1 attack.	NW Driver	CVE-2014-4322	Full control of Linux kernel
E3	[32]	2016	Vulnerability in Android's Mediaserver process which allows an unprivileged REE application to gain access the Qualcomm's TrustZone interface driver. When used with E1 and E2, allows an unprivileged application to obtain trusted OS-level arbitrary execution.	NW Service	CVE-2014-7920, CVE-2014-7921	Full control of Android Mediaserver
E4	[33]	2016	Privilege escalation attack to obtain arbitrary execution in the context of a TA. The vulnerability occurs in the Widevine TA, and can be exploited by accessing the TrustZone interface Linux driver using E3.	SW TA	CVE-2015-6639	Full control of Widevine TA
E5	[14]	2016	Lack of input validation in Qualcomm's trusted OS system calls allows a TA to write to any address within the OS and hijack the TEE kernel. Requires privilege escalation into TA through the TA's interface.	SW Kernel	CVE-2016-2431	Full control of TZ kernel
E6	[15]	2016	An attacker with TA-level execution privileges can gain control of the Linux kernel. This attack can be built upon E4.	NW Kernel	Bad system call	Full control of Linux kernel

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

(see Figure 1) in several steps. First, use exploit E3 to control Android's Mediaserver, which has privileged access to the TEE driver. Then elevate privileges into the Linux TrustZone driver to access the SMC interface (E2). A third exploit (E1) takes advantage of a bug in the TEE kernel and achieves arbitrary code execution with EL1 privileges in SW. Once in control of the TEE kernel, an attacker can launch other attacks, e.g., hijack a guest TA to extract secret keys and break Android's full disk encryption [34], or unblock the device bootloader [35]. A second way to compromise the TEE kernel only requires access to the interface of a vulnerable TA. Using E4, an attacker can hijack the Widevine TA, a DRM service for Android OS. Then, through a vulnerability in the system call interface, the attacker can further elevate privileges into the TEE kernel (E5).

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

The extent of the problem. Several other exploits have been developed for the Qualcomm TEE [17, 36–38]. Beyond mobile devices shipping Qualcomm chips, other platforms have been attacked, namely devices running Trustonic's TEE system, renamed from Mobicore to Kinibi [16, 17, 39, 40], and Huawei's proprietary TEE named Trusted Core [18, 41]. Most of these exploits adopt the divide-and-conquer strategy presented in Table I. Considering that Trustonic's TEE is estimated to run on 1.7 billion devices (mostly Samsung's) and Huawei's mobile devices are widely adopted (200 million sold in 2018), TEE flaws can have a large impact worldwide.

III. Overview

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

A. Methodology of our study

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

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

Analyzed TEE Systems: We analyzed TEE systems by Qualcomm, Trustonic, Huawei, Nvidia, and Linaro. Nvidia maintains a proprietary TEE used mostly for Nvidia chips. Linaro maintains OP-TEE, an open source TEE software very popular for TrustZone development. All these systems are actively maintained, are widely adopted for commercial purposes, and a fair amount of information can be obtained about them. We excluded research prototypes (e.g., Andix [2]) or commercial products not currently deployed at scale (e.g., SierraTEE [42]). We also consider relevant cross-cutting vulnerabilities, e.g., hardware side-channels. For the sake of readability, henceforth,

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

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

we refer to each analyzed TEE by the company name rather than by software name (e.g., Qualcomm TEE means QSEE).

Data sources: We resorted to multiple sources that we grouped into four areas (see Table II). We analyzed bug reports from the CVE database [43] relative to the TEE systems under study. We retrieved the CVE reports published officially by Qualcomm [44, 45], Nvidia [46] and Huawei [47] which are documented also in their respective security bulletins. We gathered additional CVE reports by searching for relevant keywords, e.g., the TEE names, "TrustZone", etc. We also collected bug reports from Samsung Vulnerabilities and Exposures (SVE) database [48] which have not been assigned specific CVE IDs. We analyzed scientific publications (SP) in major security conferences from the past 10 years, miscellaneous reports (MR) available online (e.g., [17, 33, 49–52]), and inspected source code (SC) for TEEs' with public source code, namely Linaro's OP-TEE. For OP-TEE, we also analyzed its changelog to identify security fixes and interviewed the system designers.

Classification of disclosed security vulnerabilities: After collecting the vulnerability reports, we manually analyzed and categorized them. For the vulnerabilities assigned with a CVSS score [53], we adopted a classification metric based on the attribute score. Our rating system comprises four categories: *critical* (CVSS $\geq$9), *severe* (CVSS [7,9[), *medium* (CVSS [5,7[), and *low* (CVSS [0,5[). The severity of a specific vulnerability may have different security implications. A critical vulnerability is normally one that can lead to a complete compromise of confidentiality or integrity in the TEE, in the REE, or both.

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

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

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

Table III
Number of disclosed CVEs per system from 2013 to 2018.

B. Summary of Observations

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

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

Although it stands out that the Qualcomm TEE accounts for the largest fraction of disclosed vulnerabilities (74%), we caution that we cannot conclude from this data that it is the least secure TEE; or similarly compare individual TEEs. This is due to the disparity in methodology with regards to the CVE reporting process and could simply be a consequence of higher reporting diligence of Qualcomm developers and users. However, these results are useful because they allow us to establish a lower bound on the vulnerabilities of such systems, reason about aggregate trends, and compare general TEE trends against the trends of other types of systems.

For instance, we observed that during the same time window the entire Linux operating system, which is several orders of

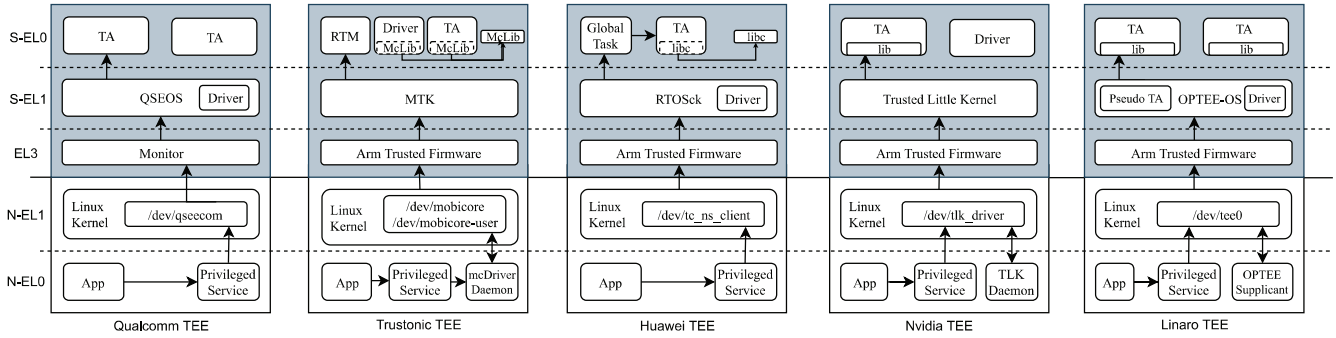

Figure 2. Detailed architecture of the studied TEE systems. A few relevant common features include: (a) the communication between a NW application and the SW is mediated by a privileged OS daemon which uses a TrustZone driver to issue SMC calls to the SW; (b) in four cases, the monitor is based on ATF [54], which consists of the reference implementation provided by Arm for the secure bootloader and monitor software.

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

C. Sources of Vulnerabilities in TrustZone-assisted TEEs

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

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

IV. Architectural Issues

This section presents the main architectural security issues of existing TEE systems. We group these issues into several categories, and refer the reader to the diagram of Figure 2 which presents the specific internal details of each system.

A. TEE Attack Surface

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

I01. SW drivers run in the TEE kernel space: In general, a TEE system requires the existence of drivers in the SW to mediate access to security-sensitive devices, e.g., a fingerprint sensor for user authentication purposes, or the display framebuffer for secure output of DRM-protected content. Given that drivers tend to be complex and a traditional source of bugs, they should not run in the TEE kernel space (i.e., in S-EL1 mode). Trustonic and Nvidia follow this approach by adopting a microkernel architecture where drivers run in the SW user space (S-EL0). In contrast, Qualcomm, Huawei, and Linaro run TEE drivers in S-EL1 mode. Both Qualcomm and Linaro adopt a monolithic architecture where all the privileged code runs in kernel space. Huawei delegates some of the trusted OS functionality to user space, namely the job of controlling the lifecycle of TAs which is assigned to a privileged TA called GlobalTask (see Figure 2).

I02. Wide interfaces between TEE system subcomponents: These interfaces have become worryingly large for TEE systems. In Android OS, at least four daemons have privileged access to the TrustZone driver. The SMC call interface exposed by the TEE kernel gives NW software access to a considerable number of TAs (e.g., Trustonic TEE counts 32 different TAs). The set of commands handled by TAs also tends to be fairly large. For instance, the Widevine TA implements 70 different commands, many of them manipulate complex media data structures. The TEE kernel exposes a large number of system calls to TAs: 69 syscalls in Qualcomm's TEE. Moreover, access permissions to the TEE system calls are frequently coarse-grained, such as in Qualcomm TEE where TAs have promiscuous access to all system calls. In certain cases, the interface provided by the secure drivers can grow very large, such as in the Trustonic TEE, where the TA that controls access to the fingerprint device driver gives access to virtually every TA deployed in the TA. Most of these issues have been instrumental for the development of the exploits listed in Table I.

I03. Excessively large TEE TCBs: Part of the design philosophy of a TEE system is that it should rely on a small TCB. To verify whether this principle holds for the studied TEE systems, we analyzed their TCB sizes based on their firmware and, when available, on their source code. Given that TAs implement security-sensitive REE functions, we include in the TCB both

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

Table IV
TCB sizes of TEE systems vs. reference OSes (respectively above and below the middle line): Values obtained from TEE binaries and loadable TAs in firmware / system image file system. For open source systems, software was compiled enabling optimizations. Lines of code were counted using SLOCCount [55].

trusted OS and TAs. Table IV presents our results comparing them against a few reference OSes. We find that TCBs of TEE systems are substantial, e.g., reaching 1.6 MB in the Qualcomm TEE. Further, these numbers are conservative since additional TAs that are not included the firmware package can be dynamically loaded. Strikingly, some TAs have individually considerable sizes. With such sizes, confidence in the full correctness of these TAs is weakened: since TAs accept inputs from the NW via SMCs, potential vulnerabilities are exposed to easy exploitation. To put TCB sizes in perspective, Table IV shows that although existing TEE kernels are significantly smaller than the Linux kernel (by about three orders of magnitude), most of them are growing considerably larger than a microkernel of comparable complexity (seL4).

B. Isolation between Normal and Secure Worlds

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

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

In contrast, Trustonic TEE prevents TAs from mapping in and modifying physical memory. Instead, this operation is restricted to specific driver TAs. Hence, TAs willing to exchange data volumes via shared memory must issue a request to a dedicated driver TA. Samsung uses this approach to split the functionality of the TrustZone-based Integrity Measurement Architecture (TIMA): a TA driver provides the ability to map physical memory while another TA uses this service to measure the integrity of system image. A white list is used to prevent access to the TA driver by arbitrary TAs. However, the white list is hard-coded in the TA driver and the number of allowed TAs reaches 34, which is fairly large. By compromising any of these TAs, an attacker has free way to hijack Android.

I05. Information leaks to NW through debugging channels: Another source of isolation breaches is caused by leakage of information from the SW to the NW via TEE debug mechanisms. Some exploits described in Table I have been facilitated by this feature. A privilege escalation attack [18] leverages a system call of the Huawei TEE that allows a TA application to dump its stack trace to a memory region in the NW. Using this mechanism, the attacker can learn the physical address space of the GlobalTask and use this information to craft the exploit. Debugging logs exposed to the NW are also common in the Trustonic TEE which may help disclose sensitive information about the internals of TAs.

C. Memory Protection Mechanisms

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

I06. Absent or weak ASLR implementations: In all analyzed TEE systems, ASLR is either absent or poorly implemented. In Trustonic TEE, TAs are all loaded into the same fixed address in the virtual address space (0x1000). Each TA is provided with a common library which is also mapped to a constant address for

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

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

each TA (0x7D01000). Thus, any vulnerability found in a TA can be exploited without requiring extra effort in determining the TA's loading address. Furthermore, this common library, named mcLib (see Figure 2), contains a substantial amount of code, which can provide a source of gadgets to call functions, invoke trusted OS system calls, etc.

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

I07. No stack cookies, guard pages, or execution protection: In addition to ASLR, modern OSes employ additional memory protection mechanisms. Stack cookies (SC) are unique values that help detect stack smashing instances and abort the program execution. Guard pages (GP) delimit the mutable data segments in each process (namely, stack, heap, and global data) to prevent attackers from using an overflow in one segment to corrupt another by triggering a fault in case of illegal access. Execution protection (XP) prevents programs from executing within certain memory regions and can be achieved by various means. On Arm, the WXN bit in the SCTLR register can be used whereby writable memory regions are implicitly marked as Execute Never (XN). Another option is to use memory page attribute XN, Unprivileged XN (UXN), and Privileged (PXN).

However, TEE systems only partially implement these mechanisms (see Table V), which has facilitated exploitation [18]. Trustonic TEE, in spite of its security-driven goals, lacks stack cookies, making it relatively easy to exploit stack overflows in vulnerable TAs. It allocates both globals and stack from the TA's data segment without providing guard pages in between. Moreover, the memory layout places the stack at the end of the data segment and the globals before it; this is the perfect configuration for overflowing one region into the other. Qualcomm TEE implements randomized pointer-sized stack cookies, but it does not provide guard pages between globals, heap, and stack. Huawei TEE has no stack canaries, no data execution protection, and no write-protected .text section, possibly because Huawei TEE is based on the Micrium μ/OS, an RTOS which leaves aside most of the said memory protection mechanisms to deliver maximum performance.

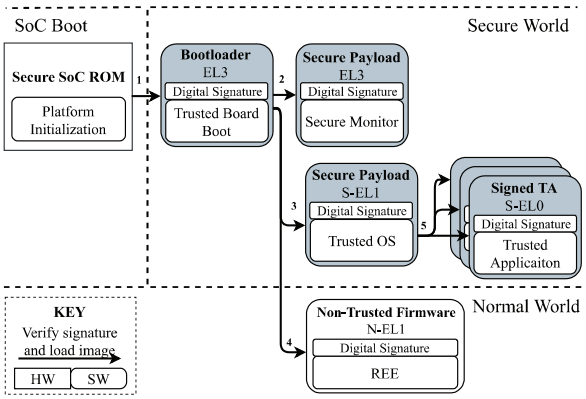

Figure 3. Secure boot process: Implements a chain of trust that starts with the execution of a trusted component – Trusted Board Boot – stored in an on-SoC ROM. Then, each loaded component verifies the authenticity and integrity of the subsequent module, or modules, and loads them if no anomalies are detected. A vendor digitally signs the SW image with its private key, while the respective public key (or its digest) is burned, or flashed into a one-time programmable memory, typically eFuses. The public key is used to verify that the binary has not been modified and it was provided by the vendor.

D. Trust Bootstrapping

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

I08. Lack of software-independent TEE integrity reporting: Secure boot ensures the authenticity of the software running on a device. Figure 3 illustrates a possible secure boot process, including the booting of TAs. However, Arm TrustZone lacks the hardware mechanisms for securely reporting the software integrity measurements to a remote third party. In the absence of hardware support, remote attestation needs to be implemented in software by one of the TEE components. This weakens the security of remote attestation as it requires the correctness of all SW software of the trust chain running in EL3 mode.

I09. Ill-supported TA revocation: Problems have been identified with the way Android OEMs deal with TA revocation [17]. TA revocation is necessary to prevent patched TAs from being downgraded. Updates allow for vulnerabilities and other errors to be corrected, increasing the overall security of the device. To make them easier to update, TAs are usually loaded from the REE filesystem and to prove their authenticity they are digitally signed. However, the TEE must revoke old TAs to prevent attackers in the REE from intentionally loading an old, known vulnerable TA and exploiting it to gain code-execution within the TEE. The successful downgrading of the Widevine TA to a previous, known vulnerable, version in Qualcomm and Trustonic TEEs has been shown [17].

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

Table VI
Number of bug reports involving implementation issues.

V. Implementation Issues

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

A. Validation Bugs

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

I10. Validation bugs within the secure monitor: By exploiting a bug in the secure monitor, an attacker can automatically gain full control of the device. For instance, the vulnerability abused by exploit E1 for hijacking the Qualcomm TEE kernel (see Table I) allowed an attacker to write a zero double word anywhere in the SW memory by crafting an input into an SMC call. To reduce the chance of critical bugs, most TEE systems (excepting Qualcomm TEE) use Arm's reference monitor (ATF) implementation (see Figure 2). Unfortunately, critical validation bugs have been reported within ATF itself. Ironically, one bug was located on a C macro whose goal was to help detect arithmetic overflows (CVE-2017-9607). Shown in Listing 1, any AArch32 code relying on this macro to detect integer overflows is not protected. This means that multiple monitor entry points that use this macro could be vulnerable.

I11. Validation bugs within TAs: Besides the secure monitor, TAs are mostly exposed to attacks from the NW through the SMC interface. As it turns out, the largest fraction of vulnerability reports in TEE systems corresponds to validation bugs within TAs. For instance, critical vulnerabilities in the ESECOMM trustlet can be leveraged to compromise client applications such as Samsung Pay [16]. In Trustonic TEE, validation bugs can be exploited systematically using the respective bug fixes [39]. Some TA validation bugs (e.g., CVE-2016-5349) may allow for direct privilege escalation into the Linux kernel through boomerang attacks [56], in which a vulnerable TA does not properly validate the input memory addresses, allowing an attacker to access NW memory region and read or write memory allocated to REE apps or OS.

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

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

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

Listing 2. Reverse-engineered syscall from Huawei TEE (RTOSck) without any input check. An attacker can overwrite memory anywhere in NW or SW.

I12. Validation bugs within the trusted kernel: By hijacking a TA, an attacker may successfully elevate its privileges by exploiting a vulnerability in the TEE kernel's system call interface. For instance, an attack on the Huawei TEE [18] relied on a vulnerable system call where its inputs are entirely unchecked for bypassing a security check within the trusted kernel (see Listing 2). Even more worrisome, the Qualcomm TEE kernel lacks any code for validating supplied input pointers, which means that all the system calls are vulnerable [14].

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

B. Functional Bugs

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

I14. Bugs in memory protection: Some functional bugs may introduce security vulnerabilities in the memory protection mechanisms of a TEE system. For instance, a vulnerability

reported for ATF [57] involves a configuration error of memory translation tables which allows read-only memory areas to always be executable in the context of the S-EL1. In OP-TEE, we identified 15 bug reports causing memory protection vulnerabilities. For instance, one error in the OP-TEE's secure monitor code responsible for saving and restoring FIQ registers for ARMv7 may allow the REE to escalate privileges to obtain code execution in the TEE [58].

I15. Bugs in configuration of peripherals: Misconfiguration of certain peripherals may also be security-critical. In Qualcomm TEE, a flaw disclosed as CVE-2016-10423, allows a TA to read data on an SPI interface previously opened by another TA due to non-exclusive access of the SPI bus. In OP-TEE, one patch [59] aimed to fix a misconfiguration of the pseudo random number generator causing an insufficient source of entropy for the cryptographic libraries used within OP-TEE.

I16. Bugs in security mechanisms: Another potential source of vulnerabilities is the existence of bugs in the implementation of security protocols or cryptography primitives. In ATF, an attacker could bypass the Amlogic S905 SoC secure boot process [51] due to a deficiency in the authentication checks, where only the integrity of the boot image was checked, not the signature. In OP-TEE, for example, a Bellcore attack vulnerability in LibTomCrypt could compromise a private RSA key (CVE-2017-1000412), and a hardcoded security key for RPMB result in the key leakage (fix on 23 Jan 2017).

C. Extrinsic Bugs

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

I17. Concurrency bugs: Caused by the interference of multiple concurrent programs, we consider concurrency bugs as extrinsic because their manifestation depends on factors external to the program itself (e.g., thread interleaving). Some concurrency bugs may introduce security vulnerabilities within TEE systems. For instance, in OP-TEE, one bug due to concurrent access to the file system by different TAs [60] allowed a TA to delete a directory on trusted storage while being created by another TA. Samsung reported two race condition vulnerabilities in the TIMA driver deployed in Trustonic TEE (SVE-2017–8974 and SVE-2017–8975). A specific instance of race conditions may lead to TOCTOU vulnerabilities, where some aspect of the system state changes after a condition check, such that the condition-check results are no longer valid. A TOCTOU vulnerability was reported in a DRM TA of the Nvidia TEE which may lead to privilege escalation (CVE-2017-6296).

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

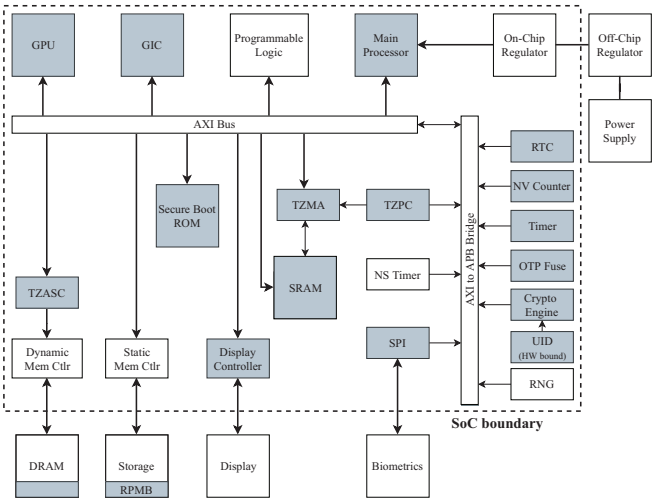

Figure 4. Hardware architecture of a TrustZone-assisted TEE system, including programmable logic present in FPGAs. The fully shaded boxes represented the trusted components exclusively allocated to the TEE software running in the SW, SPI/UART, for example, allow communication with off-SoC peripherals (e.g., for biometric authentication or smartcard interaction). Partially colored boxes represent components that can be partially, or totally, restricted to the SW, such as DRAM, and storage (e.g., to provide secure storage to TAs).

VI. Hardware Issues

TEEs rely not only on the correctness of the software architecture and implementation, but also on the correctness of trusted hardware components. Figure 4 provides an overview of the typical hardware architecture of a TrustZone-assisted TEE system and shows how these components are connected by an AXI bus. Since hardware components are part of the TCB of a TEE, the TEE developers must correctly configure and interface with these components, as well as carefully take into consideration all the implications of the microarchitecture.

A. Architectural Implications

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

I19. Attacks through reconfigurable hardware components: Reconfigurable platforms, i.e., FPGA SoCs, combine a conventional CPU architecture with programmable hardware logic. Although there is no evidence of massive adoption of reconfigurable platforms in the context of mobile devices, OP-TEE supports the Xilinx Zynq-7000 and Zynq UltraScale+ platforms on its mainline. Unfortunately, the addition of new hardware increases the attack surface. Configurable hardware within FPGA SoCs is typically connected to the main bus, which means that hardware must block access to memory regions that are managed by the software running in the main CPU. On TrustZone-enabled systems, the AMBA AXI

Component	Attack	Device	SoC	TEE	Outcomes
Cache	[19]	Freescale i.MX53	i.MX53 (ARMv7-A)	–	Cache rootkit can evade NW and SW detection
	[20]	Raspberry Pi2	BCM2836 (ARMv7-A)	Self Developed	AES 128-bit key recovery
	[21]	Galaxy S6	Exynos 7420 (ARMv8-A)	Trustonic TEE	AES 128-bit key recovery
	[22]	Freescale i.MX53	i.MX53 (ARMv7-A)	Self-Developed	AES 128-bit key recovery
	[23]	Samsung Tizen TV	(ARMv7-A)	SecureOS	Cross-Core Covert Channels demonstrated by transmitting images from SW to NW
		Hikey	Kirin 620 (ARMv8-A)	Linaro TEE	
Branch Predictor	[61]	LG Nexus 5X	Snapdragon 808 (ARMv8-A)	Qualcomm TEE	Extract 256-bit private keys from Keystore TA
DRAM	[62]	-	Cortex A-9 (ARMv7-A)	Trusty	Derive RSA private key

Table VII
Microarchitectural issues exploited to attack TrustZone-assisted TEEs.

interface includes an additional control bit (NS bit) for both read (ARPROT) and write (AWPROT) channels on the main system interconnect. This lets all hardware components become aware of the security state of the CPU. Nevertheless, some unusual exploits can take advantage of reconfigurable hardware logic to break the security of TrustZone-based systems [24, 25]. One attack explores malicious hardware deployed on an FPGA to break the secure boot process [24]. In a study about NS bit propagation to FPGA, six different attacks were exposed using small malicious modifications on the hardware logic [25].

I20. Attacks through energy management mechanisms: Software-exposed energy management mechanisms can pose significant challenges to system security, possibly in subtle ways. For instance, CLKSCREW [63] relies on a malicious (non-secure) kernel driver to push frequency and voltage regulators to operate beyond the vendor-recommended limits, until the point of inducing faulty computations. By influencing the computation of SW operations, it is possible to break the TrustZone hardware-enforced boundaries to extract secret cryptographic keys and bypass code signing operations.

B. Microarchitectural Side-Channels

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

I21. Leaking information through caches: On TrustZone-enabled processors, cache memory is shared between the secure and normal worlds. Although the secure cache lines are not accessible by the NW, both worlds are guaranteed equal rights when competing for the use of cache lines. This cache coherence design improves system performance at the cost of cache contention between the two worlds [19]. This contention is the main source of exploitation for extraction of information from the SW by monitoring caches from the NW. R. Guanciale et al. [20] implemented a low-noise cache storage channel which can successfully extract a 128-bit key from an AES encryption service. ARMageddon [21] uses the Prime+Probe technique to infer activities on the SW and distinguish whether a provided key is valid or not. TruSpy [22] also leverages Prime+Probe to recover a full 128-bit AES encryption key in two different ways. Prime+Count was also employed for enabling cross-world covert channels on TrustZone [23].

I22. Leaking information through branch predictor: The branch predictor can also be leveraged to attack TrustZone. Modern processors include a branch target buffer (BTB) unit, which stores the computed target addresses of taken branch instructions and fetches them when the corresponding branch instructions are predicted [64]. Since the BTB is shared between NW and SW, Prime+Probe can be performed to leak secure information to the NW. The process encompasses priming the BTB by executing many branches, and later let the victim process execute which will evict the attacker BTB entries. When the attacker gets control of execution, the attacker re-executes those branches to detect mispredictions. Given that the internal hardware structure of the BTB works at byte granularity instead of cache-line granularity, this particular attack vector increases considerably the spatial resolution of the probe mechanism. A 256-bit private key has been fully recovered from Qualcomm's hardware-backed keystore [61].

I23. Leaking information using Rowhammer: Rowhammer is a software-induced hardware fault that affects DRAM memories and enables an attacker to flip bits in physical memory by solely performing memory read operations [65, 66]. This type of attack has been explored to subvert TrustZone [62]. A malicious Linux kernel module is used to generate faults to a specific NW target address using Rowhammer, while a secure signature service running on a Trusty TEE instance uses the secure private RSA key to sign a specific message. If the private key is allocated in a secure memory region adjacent to the secure/non-secure memory boundary, the Rowhammer generated by high-rate memory read operations on the non-secure memory border induces faults on the secure one, corrupting the private keys and generating a faulty RSA signature. After retrieving a faulty generated signature on the Linux side, it is possible to deduce the private key. Among the discussed microarchitectural issues, this attack is harder to conduct because it generally requires a higher degree of control over the environment; plus, it is relatively easy to mitigate it.

VII. Defenses for TrustZone-assisted TEEs

This section presents a compilation of defense techniques that can help overcome the architectural, implementation,

		Architectural Issues					Implementation Issues			Hardware Issues		
		Att. Surf.	*Wor. Iso.*	*Mem. Pro.*	*Tru. Boot.*	*Val. Bugs*	*Fun. Bugs*	*Ext. Bugs*	*Arch. Imp.*	*Micro. S.D.*		
2014	TLR [67]	○	○	○	●	●	○	○	○	○		
2015	TrustICE [68]	●	○	○	○	○	○	○	○	○		
	SeCReT [69]	○	●	○	○	○	○	○	○	○		
2016	OSP [70]	●	○	○	○	○	○	○	○	○		
	CaSE [71]	○	○	●	○	○	○	○	○	○		
	R. Guanciale et al. [20], ARMageddon [21], Truspy [22]	○	○	○	○	○	○	○	○	●		
2017	BOOMERANG [72]	○	●	○	○	○	○	○	○	○		
	Komodo [73]	○	○	○	●	●	●	●	○	○		
	MIPE [74]	○	○	○	○	●	●	●	○	○		
	vTZ [3]	●	○	○	○	○	○	○	○	○		
	CLKSCREW [63], Jacob et al. [24], Benhani et al. [25]	○	○	○	○	○	○	○	●	○		
2018	TFence [75]	○	●	○	○	○	○	○	○	○		
	PrivateZone [76]	●	○	○	○	○	○	○	○	○		
	RustZone [77]	○	○	○	○	●	○	●	○	○		
2019	TEEv [78]	●	●	○	○	○	○	○	○	○		
	PrOS [79]	●	○	○	○	○	○	○	○	○		
	SANCTUARY [80]	●	●	○	○	○	○	○	○	●		
	Ginseng [81]	○	○	●	○	●	○	●	○	○		
	K. Ryan [61]	○	○	○	○	○	○	○	○	●		

Table VIII

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

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

A. Architectural Defenses

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

D01. Multi-isolated environments: This technique can be employed to reduce the excessively large attack surface of commercial TEE systems (see I01, I02, and I03). Multiple isolated environments (other than the standard TA sandboxes in SW) help to reduce exposure of TEE systems to attacks by (a) increasing the isolation granularity between TEE components, thus containing the extent of potential damage caused by a security breach, and/or (b) limiting the amount of code that runs in the SW, thereby reducing the chances of highly damaging SW privilege escalation attacks. Different variants have been proposed. One line of work aims at creating strongly isolated compartments within the NW itself where TAs would be allocated. To protect TAs, TrustICE [68] and SANCTUARY [80] leverage different features of the TZASC. OSP [70], PrivateZone [76], and vTZ [3] instead, explore the hardware virtualization extensions available in NW (NS-EL2) to implement isolated environments. A second line of research retains TAs within the SW but aims to strengthen the isolation between them, e.g., TEEv [78] and PrOS [79] implement a minimalist hypervisor in SW, allowing TAs to run on multiple isolated secure guest OSes. Due to the current lack of hardware virtualization support in SW, both systems use same-privilege isolation to secure the hypervisor from secure guest OSes.

D02. Secure cross-world channels: Isolation between worlds can be threatened by vulnerabilities in SW triggered from the NW. In particular, the reported TEE deficiencies that can undermine this isolation (see I04 and I05) may lead to the extraction of sensitive data from SW. Although these specific issues can be addressed by fixing vulnerable TEE kernel system calls, cross-world isolation can further be strengthened by secure NW-SW channels. Proposed by different authors, these mechanisms help to overcome two existing limitations in mainstream TEEs: (1) absent or weak authentication when accessing TEE resources from NW and (2) potentially insecure shared-memory for data exchange within the channel. SeCReT [69] provides a session key (to REE applications) that can be utilized to encrypt the messages. To protect the session key from untrusted REE kernel, SeCReT interposes mode switches from/to the kernel and removes the key from memory during kernel mode execution. TFence [75] further removes this kernel dependency by creating a partially privileged process – a shielded portion of the REE application process – which can directly communicate with TEE. Both TEEv [78] and SANCTUARY [80] implement exclusive shared memory, and PrivateZone [76] enables communication without sharing memory, i.e. through data copies. Aravind et al. [72] use pointer sanitization for preventing boomerang attacks.

D03. Encrypted memory: Existing deficiencies in TEE memory protection (I06 and I07) can mostly be addressed with mechanisms from mainstream OSes (e.g., ASLR, stack cookies). Nevertheless, commercial TEEs can provide stronger security defenses, e.g., against cold boot attacks, by implementing encrypted memory capability. In contrast to Intel SGX, TrustZone does not provide built-in support for on-chip memory encryption. To bridge this gap, CaSE [71] allows TAs to run entirely from the cache and ensures that their state is encrypted while written back to main memory. Along the same vein, Ginseng [81] protects variables tagged by the application programmer as "sensitive", by allocating them on CPU registers and encrypting them at runtime before saving them in memory.

D04. Trusted computing primitives: Commercial TEEs rely on secure boot to guarantee the integrity of the TEE image. However, this mechanism, *per se*, is insufficient to enable a TA's client – local or remote – to verify the integrity and identity of both TEE and TA binaries (see I08, I09). To overcome this limitation, commercial TEEs can implement additional trusted computing primitives that help provide such guarantees, namely remote attestation and sealed storage. For instance, TLR [67] includes a sealed storage primitive that allows for protecting data cryptographically and bind it to specific hash values of the TEE/TA software stack. Komodo [73] demonstrates how to implement, for TrustZone-assisted TEEs, the security protocols of sealed storage and remote attestation as originally specified for enclaves (i.e., Intel SGX's secure environments for TAs). There is also a body of work in trusted I/O path primitives [82, 83] which aims at providing secure access to peripherals. Given that we identified a relatively small number of vulnerabilities involving access to peripherals, which can be mitigated using standard hardware features for I/O mediation (e.g., SMMU, bus bridges), Table VIII omits such references.

B. Implementation Defenses

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

D05. Managed code runtimes: Commercial TEE systems are mostly written in the C programming language which allows for compiling highly efficient code but do not provide memory safety. However, many validation bugs are caused by memory violation errors introduced by the programmer. In alternative TEE systems, such as in TLR [67], TAs are not compiled to native code, but rather to .Net managed code which is then interpreted by a small-sized managed code runtime (akin to a JVM). At the expense of some performance overhead, the managed runtime helps to prevent validation bugs, e.g., by implementing run-time memory checks and garbage collection.

D06. Type-safe programming languages: Researchers have explored the idea of using type-safe programming languages to implement specific components of TrustZone-assisted TEE software. Notably, RustZone [77] is an extension for OP-TEE where TAs are implemented in the Rust programming language. Given that Rust provides memory and thread-safety, RustZone can help prevent validation bugs and some concurrency bugs responsible for crippling TA software (see I11). The Rust programming language has also been used in Ginseng [81] for implementing a large part of the software that runs in monitor mode, i.e., the GService (see I10).

D07. Software verification: Implementation bugs tend to exist due to a mismatch between the expected requirements of a piece of software and its actual implementation. Software verification, which comprises techniques such as model checking, symbolic execution, and formal methods, aims at preventing this mismatch by ensuring that the implementation fully satisfies all envisioned requirements. For this reason, it has the potential to help prevent all three classes of prevalent TEE implementation bugs. However, these techniques can be challenging to apply in practice, not only because they require considerable effort and skill, but also because they are difficult to scale for complex programs. Despite these obstacles, important advances have been achieved with the formal verification of specific TEE components, e.g., a small TEE monitor named Komodo [73], which implements the specification of Intel SGX enclaves, and a memory manager called MIPE [74].

C. Hardware Defenses

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

D08. Architectural countermeasures: Hardware manufacturers tend to increasingly pack more components into the SoC chips, becoming very difficult for TEE designers to fully understand its implications to the security of a TEE system. To prevent a growing abuse of reconfigurable hardware technology (see I19), researchers have proposed: (1) the inclusion of a small hardware wrapper into all IP cores endowed with an AXI interface so as to restrict their operation during system boot [24]; (2) the implementation of a dedicated AXI interconnect for secure devices [25]; and (3) the inclusion of a non-secure only port to connect all non-sensitive memory-mapped IP cores and restrict its operation through memory protection mechanisms (e.g. SMMU) [25]. To prevent misuse of hardware voltage regulators (see I20), a possible approach is to place specific operation limits into the software (i.e., drivers) or into the hardware itself [63].

D09. Microarchitectural countermeasures: One way to prevent cache side-channels (see I21) is through careful implementation of cryptographic algorithms in software [20–22, 61] or using dedicated hardware (e.g., specific ISA instructions such as AESD and AESE in Armv8-A) [21] to prevent information leakage in cryptographic-related operations. Another path is to leverage cache maintenance techniques to prevent information leakage through caches. For TrustZone-assisted TEEs that do not use shared L2 cache, one approach is to flush the L1 cache on every SW exit [80]. If shared L2 cache is used, although cache flushing (total or selective) or cache normalization operations performed at every SW entry and

		Dedicated RAM	Cross-World Isol.	Encryp. Mem.	Protection Ring	Attestation	Previously Exploited	Communication w/ REE
CPU Extensions	Arm TrustZone [1]	○	MMU + HW	○	-2	sec. boot.	●	sh. mem.
	Intel SGX [84]	○	MMU + HW	●	1	remote att.	●	data copy
	Intel SMM [85]	○	MMU	○	-2	sec. boot.	●	sh. mem.
	Sanctum [86]	○	MMU + HW	○	-2	sec. boot.	○	data copy
Co-Processors	Apple SEP [87]	●	Phys. + HW	●	-3	sec. boot.	○	sh. mem.
	Qualcomm SPU [88]	●	Phys. + HW	●	-3	sec. boot.	○	sh. mem.
Chips	Intel ME [89]	●	Phys.	●	-3	sec. boot.	●	sh. mem. + HECI
	Google Titan-M [90]	●	Phys.	○	-3	sec. boot.	○	SPI/USB/I2C
	TPM [91]	○	Phys.	○	-3	sec. boot.	●	SPI/I2C/LPC
Virtualization	Windows VSM [92]	○	MMU	○	-1	sec. boot.	●	sh. mem.
	AMD SEV [93]	○	MMU	●	-1	remote att.	●	sh. mem.
RISC-V	Multizone [94]	○	PMP	○	-2	sec. boot.	○	data copy
	Keystone [95]	○	PMP	○	-2	remote att.	○	sh. mem.

Table IX

Dedicated RAM: used for allocation of security-sensitive state and isolation from potentially insecure main RAM. *Cross-world isolation:* implemented using memory management components (MMU / PMP) or in combination with HW-specific features (e.g., TrustZone's TZASC); dedicated off-SoC chips achieve isolation through physical separation. *Encrypted memory:* filled circle indicates that hardware-enforced memory encryption is supported. *Protection Ring:* classified in five levels [26], i.e., 1 (user), 0 (kernel), -1 (hypervisor), -2 (monitor), -3 (off-chip). *Attestation:* if the TEE runtime can perform local attestation only (i.e. secure boot), or remote attestation also. *Previously exploited:* black circle indicates publicly known exploits to TEE systems enabled by that particular technology. *Communication mechanisms with REE:* shared memory, data copying, and communication bus (e.g. USB or SPI).

exit may be sufficient to prevent cache-storage attacks [20], L1 flushing may not be able to prevent Prime+Probe attacks in multicore systems [21]. In this case (which also holds for all aforementioned cases), cache partitioning can prevent an attacker from leveraging contention with victim [21, 22, 80]. Carefully implemented cryptographic algorithms seem also to be effective at preventing breaches through the BTB (see `I22`). This was shown and highlighted by Keegan et al. [61], where different versions of an algorithm were able to render side-channels ineffective. To prevent Rowhammer attacks (see `I23`), TEEs must avoid the use of memory at the NW-SW boundary.

VIII. Beyond TrustZone-assisted TEEs

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

One class of hardware technologies provides a set of CPU extensions where the processor is augmented with circuitry that implements specific TEE-enabling security functionality. TrustZone fits this category as well as technologies such as Intel Software Guard Extensions (SGX) [84], Intel System Management Mode (SMM) [85], and Sanctum [86], for instance. Separate co-processors in the SoC, such as Apple Secure Enclave Processor (SEP) [87] or Qualcomm Secure Processing Unit (SPU) [88], may include dedicated non-volatile storage and RAM which allows for reducing shared hardware resources and help prevent side-channel attacks [21, 96]. In dedicated security chips, the runtime environment comprises a processor, memory, and persistent storage. For instance, Intel Management Engine (ME) [89] is a firmware based on Minix OS that runs on a separate processor in Intel systems. It is designed to be an almost fully independent system, with access to many peripherals and its own secure boot functionality. Some security chips may be equipped with tamper detection, as in the case of the Titan-M [90]. Others, such as Trusted Platform Module (TPM) [91], implement specific functions for trusted boot, remote attestation, and other primitives. Hardware support for virtualization can also be used for implementing TEEs. In Windows' Virtual Secure Mode (VSM) [92] the hypervisor establishes two hierarchical privileges modes VTL0 (analogous to the normal world) and VTL1 (analogous to secure world). AMD Secure Encrypted Virtualization (SEV) [93] provides the ability to encrypt virtual machine memory using hardware-accelerated memory encryption. Lastly, RISC-V is an instruction set architecture which, although not widely deployed yet, can also be used for implementing TEEs [94, 95].

IX. Conclusion

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

Acknowledgments: We thank our shepherd David Kohlbrenner and the anonymous reviewers for their comments and suggestions. We are grateful to Joakim Bech for the insightful discussions about OP-TEE. This work was supported by national funds through Universidade do Minho, Instituto Superior Técnico / Universidade de Lisboa, and FCT via projects UID/CEC/00319/2019 and UID/CEC/50021/2019. David Cerdeira was supported by FCT grant SFRH/BD/146231/2019.

References

[1] Arm, "ARM Security Technology. Building a Secure System using TrustZone Technology ARM," *Arm white paper*, p. 108, 2009.

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

[3] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in *USENIX Security Symposium*. USENIX Association, 2017, pp. 541–556.

[4] N. Asokan, T. Nyman, N. Rattanavipanon, A. Sadeghi, and G. Tsudik, "ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2290–2300, Nov 2018.

[5] P. Sparks, "The route to a trillion devices," White Paper, ARM, 2017.

[6] Symantec, "iOS malware, XcodeGhost, infects millions of Apple Store customers," 2018, accessed: December 10th 2018. [Online]. Available: https://us.norton.com/internetsecurity-emerging-threats-ios-malware-xcodeghost-infects-millions-of-apple-store-customers.html

[7] Dan Goodin, "Found: New Android malware with never-before-seen spying capabilities," 2018, accessed: December 10th 2018. [Online]. Available: https://arstechnica.com/information-technology/2018/01/found-new-android-malware-with-never-before-seen-spying-capabilities/

[8] ——, "Malware found preinstalled on 38 Android phones used by 2 companies," 2017, accessed: December 10th 2018. [Online]. Available: https://arstechnica.com/information-technology/2017/03/preinstalled-malware-targets-android-users-of-two-companies/

[9] Swati Khandelwal, "New Android Malware Framework Turns Apps Into Powerful Spyware," 2018, accessed: December 10th 2018. [Online]. Available: https://thehackernews.com/2018/08/android-malware-spyware.html

[10] X. Jiang and Y. Zhou, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 95–109.

[11] Android, "Fingerprint HAL," 2018, accessed: December 18th 2018. [Online]. Available: https://source.android.com/security/authentication/fingerprint-hal.html

[12] Trustonic, "Trustonic application protection delivers comprehensive security for mobile financial services," 2018, accessed: December 18th 2018. [Online]. Available: https://www.trustonic.com/markets/financial-services/

[13] Michael Lu, "TrustZone, TEE and Trusted Video Path Implementation Considerations," 2018, accessed: December 18th 2018. [Online]. Available: https://www.arm.com/files/event/Developer_Track_6_TrustZone_TEEs_and_Trusted_Video_Path_implementation_considerations.pdf

[14] Gal Beniamini, "TrustZone Kernel Privilege Escalation (CVE-2016-2431)," 2016, accessed: April 25th 2019. [Online]. Available: http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html

[15] ——, "War of the Worlds - Hijacking the Linux Kernel from QSEE," 2016, accessed: April 25th 2019. [Online]. Available: https://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html

[16] Daniel Komaromy, "Unbox Your Phone," 2018, accessed: April 25th 2019. [Online]. Available: https://medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c

[17] Gal Beniamini, "Trust Issues: Exploiting TrustZone TEEs," 2016, accessed: April 25th 2019. [Online]. Available: https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html

[18] Di Shen, "Attacking your "Trusted Core" Exploiting TrustZone on Android," BlackHat USA, https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android.pdf, 2015, accessed: April 25th 2019.

[19] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou, "CacheKit: Evading Memory Introspection Using Cache Incoherence," in *IEEE European Symposium on Security and Privacy*, March 2016, pp. 337–352.

[20] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures," in *IEEE Symposium on Security and Privacy*, May 2016, pp. 38–55.

[21] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Conference on Security Symposium*. USENIX Association, 2016, pp. 549–564.

[22] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices," *IACR Cryptology ePrint Archive*, vol. 2016, p. 980, 2016. [Online]. Available: http://dblp.uni-trier.de/db/journals/iacr/iacr2016.html{#}ZhangSSLH16

[23] H. Cho, P. Zhang, D. Kim, J. Park, C.-H. Lee, Z. Zhao, A. Doupé, and G.-J. Ahn, "Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: ACM, 2018, pp. 441–452.

[24] N. Jacob, J. Heyszl, A. Zankl, C. Rolfes, and G. Sigl, *How to Break Secure Boot on FPGA SoCs Through Malicious Hardware"*. Cham: Springer International Publishing, 2017, pp. 425–442.

[25] E. M. Benhani, C. Marchand, A. Aubert, and L. Bossuet, "On the security evaluation of the ARM TrustZone extension in a heterogeneous SoC," in *IEEE International System-on-Chip Conference*, Sept 2017, pp. 108–113.

[26] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 130:1–130:36, Jan. 2019.

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

[28] S. Pinto, H. Araújo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on TrustZone-enabled Microcontrollers? Voilà!" in *25th IEEE IEEE Real-Time and Embedded Technology and Applications Symposium, Montreal, Canada*, 2019.

[29] Gal Beniamini, "Full TrustZone exploit for MSM8974," 2015, accessed: April 25th 2019. [Online]. Available: http://bits-please.blogspot.com/2015/08/full-trustzone-exploit-for-msm8974.html

[30] ——, "Exploring Qualcomm's TrustZone implementation," 2015, accessed: April 25th 2019. [Online]. Available: http://bits-please.blogspot.com/2015/08/exploring-qualcomms-trustzone.html

[31] ——, "Android linux kernel privilege escalation vulnerability and exploit (CVE-2014-4322)," 2015, accessed: April 25th 2019. [Online]. Available: http://bits-please.blogspot.com/2015/08/android-linux-kernel-privilege.html

[32] ——, "Android privilege escalation to mediaserver from zero permissions (CVE-2014-7920 + CVE-2014-7921)," 2016, accessed: April 25th 2019. [Online]. Available: http://bits-please.blogspot.com/2016/01/android-privilege-escalation-to.html

[33] ——, "QSEE privilege escalation vulnerability and exploit (CVE-2015-6639)," 2016, accessed: April 25th 2019. [Online]. Available: http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html

[34] ——, "Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption (Jun 2016)," 2016, accessed: April 25th 2019. [Online]. Available: https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html

[35] ——, "Unlocking the Motorola Bootloader," 2016, accessed: April 25th 2019. [Online]. Available: http://bits-please.blogspot.com/2016/02/unlocking-motorola-bootloader.html

[36] D. Rosenberg, "QSEE Trustzone Kernel Integer Overflow Vulnerability," in *Black Hat conference*, 2014.

[37] Sean Beaupre, "TRUSTNONE," 2015, accessed: April 25th 2019. [Online]. Available: http://theroot.ninja/disclosures/TRUSTNONE_1.0-11282015.pdf

[38] Josh Thomas, "A story of Research for PacSec 2014," PacSec, https://pacsec.jp/psj14/PSJ2014_Josh_PacSec2014-v1.pdf, 2014, accessed: April 25th 2019.

[39] David Berard, "Kinibi TEE: Trusted Application Exploitation," 2018, accessed: April 25th 2019. [Online]. Available: https://www.synacktiv.com/posts/exploit/kinibi-tee-trusted-application-exploitation.html

[40] HC Ma, "How Samsung Secures Your Wallet and How To Break It," BlackHat USA, https://www.blackhat.com/docs/eu-17/materials/eu-17-Ma-How-Samsung-Secures-Your-Wallet-And-How-To-Break-It.pdf, 2017, accessed: April 25th 2019.

[41] Nick Stephens, "Behind the PWN of a TrustZone," 2016, accessed: April 25th 2019. [Online]. Available: https://www.slideshare.net/GeekPwnKeen/nick-stephenshow-does-someone-unlock-your-phone-with-nose

[42] "SierraTEE," accessed: December 28th 2018. [Online]. Available: https://www.sierraware.com/open-source-ARM-TrustZone.html

[43] "Common Vulnerabilities and Exposures," accessed: September 3rd 2018. [Online]. Available: https://cve.mitre.org/about/

[44] "Qualcomm Product Security - Security Advisories," accessed: September 3rd 2018. [Online]. Available: https://www.qualcomm.com/company/product-security/security-advisories

[45] "Code Aurora Forum Security Bulletin," accessed: September 3rd 2018. [Online]. Available: https://www.codeaurora.org/security-bulletin

[46] "Nvidia Product Security," accessed: December 28th 2018. [Online]. Available: https://www.nvidia.com/en-us/security/

[47] "Huawei Security Advisories," accessed: September 3rd 2018. [Online]. Available: https://www.huawei.com/en/psirt/all-bulletins

[48] "Samsung Mobile Security - Android Security Updates," accessed: September 3rd 2018. [Online]. Available: https://security.samsungmobile.com/securityUpdate.smsb

[49] "PS Vita," accessed: December 28th 2018. [Online]. Available: http://www.vitahacker.com/

[50] "Project Zero," accessed: December 28th 2018. [Online]. Available: https://googleprojectzero.blogspot.com/

[51] F. Basse, "Amlogic S905 SoC: bypassing the (not so) Secure Boot to dump the BootROM," https://fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html, 2016, accessed: October 30th 2018.

[52] G. Delugré and I. Arce, "Vulnerabilities in High Assurance Boot of NXP i.MX microprocessors," https://blog.quarkslab.com/vulnerabilities-in-high-assurance-boot-of-nxp-imx-microprocessors.html, 2016, accessed: October 30th 2018.

[53] "Common Vulnerability Scoring System v3.0: Specification Document," accessed: December 28th 2018. [Online]. Available: https://www.first.org/cvss/specification-document

[54] ARM, "Trusted Firmware-A - version 2.0," 2017, accessed: December 19th 2018. [Online]. Available: https://github.com/ARM-software/arm-trusted-firmware

[55] D. Wheeler, "SLOCCount," accessed: September 15th 2019. [Online]. Available: https://dwheeler.com/sloccount

[56] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments," *Proceedings 2017 Network and Distributed System Security Sympo-*

sium, no. March, 2017.

[57] ARM, "ARM Trusted Firmware Security Advisory TFV 3," 2017, accessed: December 17th 2018. [Online]. Available: https://github.com/ARM-software/arm-trusted-firmware/wiki/ARM-Trusted-Firmware-Security-Advisory-TFV-3

[58] J. Wiklander, "arm: sm: [bugfix] save/restore fiq core registers," 2016, accessed: December 17th 2018. [Online]. Available: https://github.com/OP-TEE/optee_os/commit/f2dec49b7aeacd4cf36cacecc2a62ca925800e7a

[59] J. Forissier, "Core: pager: ltc: prng: add entropy to the AE key for paged TAs," https://github.com/OP-TEE/optee_os/commit/93d3c451da7014193220c3f686c4b6379a1c5095, 2017, accessed: November 26th 2018.

[60] ——, "storage: protect TA directory with a mutex," https://github.com/OP-TEE/optee_os/commit/b81882b289e70aa571b387f2b54aea853d74b31e, 2016, accessed: November 28th 2018.

[61] Keegan Ryan, "Hardware-BackedHeist: Extracting ECDSA Keys from Qualcomm's TrustZone," 2019, accessed: April 27th 2019. [Online]. Available: https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2019/hardwarebackedhesit.pdf

[62] P. Carru, "Attack ARM TrustZone using Rowhammer," in *GreHack*, 2017. [Online]. Available: https://grehack.fr/data/2017/slides/GreHack17_Attack_TrustZone_with_Rowhammer.pdf

[63] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *USENIX Security Symposium*. USENIX Association, 2017, pp. 1057–1074.

[64] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 557–574.

[65] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 361–372.

[66] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2016, pp. 1675–1689.

[67] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications," *SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 67–80, Feb. 2014.

[68] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 367–378.

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

[70] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 565–578.

[71] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "CaSE: Cache-Assisted Secure Execution on ARM Processors," in *IEEE Symposium on Security and Privacy*, May 2016, pp. 72–90.

[72] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments," in *Network and Distributed System Security Symposium*, 2017.

[73] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using Verification to Disentangle Secure-enclave Hardware from Software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 287–305.

[74] R. Chang, L. Jiang, W. Chen, Y. Xiang, Y. Cheng, and A. Alelaiwi, "Mipe: a practical memory integrity protection method in a trusted execution environment," *Cluster Computing*, pp. 1–13, 2017.

[75] J. Jang and B. B. Kang, "Retrofitting the Partially Privileged Mode for TEE Communication Channel Protection," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2018.

[76] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang, "PrivateZone: Providing a Private Execution Environment Using ARM TrustZone," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 797–810, Sep. 2018.

[77] E. Evenchick, "RustZone: Writing Trusted Applications in Rust," 2018, accessed: April 21st 2019. [Online]. Available: https://i.blackhat.com/eu-18/Thu-Dec-6/eu-18-Evenchick-RustZone.pdf

[78] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang, "TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019. New York, NY, USA: ACM, 2019, pp. 2–16.

[79] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek, "PrOS: Light-weight Privatized Secure OSes in ARM TrustZone," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2019.

[80] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with User-

space Enclaves," in *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[81] M. H. Yun and L. Zhong, "Ginseng: Keeping Secrets in Registers When You Distrust the Operating System," in *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[82] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee, "SeCloak: ARM Trustzone-based Mobile Peripheral Control," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '18. New York, NY, USA: ACM, 2018, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/3210240.3210334

[83] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C. Chu, and T. Li, "Building Trusted Path on Untrusted Device Drivers for Mobile Devices," in *Asia-Pacific Workshop on Systems*. ACM, 2014, pp. 8:1–8:7.

[84] Intel, "Intel Software Guard Extensions," 2019, accessed: September 12st 2019. [Online]. Available: https://software.intel.com/en-us/sgx/

[85] R. R. Collins, "Intel's System Management Mode," 1997, accessed: September 12st 2019. [Online]. Available: http://www.rcollins.org/ddj/Jan97/Jan97.html

[86] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation," in *USENIX Security Symposium*. USENIX Association, 2016, pp. 857–874.

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

[88] Qualcomm, "Qualcomm Secure Processing Unit SPU230 Core Security Target Lite," 2019, accessed: September 12st 2019. [Online]. Available: https://www.commoncriteriaportal.org/files/epfiles/1045b_pdf.pdf

[89] J. O., "Getting Started with Intel Active Management Technology (Intel AMT)," 2019, accessed: September 12st 2019. [Online]. Available: https://software.intel.com/en-us/articles/getting-started-with-intel-active-management-technology-amt

[90] X. Xin, "Titan M makes Pixel 3 our most secure phone yet," 2018, accessed: September 12st 2019. [Online]. Available: https://www.blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/

[91] T. C. Group, "Trusted Platform Module (TPM)," 2018, accessed: September 12st 2019. [Online]. Available: https://trustedcomputinggroup.org/work-groups/trusted-platform-module/

[92] Windows, "Virtualization-based Security (VBS)," 2017, accessed: September 12st 2019. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs

[93] AMD, "AMD Secure Encrypted Virtualization (SEV)," 2019, accessed: September 12st 2019. [Online]. Available: https://developer.amd.com/sev/

[94] Hex-Five, "Hex Five Security Adds MultiZone™ Trusted Execution Environment to the SiFive Software Ecosystem," 2018, accessed: September 12st 2019. [Online]. Available: https://hex-five.com/2018/08/22/hex-five-adds-multizone-security-to-sifive-software-ecosystem/

[95] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović, "Keystone: A framework for architecting tees," *arXiv preprint arXiv:1907.10119*, 2019.

[96] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *27th USENIX Security Symposium*. USENIX Association, 2018, p. 991–1008.