Computers & Security 129 (2023) 103180

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

A survey on the (in)security of trusted execution environments

Antonio Muñoz*, Ruben Ríos, Rodrigo Román, Javier López

Network, Information and Computer Security (NICS) Lab, University of Malaga, Spain

ARTICLE INFO

Article history:
Received 16 November 2022
Revised 4 February 2023
Accepted 9 March 2023
Available online 14 March 2023

Keywords:
Computer security
Secure hardware
Trusted execution environments
Hardware attacks
Software attacks
Side-channel attacks

ABSTRACT

As the number of security and privacy attacks continue to grow around the world, there is an ever increasing need to protect our personal devices. As a matter of fact, more and more manufactures are relying on Trusted Execution Environments (TEEs) to shield their devices. In particular, ARM TrustZone (TZ) is being widely used in numerous embedded devices, especially smartphones, and this technology is the basis for secure solutions both in industry and academia. However, as shown in this paper, TEE is not bullet-proof and it has been successfully attacked numerous times and in very different ways. To raise awareness among potential stakeholders interested in this technology, this paper provides an extensive analysis and categorization of existing vulnerabilities in TEEs and highlights the design flaws that led to them. The presented vulnerabilities, which are not only extracted from existing literature but also from publicly available exploits and databases, are accompanied by some effective countermeasures to reduce the likelihood of new attacks. The paper ends with some appealing challenges and open issues.

1. Introduction

Nowadays, a wide range of mechanisms are emerging to mitigate current and future security threats associated with the development of an ever increasing number of heterogeneous computing devices. Computing platforms are continuously evolving, running sophisticated operating systems and hosting countless applications from possibly untrustworthy vendors. In these highly complex environments, the risk of a security breach is extremely high and hence the need for execution environments capable of isolating security-sensitive applications. The inclusion of secure execution environments enables them hosting a wide variety of applications and protecting the integrity of their own internal state.

Among these mechanisms, a relevant choice is the use of Trusted Execution Environments (TEE), which are hardware-isolated areas in microprocessors that enable the secure execution of applications thereby assuring the confidentiality and integrity of data and code. In fact, in the definition of the TEE standard (Ekberg et al., 2012) it appears as an isolated environment that coexists and cooperates with the operating system. The main purpose of this isolation is to provide security to the whole system. TEE technology is certainly a trend in modern platforms, due in part to the adoption of smartphones as our primary platform of interaction with other devices.

ARM's TrustZone design stands out among the various system-on-chip (SoC) isolation solutions. TrustZone (TZ) is the collection of hardware mechanisms that enable TEEs to implement the required isolation from the main operating environment. TEEs have been considered as secure elements and as such have been used for protecting sensitive applications in a number of verticals, such as cyber-physical systems (CPS) (Pinto et al., 2017) or embedded systems (Janjua et al., 2019). Nevertheless, some recently found vulnerabilities and attacks on different TEE implementations, should make us re-examine existing assumptions on the security provisions of TEEs.

There are various works, such as (Komaromy, 2018; Lipp et al., 2016; Machiry et al., 2017; Rosenberg, 2014; Tang et al., 2017), that provide a nice perspective on the situation of security in TEE. In addition, other works provide additional analyses on this subject. For example, Sabt et al. (2015) describe the fundamental properties of TEE and provide a comparative study of different TEEs based on ARM TZ, but this work does not analyze their impact nor discuss the main reasons that may lead to attacks. Other examples, such as Arfaoui et al. (2014), provide a perspective according to GlobalPlatform (GlobalPlatform) standards, in terms of security, with various TEE technologies, and Asokan et al. (2014) present a comprehensive review of the current role of trusted computing technology in the field of mobile devices.

* Corresponding author.
E-mail addresses: amunoz@lcc.uma.es (A. Muñoz), ruben@lcc.uma.es (R. Ríos), roman@lcc.uma.es (R. Román), jlm@lcc.uma.es (J. López).

https://doi.org/10.1016/j.cose.2023.103180

Our approach differs from the aforementioned papers in the sense that our study focuses on classifying existing vulnerabilities and identifying their impact on the different TZ-based TEE implementations. For this purpose, various devices in the market have been taken as a reference. Note that there have been other papers that analyze such issues, but only partially. For example, Santos et al. (2014) provide a taxonomy of vulnerabilities in commercial TEE, but without delving into the particularities of the attacks. Another example is Cerdeira et al. (2020), which provide an analysis of the security vulnerabilities found, until then, in those commercial TEE implementations based on TrustZone. Their paper was limited to the analysis of Qualcomm[1], Trustonic, Huawei, Nvidia (Corporation, 2015) and Linaro OP-TEE Brand TEE systems. Finally, other works, such as Busch et al. (2020) and Meng et al. (2018) also provide a thorough critical review, although limited to Huawei's TEE and Android vulnerabilities, respectively.

This paper includes an exhaustive analysis of the security limitations and associated countermeasures of TrustZone-based TEEs. More specifically, the main contributions of this paper are as follows:

1. An extensive review and analysis of the state of the art of TZ security extensions, including TEE implementations and their features.
2. A comprehensive categorization of existing vulnerabilities and attacks against TEE implementations.
3. A detailed analysis of existing countermeasures for the described attacks and vulnerabilities.
4. A discussion on open challenges and recommendations for future implementations of secure TEEs.

The rest of the paper is organized as follows: Section 2 provides a relevant background on TEE including the evolution of the standardization, a description of its main capabilities and applications, and some implementation details. Section 3 presents a novel taxonomy of TEE attacks that will guide the exposition throughout the rest of the paper. Software-based attacks are detailed in Section 4, architectural attacks in Section 5. Side-channel attacks are analyzed separately in Section 6 and micro-architectural attacks in 7. In Section 8 a series of existing countermeasures are compiled and analyzed. Finally, open challenges are discussed in Section 9, and conclusions and future works are presented in Section 10.

2. Background

2.1. The evolution of trusted execution environments

Software security mechanisms are not sufficient to counter advanced attacks in many real-world situations. In such cases, building secure solutions requires the involvement of secure hardware elements. Doubtlessly, the need for secure elements boosted the development of the TPM (*Trusted Platform Module*), whose first version dates from 2003 and was followed by TPM 2.0 (TCG, 2013), which appeared several years later, in 2012. However, both of these standards have been considered unsuitable for mobile computing devices for various reasons, such as limitations derived from the use of batteries, the computational restrictions imposed by mobile devices or the increased price implied by the integration of a TPM chip, which in some cases can represent a high percentage of the device's hardware budget. In this line, the *Trusted Computing Group (TCG)* (TCG, 2013) defined in 2007 the specifications of the Mobile Trusted Module (MTM) (Ekberg et al., 2007), which appears as an branch of TPM v1.2 with changes to adapt it to mobile platforms. Nevertheless, as a consequence of the physical resource limitation of mobile devices, but MTM implementation was never widely adopted. Later TPM Mobile (McGill, 2013) was proposed as an attempt to adapt the TPM 2.0 specification to mobile devices. Although that specification was designed to cover implementation on a wide range of mobile devices, TPM Mobile was only implemented in a small number of devices due to the lack of trust in a software-based solution. There have been alternative implementations of a mobile TPM, such as simTPM (Chakraborty et al., 2019), which relies on the SIM card available in mobile platforms to avoid most of mobile TPM and MTM issues without the need for additional hardware. Notwithstanding, the main disadvantage with this solution was that the SIMs were not tamper-proof resistant, unlike the TPM chip, and therefore cannot be considered as a reliable secure element.

As a consequence of these issues, *GlobalPlatform*[2], a non-profit association, defined specifications for secure chip technologies, gathering the fundamental security requirements of mobile devices and describing the ideal security guard for mobile devices. This specification, known as Trusted Execution Environment (TEE), quickly gained traction on the market – to the point that a number of companies that were initially reluctant to the initiative finally joined. TEE architecture proposed by GlobalPlatform highlighting the separation of worlds[3] as the most relevant design novelty. Nokia and Trusted Logic were the first in the long list of companies that joined, followed by other companies such as ARM, NVIDIA (Corporation, 2015), AMD, ST, Qualcom, Ericsson and Samsung, which are now fully involved in the development of the TEE specifications. As of today, TEE is a well-defined security element, whose technical specifications not only define the architecture but also the services available for the applications running on top of it[4]. GlobalPlatform initially focused on TEE standardization (System Architecture specifications and client API interface). Later, GlobalPlatform released a specification for the Secure OS, including the internal API and TEE applications.

The main goal of the TEE is to guarantee the *secure execution* of programs[5] For this purpose, TEE isolation capability enables a secure area for handling sensitive data, thus eliminating the need to trust the software running in the device. In particular, ARM TrustZone (Pinto and Santos, 2019), which is the most extended trusted hardware TEE systems rely on, defines two protection domains or realms: the Secure World (SW) and the Normal World (NW).

2.2. TEE capabilities and applications

The TEE design enables to implement security-sensitive services by taking advantage of its assurance and secure storage functionalities necessary to preserve both the confidentiality and integrity of data and code. In current implementations, the decision to deny or allow the installation of a new service in the TEE is made by the TEE developer playing the role of a central authority.

Among the different capabilities offered by the TEE, we highlight the following:

- Isolated execution: This functionality allows the separated execution of applications, some of them in a secure environment and others in a normal environment. It is highly recommended that isolation is achieved by means of hardware mechanisms in order to prevent this mechanism from being controlled from

[1] Qualcomm Product Security. Available: https://www.qualcomm.com/company/productsecurity/securityadvisories.

[2] https://globalplatform.org/.

[3] Some authors refers to realms instead of worlds, both terms are the same concept along this paper.

[4] http://globalplatform.org/specificationsdevice.asp.

[5] Henceforth, we use indistinguishably the terms trustlets and trusted applications (TAs) to software executed in the TEE as secure programs, applications or processes.

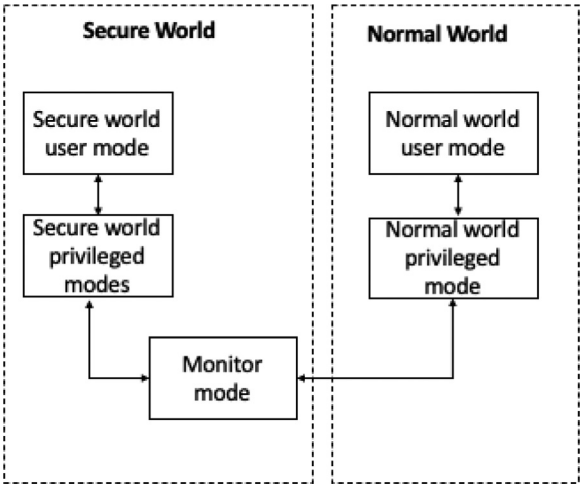

Fig. 1. Relationship between the Secure World and the Normal World.

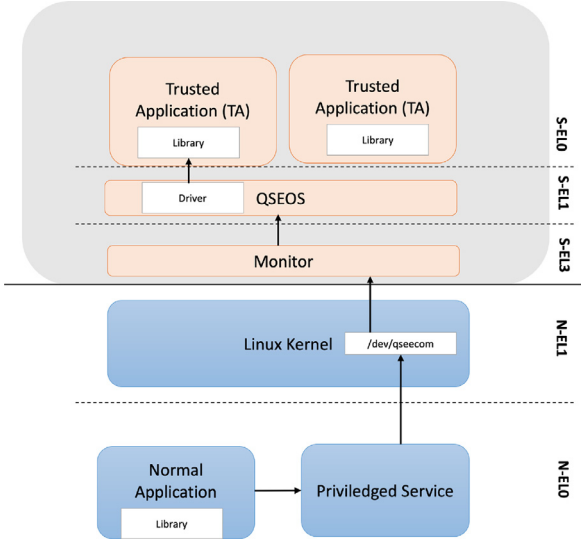

Fig. 2. TEE Worlds in Qualcomm TEE. Communication between wolds is mediated by a priviledged OS daemon by SMC calls.

the non-secure world. Isolated execution can be considered as the primary purpose of a TEE.

- Secure Storage: The TEE provides Trusted Storage of data and keys. Trusted storage is tied to a particular TEE and device. This prevents any attacker from accessing and modifying the stored data unless they have the appropriate permissions.
- Platform Integrity: Secure boot ensures both the integrity and authenticity of the platform. It allows the trusted OS execution environment to be instantiated from a trusted root within the TEE. The process uses assets linked to the TEE and isolated from the normal OS. Besides, according to the TEE description, the TEE is protected against some physical attacks. However, note that attacks breaking the IC package are beyond the scope of TEE protection.

Based on the above core capabilities, existing TEEs, such as TrustZone, can build a large variety of functionalities and applications. Some examples are secure credentials generation (Elenkov, 2013), secure key storage (Android Keystore, dmverity) (Cooijmans et al., 2014), secure boot (Dietrich and Winter, 2009; Ge et al., 2014), kernel integrity verification, (e.g., Samsung's TIMA Azab et al. (2014)), trusted peripherals and sensors (Liu et al., 2012), mobile payments using emulation of secure elements (Pirker and Slamanig, 2012; Pirker et al., 2012), digital content protection systems (Ahmad et al., 2013; Tögl et al., 2013), services to manage and issue online tickets (Hussin et al., 2005; 2006; Tamrakar et al., 2011), cloud storage access authentication mechanisms (Ekberg et al., 2012; Shin et al., 2012), security of IoT devices (González and Bonnet, 2013; Guan et al., 2017), and many more.

2.3. Trusted execution environment & ARM TrustZone architecture

As mentioned above, ARM TrustZone is a particular implementation of TEE that enables the isolation of CPU state, memory, I/O data, etc. It is built around the concept of protection domains, namely the SW and NW, as aforementioned. This system-wide approach assign two virtual cores (in the SW and NW respectively) to each physical processor, together with the mechanism to securely switch between both realms (cf. Qualcomm TEE in Fig. 2). In most cases, a security-oriented OS is deployed on the TEE, which operates and hosts a number of trusted applications (TAs).

The separation between worlds is articulated by different interrupts, I/O hardware, memory views, etc. while prioritizing requests from the SW. This process is orchestrated by means of the *monitor mode mechanism*, which plays the role of the gatekeeper by switching between realms (Sabt et al., 2015).

The *secure monitor call* (SMC) is the component in charge of actually implementing the monitor mode mechanism. SMC requests switching between worlds (secure and normal). Besides, the SMC provides an API within system calls (syscalls) for inter-realms communications. For example, whenever a process running in the NW needs any service provided by a TA, a run state transfer is requested from the NW to the SW kernel (Holding, 2009).

Memory sharing between realms is articulated with two functions *SMC_TYPE_FAST* and *SMC_TYPE_YIELD*[6]. *SMC_TYPE_YIELD* is used for the allocation of a memory area belonging to the NW to be shared with SW, which is particularly useful when high-volume data transfers are involved and in the case of synchronous trusted applications are needed (e.g., video streaming protection). On the other hand, *SMC_TYPE_FAST* enables a mechanism for fast information exchange. It relies on the use of registers with up to a total of four variables to perform data transfers between the two realms.

In Fig. 2, the Exception Level (EL) realms separation is depicted. In this line, N-EL1 means Exception level 1 in non-secure world while S-EL0 is Exception level 0 in secure world. The grey shaded area corresponds to the components that implement the secure world execution. Whereas the blue boxes are components that belong to the non-secure world.

Other components, such as the *TZASC* and *TZMA*, are used for memory management SRAM and DRAM respectively – as depicted in Fig. 3. These implement protection schemes for the static on-chip and for the dynamic off-chip memory. As such, they prevent attempts to access memory within a memory controller by the TZ kernel from the normal global environment. In such a case, the CPU aborts and reacts according to the configured specification, i.e. rebooting the device due to a violation (Holding, 2009).

We notice how TrustZone architecture does not define the way to implement TAs accesses with TrustZone services. Indeed, there are TZ-based implementations with different service definitions, but all sharing the common architecture described.

Access properties are another aspect related to memory management articulated through memory page permissions. For example, those memory regions with write capability are filled up at runtime, and therefore must be located in a modifiable memory area. On the other hand, as in the case with code pages, which

[6] ARM Trusted Firmware. (n.d.). (ARM & Linaro) Retrieved from https://www.trustedfirmware.org/.

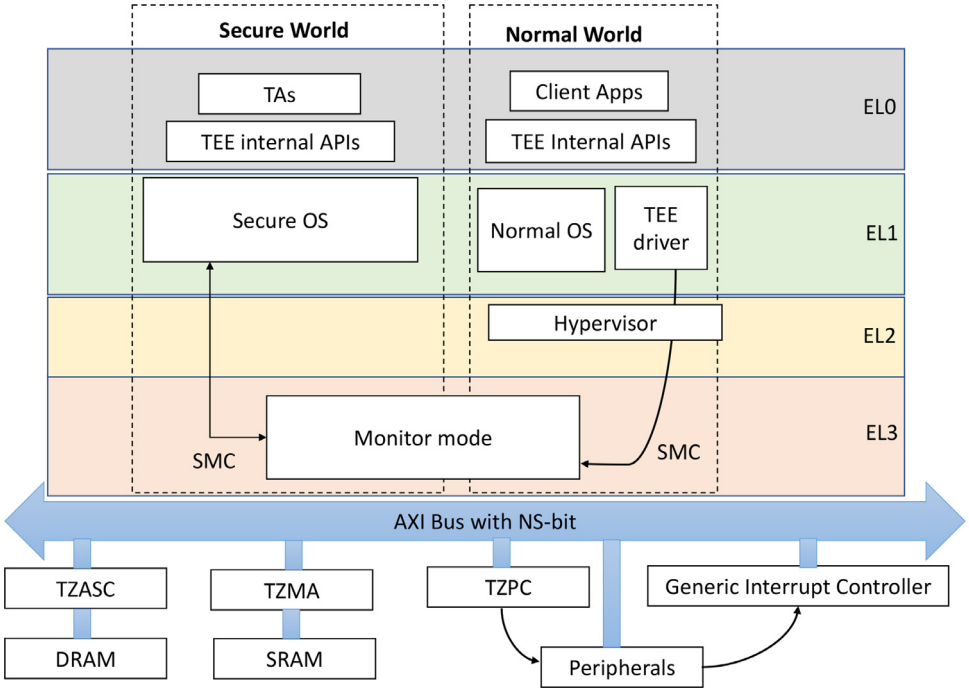

Fig. 3. Architecture on TZ-assisted SoC.

only have read and execute permissions, they may not be modified in any way. The *Domain Access Control Register (DACR)* mechanism is in charge of restricting the access of TEE applications to memory regions of other trusted applications. This is implemented in the *Memory Management Unit*, or MMU. Certain bits (linked to a given memory region) are checked by MMU in the DACR register to specific access properties. In addition, the MMU is in charge of enabling read and write access to the memory allocated to that domain.

Bus management connectivity is articulated using the *APB* and the *AXI* components. AXI is the bus interface implementation for the main system at the chip level. APB implements a low-bandwidth single peripheral bus interface. This interconnection between AXI and APB is implemented with a bridge. Among the different capabilities offered by the AXI interface is the separation of peripherals into realms, allowing both reliable and unreliable ones. For this purpose, it makes use of an extended signaling system together with a flag bit (NS-bit). There is no similar mechanism for the APB bus so the security is managed by the aforementioned AXI-to-APB bridge (Holding, 2009).

We have so far focused on describing the most relevant components to facilitate the understanding of the attacks and flaws presented in the following sections. A full description of the ARM architecture is beyond the scope of this paper, but interested readers can refer to (Ngabonziza et al., 2016) for further details on it.

2.4. TEE implementations

At present there are many different implementations of TEEs, and in the literature it is possible to find different criteria to classify them. The taxonomy presented in Fig. 4 focuses on how the TEE is implemented. On the one hand, there are implementations in which the TEE is implemented with software, such as Overshadow, OpenTEE, OPTEE, etc. On the other hand, there are various hardware implementations of TEE, including Intel SGX, Qualcomm, and others. Another parameter that is used to classify the different implementations is the level of privilege with which they are executed, i.e. if we are dealing with a privileged or non-privileged TEE. Non-privileged TEEs support multiple deployments, allowing to include a new functionality by simply adding new instances without extending the system trusted computing base – which would increase the attack surface of the system. Most of these TEEs make use of a secure monitor from the design stage (which is usually software-based) or by taking direct advantage of hardware-supported secure enclaves (SGX, TPM, AMD-SEV, etc.). On the other hand, priviledged TEEs, in most cases, have access to all system resources.

Table 2 provides a classification of existing TEE implementations according to the taxonomy introduced in the previous paragraph – that is, hardware vs software implementations and privileged vs non-privileged implementations. Note, however, that there are two distinct groups of implementations among the privileged TEE hardware-based implementations. Firstly, there are commercial solutions (Trusty (Google), QSEE (Beniamini), Trustonic (Felton), etc.) and secondly, academic or open source solutions (OPTEE (Brand), Kinibi (Lapid and Wool, 2018), SafeG (Takei et al., 2009), etc.). In addition, we propose TPM as an alternative for Trusted Execution Environments.

2.5. Implementation details of qualcomm's secure execution environment

It is common practice for NW applications to require interaction with others running in SW. KeyStore is the process in charge of managing cryptographic keys in Android, which requires direct communication with the *KeyMaster*. This is a trusted application that provides key secure management using TrustZone capabilities (e.g., secure storage, isolation, etc.). Yet we have to consider that, on the basis of QSEE, user-mode applications are not allowed to perform SMC calls to enter the SW. This limitation is due to the fact that kernel-space privileges are required. In order to overcome this limitation, the Linux kernel driver QSEECOM – QSEE Communicator – allows user-space processes to access several TZ-based operations, such as those related to the communication with the loaded TAs or the actual loading of the TAs in the SW.

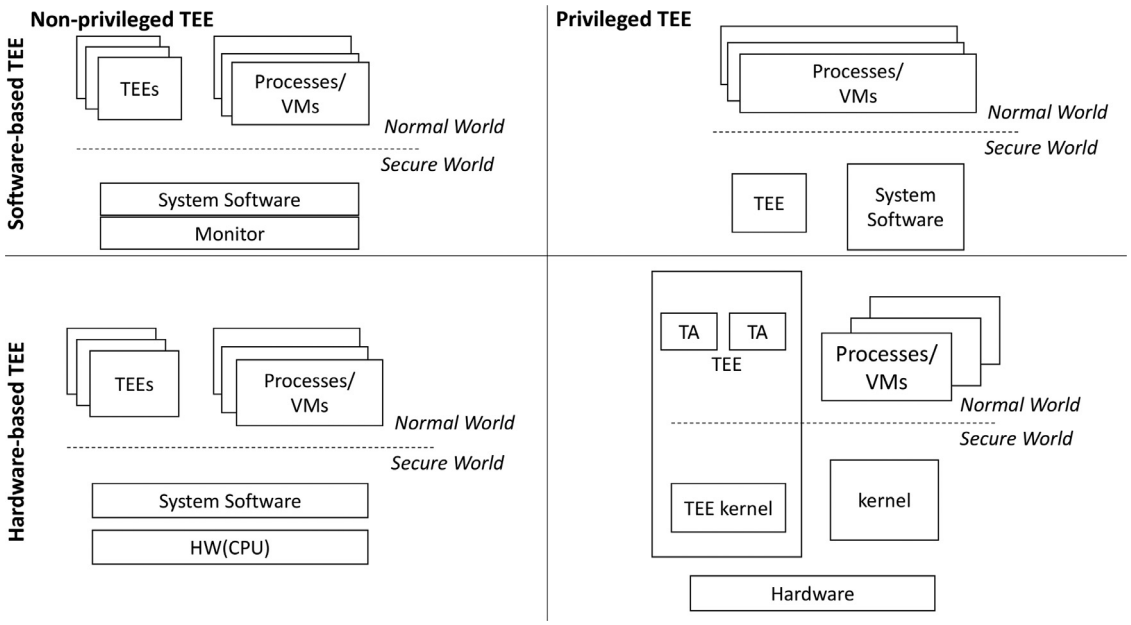

Fig. 4. TEE Implementation Classification.

Table 1
Definitions.

Acronym	Definition	Acronym	Definition
AES	Advanced Encryption Standard	PXN	Privileged execute never
ALSR	Address Space Layout Randomization	QSEE	Qualcomm Secure Execution Environment
AMBA	Advanced Microcontroller Bus Architecture	QSEECOM	QSEE Communicator
APB	Advanced Peripheral Bus	REE	Rich Execution Environment
AXI	Advanced Extensible Interface	ROM	Read Only Memory
BTB	Branch Target Buffer	RO-IoT	Reboot Oriented IoT
CCNT	Cycle Counter Register	ROP	Return Oriented Programming
CLI	Command Line Interface	SC	Stack Cookies
CPS	Cyberphysical Systems	SCA	Side Channel Attack
CRT	Chinese Remainder Theorem	SCM	Secure Channel Manager
DACR	Domain Access Control Register	SCP	Secure Channel Protocol
DCISW	Data Cache line Invalidate by Set/Way	SCTRL	System Control Register
DDR	Double Data Rate	SGX	Software Guard Extensions
DFA	Deterministic Finite Automata	SHA	Secure Hashing Algorithm
DoS	Denial of Service	SMC	Secure Monitor Call
DVFS	Dynamic Voltage and Frequency Scaling	SMMU	System Memory Management Unit
EMFI	Electromagnetic Fault Injection	SoC	System on a Chip
FDE	Full Disk Encryption	SVC	Service Message
FIFO	First In First Out	SVE	System Vulnerability & Effectiveness
FIQ	Fast Interrupt Query	SW	Secure World
FPGA	Field-Programmable Gate Array	Syscall	System Call
GP	Guard Page	TA	Trusted Application or Trustlet
IoT	Internet of Things	TCB	Trusted Computing Base
I/O data	Input/Output data	TCG	Trusted Computing Group
IP	Intellectual Property	TCI	Trustlet Connector Interface
IRQ	Interrupt request	TEE	Trusted Execution Environment
L1	Level One	TEEv	TEE Virtualized
L2	Level Two	TLC	Trustlet Connector
MTM	Mobile Trusted Module	TLV	Type Length Value
NW	Normal World	TPM	Trusted Platform Module
MMU	Memory Management Unit	TZ	TrustZone
MPU	Memory Protection Unit	TZASC	TZ Address Space Controller
ObC	On Board Credential	TZMA	TZ Memory Adapter
OEM	Original Equipment Manufacturer	UART	Universal Asynchronous Receiver/Transmitter
OTA	Over The Air	UUID	Universal Unique Identifier
OP-TEE	Open Portable TEE	UXN	Unprivileged Execute never
OS	Operating System	VBAR	Vector Base Address Register
OU	Organizational Unit	XP	Execution Protection
PLL	Phase-Locked Loop	XPU	External Protection Unit

Table 2
TEE Implementations.

	Non Privileged TEE	Privileged TEE	
		Commercial	Open/Academic
Hardware TEE	SecureBlue+(Boivie and Williams, 2012)	Google TrustyGoogle	Linaro OPTEEBrand
	Sanctum(Costan et al., 2016)	Qualcomm QSEE(Qualcomm, 2018)	ARMithril(Shah et al., 2012)
	AMD-SEV(AMD, 2021)	Trustonic t-baseFelton	GenodeTEE(Feske, 2015)
	OSP(Cho et al., 2016)	Samsung TZ-RKP(Azab et al., 2014)	Microsoft TLR(Santos et al., 2014)
	TrustICE(Sun et al., 2015b)	AuroraLammens	Case(Zhang et al., 2016a)
	Sanctuary(Brasser et al., 2019)	SierrawareSierraWare	TrustOPT(Sun et al., 2015a)
	Intel SGX(Intel, 2014)	Solacia SecuriTEESolacia	SafeG(Takei et al., 2009)
	Haven(Baumann et al., 2015)*	mTower(Drozdovskyi and Moliavko, 2019)	VimoExpress(Oh et al., 2012)
	SCONE*(Arnautov et al., 2016)	T6TrustKernel	Kinibi_M(Trustonic, 2017)
	Graphene-SGX*(Tsai et al., 2017)	ObC (Kostiainen et al., 2009)*[deprecated]*	Andix OS(Fitzek et al., 2015)
	Panoply*(Shinde et al., 2017)		
Software TEE	Overshadow(Chen et al., 2008)		
	Virtual Ghost(Criswell et al., 2014)	Nested Kernel(Dautenhahn et al., 2015)	
	Inktag(Hofmann et al., 2013)	OpenTEE(McGillion et al., 2015)	
	Flicker(McCune et al., 2008)	MicroTEE(Ji et al., 2019)	
	TrustVisor(McCune et al., 2010)	SoftTEE(Lee and Park, 2020)	
	Multizone(Pinto and Garlati, 2020)	Trustshadow(Guan et al., 2017)	
	Utango(Oliveira et al., 2021)	Kinibi(Lapid and Wool, 2018)	
	Sego(Kwon et al., 2016)	SKEE(Azab et al., 2016)	
	SICE(Azab et al., 2011)		

For the implementation of Secure Monitor calls from the kernel space an interface was included in the driver. This interface between QSEECOM and the SW is known as *SCM*, which is considered the widest attack surface of the TEE since is one of a small number of communication channels between the outside world and the SW. Therefore, a limited number of processes are allowed access to QSEECOM for the sake of security. As such, Beniamini's et al. Beniamini implementation limits the number of processes which can access the QSEECON from the normal world to only four:

- SurfaceFlinger (running with "system" user-ID): This is a system service in charge of the composition of the application and system surfaces, for which a shared buffer is enabled.
- DrmServer (running with "drm" user-ID): This element is in charge of managing digital rights.
- MediaServer (running with "media" user-ID): This element is in charge of handling multimedia services.
- KeyStore (running with "keystore" user-ID): This element is in charge of creating, storing and managing cryptographic keys.

Note that vulnerable processes should not have access to the TEE because if the vulnerability is exploited by an attacker, the attacker could gain access to any application running in the SW bypassing the Linux kernel filter on the process. A known weak point is the language in which trusted applications are written. Most applications use the C language instead of safe languages that potentially decrease the possibility of vulnerabilities.

The TrustZone fast and yield commands used for memory sharing are implemented by Qualcomm[7] using two functions: *SMC_TYPE_YIELD* and *SMC_TYPE_FAST*. The first one allocates a common memory area for communications between worlds. When this function is called a memory record is populated. The record includes the maximum buffer size, the buffer headers, as well as offsets of the data to be sent and received. The second is used to start a short-term communication where the data to be exchanged are relatively small. Either function can be used to issue an SMC or to call a service.

As previously mentioned, the first defense mechanism in this situations is the DACR provided by ARM, which prohibits altering any of the TZ kernel pages. Some recent TrustZone-enabled Qualcomm System on a Chip (SoC) integrate an additional mechanism for memory access control. This hardware-based Memory Protection Unit (MPU) are pre-configured to mark as write-protected certain memory regions predefined by the manufacturer.

In Qualcomm these MPU units are called External Protection Units (XPUs). Among the tasks carried out by the XPUs is preventing access from the NW to the SW and to the memory areas restricted by the manufacturer. As an example, the XPU mechanism is used to allocate TrustZone kernel code into write-protected memory areas, which are checked during the secure boot of the system to ensure that it has not been altered.

One sensitive aspect is how to load trusted applications and their revocations when Qualcomm secure booting actually takes place. In this line, regular Executable and Linking Format (ELF) files are signed by Qualcomm. These files attach a single hash table segment, which is a signature blob with the hashes of each ELF segment, along with the certificate chain. Verification of the signature with the concatenated blob of hashes is performed with the public key of the attestation certificate (the last one in the chain). Validation is performed by comparing the hash of the root certificate and the *Root Key Hash* stored on the device. It is stored in the ROM of the device and integrated in the SoC.

We now briefly describe how the chain of trust workflow is implemented. The procedure begins with the issuance of a hardware-bound key for the validation of the certificates. Later, these certificates can be used to validate the binary signature. In addition, Qualcomm includes additional Organizational Unit (OU) fields with information necessary for security enhancement in the binary signatures.

Note that since TEEs are considered entities with high privileges the Normal World has no inherent mechanisms, not even DACR or XPUs, to protect against unauthorized memory accesses and manipulations from the Secure World. Therefore, it is trivial gaining access to the NW kernel for an attacker in case a TEE becomes compromised, even if no vulnerabilities were present in it.

3. Taxonomy of attacks

Although TEE has been designed to provide advanced means of secure code execution that traditional operating systems do not implement, they can still be attacked. Here we describe the taxonomy of attacks that will be used throughout the rest of the article.

[7] Qualcomm Product Security. Retrieved from: https://www.qualcomm.com/company/product-security.

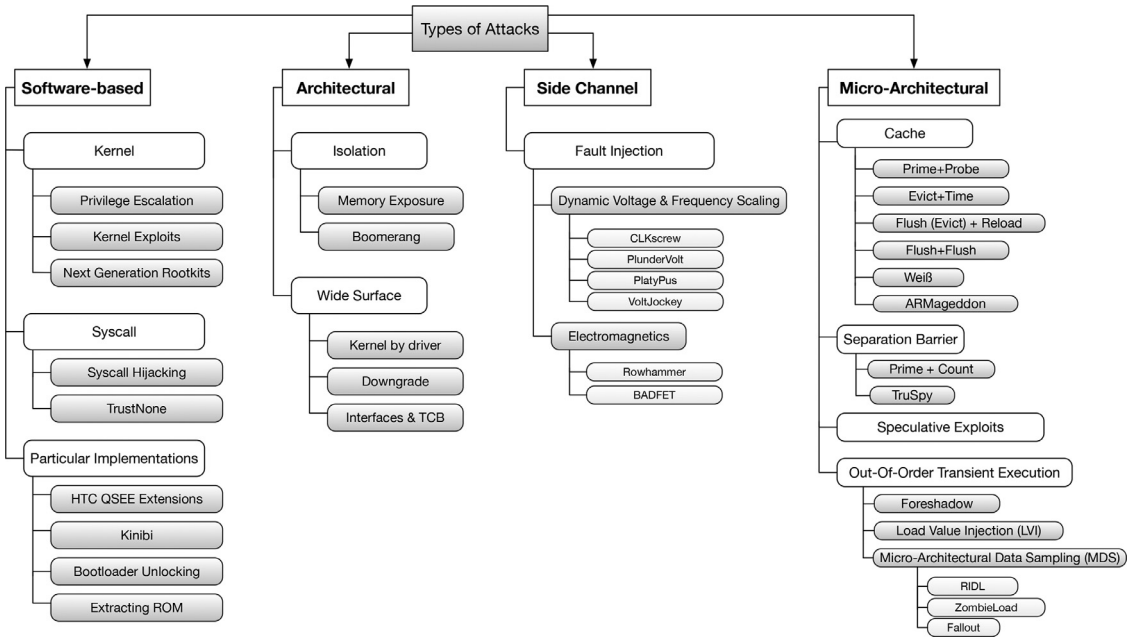

Fig. 5. Taxonomy of Attacks to TEE Implementations.

In addition, Fig. 5 provides a summary of every specific attack for each category.

- **Software-based attacks** (Section 4) are dedicated to exploit different elements of software stack, including operating system and the applications running on it.
- **Architectural attacks** (Section 5) exploit fundamental design flaws in the hardware architecture of the system, rather than software bugs.
- **Side-Channel attacks** (Section 6) are focused on the transmission of data between the Normal and Secure Worlds by modulating the behaviour of some system elements, such as execution times or power consumption.
- **Micro-architectural attacks** (Section 7) are a particular type of attack that focuses on micro architecture elements such as exploiting the cache or Branch Target Buffer (BTB).

4. Software-based attacks

Programming errors cause functional inconsistencies, which can lead to bugs in the memory protection mechanisms, in the security mechanisms themselves, or in peripherals configuration. These bugs can appear randomly during the system execution, either during its validation with the trusted kernel, the secure monitor, the boot loader, or the applications themselves. Such bugs can be exploited through various means (e.g. parameter validation, buffer overflows) for various purposes – from revealing sensitive information to exploiting the kernel. In this section, the most representative TEE vulnerabilities caused by implementation bugs are described. Since each implementation has particularities in its architecture, which directly affect the way Trusted Applications (TAs) interact, we describe some of the most relevant cases exemplified in concrete implementations.

4.1. Kernel attacks

This section describes direct attacks on the system kernel. This includes privilege escalation attacks, kernel exploits and a new generation of rootkits.

4.1.1. Trustzone privilege escalation

Qualcomm's implementation, known as QSEE, is used in several smartphones – such as Pixel, LG, Xiaomi, Sony, HTC, OnePlus, and Samsung, among other devices. Due to its importance, there are various software-based attacks that specifically targets the Qualcomm implementation. One of such attacks focuses on accessing the protected memory of QSEE through escalation of privileges (Beniamini (2015b),Beniamini,Beniamini,Beniamini (2016a)).

Fig. 6 shows the first three-stepped (Beniamini, 2015b) privilege escalation attack. Firstly, the attacker exploits a vulnerable implementation of the MediaServer Android application. This runs in the NW with zero permissions. Still, MediaServer was granted privilege for accessing the QSEECOM driver for communications with the TEE and therefore with the WineDive TA. Subsequently, the attacker could exploit a vulnerability in the QSEECOM driver and gain control of the kernel through the MediaServer. We highlight that this driver runs in NW context. Henceforth, the attacker with kernel privileges in the NW can make direct SMC calls to the SW. As a consequence, the attacker can manage to execute the code of his choice in the context of a TA. Moreover, since by making use of the SMC syscalls the privileged kernel applications have direct access to the TEE, the attacker can now execute various privilege escalation attacks to run shellcode within the TrustZone kernel.

Now, we will explain what additional steps need to be executed once an attacker gains control of QSEECOM. At this point, the attacker can execute SCM calls to write a zero DWORD in any specific memory address, in an operation known as 'zero-write primitive'. This can be used to disable the mechanism used for checking bounds on all memory addresses passed to the SW. Once this operation is disabled, the attacker can exploit other SCM calls creating different primitives. For example, once the control mechanisms are invalidated, the attacker can use the SMC calls to transform what was a 'zero-w primitive' to an arbitrary 'w-r primitive'. Once the attacker has achieved write permissions, he still has to identify those memory regions where to host his own shellcode, so as to bypass the TZ kernel pages protection mechanism. Since privileged kernel applications have direct TEE access, making use of SMC syscalls enables an important attack vector that may result in privilege escalation attacks.

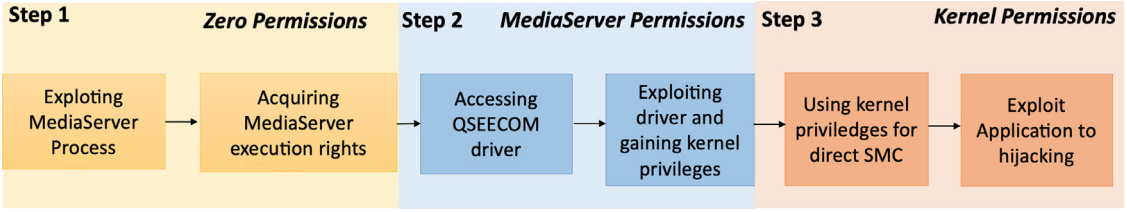

Fig. 6. Three Stepped Privilege Escalation Attack.

The Domain Access Control Register (DACR) register from ARM MMU is responsible for protecting the TrustZone memory by controlling accesses to it. However, by making use of the arbitrary write primitives already described, it is possible to modify the value of the DACR and thus enable reading and writing the memory regions controlled by the mechanism. By doing so, the attacker can now insert his shellcode in memory areas reserved for execution within the kernel. Moreover, since these areas are never used by the kernel, any modification in them goes unnoticed.

4.1.2. Kernel exploit in TrustZone

This exploit describes how it is possible to take control of the operating system kernel through a series of chained exploits. This opens the door for the attacker to gain privileges to the TrustZone kernel. An example of this exploit is provided by Beniamini et al. (Beniamini,Beniamini), which describes how a series of chained exploits provide an alternative way to the previous attack. These chain of exploits take advantage of buffer overflows and vulnerable syscalls to ultimately execute arbitrary code with TrustZone kernel privileges.

The attack starts once the attacker has gained control of the QSEECOM driver, located in the NW. Now, the trusted Widevine application (located on the SW) can be exploited by causing buffer overflows, using a disused function called *PRDiagVerifyProvisioning()*. Once the buffer overflow is achieved, any code within the context of the trusted application can be executed. Still, although the attacker can make use of a Return-Oriented Programming (ROP) chain to execute his code, the application's executable code fragments are inserted as read-only. For this reason, the code execution must be split into two parts, where any part of code that does not require QSEE privileges will have to be executed within the Normal World.

At this point, access to the TEE is allowed indirectly through the use of certain (privileged) applications as intermediaries – and these, in turn, can then establish communication with the TEE through the driver. Even so, the attacker is restricted to running code in the QSEE user space, since he is not yet granted TZ kernel privileges. However, the attacker can exploit vulnerabilities in syscalls API provided by the TZ kernel.

The SVC instruction allows applications to call the syscalls of the TZ. This instruction is handled using the Vector Base Address Register (VBAR). Whenever a syscall is performed, control of the code and the execution flow passes to the NW kernel. However, the TZ only performs very basic validity checks on the provided input buffers: all arguments provided in legitimate application syscalls are accepted as valid. Therefore, once the attacker has identified a vulnerable syscall that allows him to overwrite any syscall handling function pointer, he can use the WideVine TAs to exploit the TZ kernel and modify the syscall handling functions. All that remains to be done is to identify a suitable memory area for inserting the shellcode. Despite of TA code segments can be considered write-protected due to the DACR mechanism, but in fact these segments are still susceptible to be overwritten with the described syscall bug.

Thereafter, as a consequence of disabling the DACR mechanism, the attacker can insert his shellcode anywhere in the application code. Likewise, he may also use mutated syscall control functions to execute his shellcode within the context of the TZ kernel and execute any arbitrary code. Note that classical security measures such as ASLR[8] could prevent common code execution and privilege escalation attacks, but they are not implemented in this context.

Precisely, Project Zero (Beniamini, 2017) provided an analysis on the implementation of such security measures in TEEs. They conclude that Qualcomm and Kinibi, the leading exponents of TEE implementations, only implement very few security mechanisms. In the case of Kinibi, it does not offer any type of ASLR mechanisms, forcing all applications to be loaded at a fixed memory address. On the other hand, Qualcomm's TEEs only offer a weak implementation of ASLR. Therefore, the security boundary between the TZ kernel and applications is very fragile, at least in concrete implementations like QSEE. In fact, when the attacker manages to enter the Secure World and takes over an application, the communication channel between TZ kernel and application is constructed in such a way that no input validation mechanism is implemented, and it is trivial for the attacker to compromise the kernel.

4.1.3. Next generation rootkits

A series of rootkits considered to be new generation rootkits are included in this section, as they take advantage of several of the weaknesses already described and even others yet to be described related to architecture, side-channel or micro-architecture to explore weaknesses in the system.

Roth (2013) shows weaknesses in TEE combined with a specific architecture. They also describe how these weaknesses allow the development of rootkits such that they can control the system in a way that goes unnoticed. Since the SW has privileged access to the memory, it also has the ability to modify the NW kernel structures. Moreover, it can also block the NW from accessing its own memory. In particular, what Roth provided was several mechanisms to hide the visibility of the code running in the SW, thus complicating detection. Some of these rootkits exploit flaws in the TEE architecture itself to exploit vulnerabilities as described in Section 5, but these rootkits are software and although they also make use of attacks from other categories, they are eminently software for the most part and are therefore included here.

4.2. Attacks using system calls

This section includes attacks that make use of the set of system calls. Particular attacks such as TrustNone or hijacking attacks are included.

4.2.1. Syscall hijacking

Certain attacks focus on executing various syscall hijackings in the context of the TEE in order to gain access to protected information. Along these lines, Beniamini (2016a) describe an attack that

[8] Address space layout randomization (ASLR) is a computer security technique used for preventing memory corruption vulnerabilities exploitation.

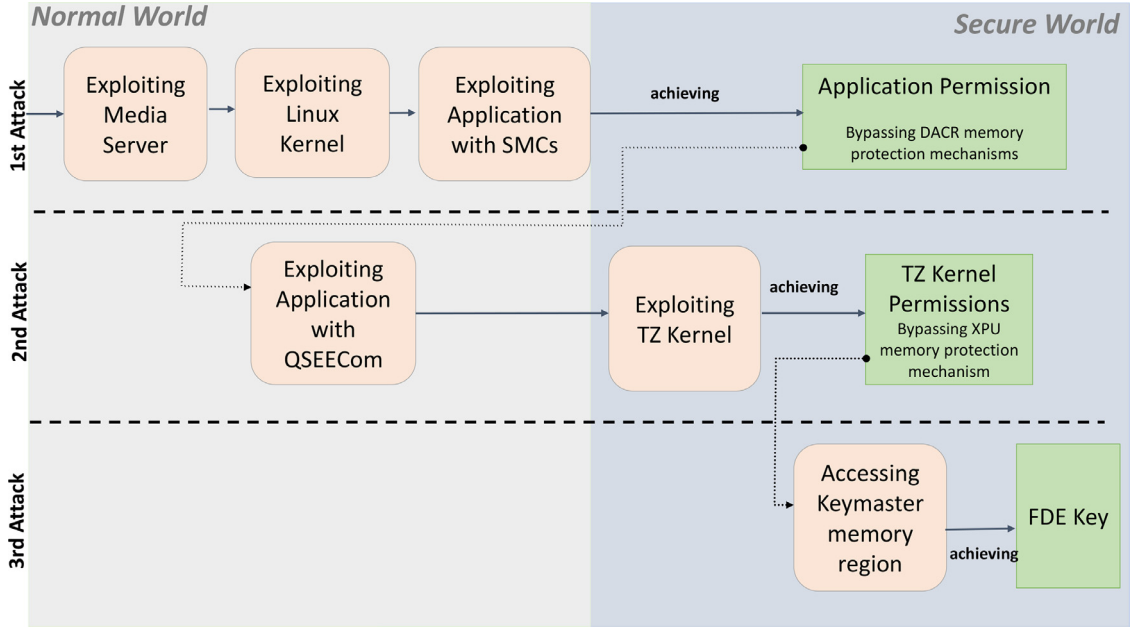

Fig. 7. Three attacks Overview.

can extract any key residing in TEE, as with the full disk encryption (FDE) key. This allows the attacker who successfully perpetrates the attack to decrypt any encrypted disk on the android device. This attack makes use of the different exploits described in Sections 4.1.1 and 4.1.2. For the sake of clarity, an overview of such attacks, including a description of how they are chained together, is shown in Fig. 7.

It essentially chains the previously mentioned exploits up to the point where the attacker has gained control of the QSEECOM driver and has exploited the vulnerable WideVine TA. However, because of the aforementioned XPU protection, QSEE applications do not have access to the memory of other QSEE applications. In particular, the Widevine TA cannot access the Keymaster memory. Still, every QSEE application has access to TZ kernel code segments as long as they are executed in the context of the kernel. The Widevine TA can execute the shellcode and thus access the Keymaster memory once the shellcode is hosted in the TZ Kernel. Therefore, the ultimate goal of the attacker is to insert the shellcode within the TZ Kernel and execute it through the Widevine TA. The shellcode will then access the Keymaster memory and extract the FDE Key.

In order to succeed in inserting the shellcode in the TZ kernel code segments, it is necessary to bypass various security mechanisms. The first mechanism to bypass is the DACR memory protection mechanism. The MMU manages access to any memory region, using bits of the DACR register. However, there is a piece of code inside the TZ core that can change the value of DACR, known as the DACR modifying gadget. If the attacker calls the DACR modifying gadget to set all bits to 1, then all memory regions are then enabled and available to perform read and write operations on them. The first goal of the attacker is to execute this DACR modifying gadget.

In order to execute this gadget, the attacker can take advantage of the design of the system call table. System calls are used indirectly using a system call table. Although this table cannot be changed, as it is protected by the memory protection unit (XPU) pointers, the reference to this table is not protected: it must reside in a modifiable memory region, because it is only filled at runtime. Because of this, the attacker can execute a sycall hijacking attack: the attacker stores in memory a fake system table with one system call pointing to the DACR modifying gadget, and then modify the reference to the system call table so it points to the malicious one. This way, once the (modified) syscall is called, the DACR modifier gadget will be invoked instead – modifying the DACR register to allow write and read access.

The second security mechanism that needs to be bypassed is the memory protection unit (XPU), which prevents access to protected areas from unprivileged code. The issue here is that the attacker can execute code in the kernel context, yet the source of the code is in the trusted WideVine application – and is therefore considered unprivileged. The attacker then must find a way to insert the malicious code in the TZ kernel and to invoke it.

The attacker first needs to implements a script in order to identify unprotected code regions in the TrustZone kernel. This allows finding a “cave” to host the final shellcode of the exploit, which will be considered as priviledge code and will bypass the XPU protection mechanism. Once the script successfully finds a “cave” and the shellcode that extracts the encryption key from the memory disk is inserted, a final step remains: how to execute such shellcode. In order to do so, another system call hijacking is needed. For example, the attacker can overwrite the qsee-hmac() system call. As a result, when the qsee-hmac() is called from the malicious QSEE application, instead of the intended function the shellcode will be executed. This allows the FDE key to be extracted from the KeyMaster application and then written to the shared buffer.

The cause of this attack is that disk encryption is not implemented with a hardware-based key. The key is generated by software and stored inside the TZ kernel memory. Since the key resides within the software, once the TZ kernel is exposed, it can be easily extracted. Therefore, the disk encryption system offered by Android becomes resistant to attacks of different kinds such as those of the TZ kernel security or TA’s own keymaster. Any flaw in either of them can potentially leak the FDE master key.

In addition to the ability of applications to map physical memory, there is another attack gap arising from TEE’s debugging mechanisms. What privilege escalation attacks are and how they work has already been described in Section 4.1.1. Making use of this type of attack, Shen (2015) implements an attack on Huawei’s TEE. It exploits a syscall that allows any application to perform a stack dump in a memory area belonging to the NW. This becomes

Fig. 8. Kinibi Architecture.

the attacker aware of the physical address space of the GlobalTask to have enough information to successfully implement the attack.

4.2.2. Trustnone

Communication with the TZ kernel is facilitated through the SMC instruction, as aforementioned. This allows the NW to use system calls that are exported by the TZ kernel, for which an API is provided in the Android/Linux kernel.

XPU units protect those on-chip and off-chip memory regions that contain the TZ kernel. These are configured by the first boot loaders. This allows only certain runtime environments to access certain memory areas.

Beaupre (2015) describes that a number of TZ vulnerabilities are related to system calls. With special emphasis on those that do not implement any validation, or do not do it properly. More specifically, in the user input, at this point the attacker could safely write as many zeros as desired in a memory area, thus bypassing the implemented security mechanisms obtaining read and write permissions in the TZ kernel context.

The attack is particularly relevant because it affects all devices using the Snapdragon 805 SoC and thus the QSEE. In his experiment, Beaupre used the exploit to unlock the bootloader of a Motorola Snapdragon 805[9]

4.2.3. Attacks on HTC QSEE extensions

Beyond the vulnerabilities that can be found on QSEE, there are also vulnerabilities that affect certain QSEE extensions from specific manufacturers. For example, in Keltner and Holmes (2014), Keltner et al. describe the implementation of a new attack against a version of Qualcomm's QSEE used and extended by HTC. To create this attack, they reverse-engineered that specific implementation/version of QSEE, which proved highly successful in finding a number of vulnerabilities in the code added by the HTC extensions.

Examples of such vulnerabilities include i) flaws in the zero-write primitive in certain address range allowing to circumvent all memory operations security checks, and ii) flaws in the "tzbsp_oem_memcpy" function, which give the attacker full control of all the memory. As a consequence of all the weaknesses, it is easier for the attacker to securely extract data and modify validation mechanisms in memory regions.

4.2.4. Implementation bugs

The previous sections have focused on the QSEE TEE by Qualcomm. Yet this is not the only vulnerable implementation of the standard: other vulnerabilties have also appeared in other implementations of the TrustZone technology, such as Kinibi (Lapid and Wool, 2018) from Trustonic.

One important work in this area is proposed by Komaromy (2018) that described certain important vulnerabilities affecting the Trustonic implementation. These six vulnerabilities were caused by software bugs, and most of them are located in components that manage inter-realms communications.

Before describing these vulnerabilities, it is important to provide a very brief introduction on the Trustonic architecture. Trustonic (cf. Fig. 8)includes an application connector or gatekeeper known as TLC (trustlet connector) that enables communication to pass through to the Kinibi device. An interface is offered to NW by TLC that can be accessed through UNIX domain sockets. These domain sockets make use of MAC/DACs schemes for access control and only certain applications, such as *tlc_server*, have access to them. In addition, sanity checks are performed on TEE requests, and are further protected through SELinux.

Komaromy (2018) found a way to circumvent this access control by disassembling the *tlc_driver* binary. It was found that although almost all commands implemented a process for checking the caller's permissions, there was one command that, for some reason, did not have this security check implemented. This vulnerability, *Vuln 0*, allowed an arbitrary user-space application to make use of the handler and initiate a session to a TA and subsequently send any commands at will to it.

One of such trusted applications (TA or trustlet) is ESECOMM, which is used for secure payment transactions. ESECOMM implements the "SCP03 Global Platform Secure Channel Protocol", where messages are sent encoded in TLV (Type-Length-Value) format via APDUs (Application Protocol Data Units). The trusted application performs certain parsing ckecks on the TLV-enconded messages but does not control whether the maximum number of TLVs to store for each structure is exceeded. This may result in overflow (*Vuln 1*) attacks, which opens up the range of possible attacks since these structures are allocated on both the heap and the stack. In addition, the TLV parser does not properly check the input buffer (allocating TLVs) length – the only check performed is whether the offset remains unchanged until the end of the buffer, it does not check that it is less than it. Therefore, this allows an attacker to trivially read out of bounds (*Vuln 2*).

However, these are not the only vulnerabilities that affect the ESECOMM trustlet. There is another stack buffer overflow in the "*parse_ca_cert()*" function. Again, no check is made on the length of the TLV input value, so it is possible that another buffer overflow may occur. Although the size of TLVs is restricted to 0x400 bytes, since the size of the input buffer is limited to 32 bytes, the proposed restriction is not sufficient to prevent the attack (*Vuln 3*).

There is another function, "*parse_scp_param()*", with a similar vulnerability. This function is used to parse the *Diffie-Hellman* Diffie and Hellman (1976) parameters used for establishing a secure channel between Kinibi and the secure element. As in the previous case, the function parses and checks most of the parameters but there is one parameter that is not fully checked, thus enabling another overflow (*Vuln 4*) attack.

Finally, the fifth vulnerability (*Vuln 5*) is a memory corruption vulnerability that requires the user to have root privileges. The main problem lies in the common buffer shared by that both worlds, NW and SW. In this buffer, known as TCI, there is a flaw in the way memory offsets are specified. In particular, within the buffer there is a file (*envelope_len*) with the offset where the response begins. The tlc_driver is in charge of setting this field, but any other trusted application can also do it. As a result, if an attacker is able to become root, he would be able to arbitrarily modify this field and thus specify whatever write offset he wishes, even beyond the buffer bounds.

While we have focused on vulnerabilities that affect the Kibini implementation, that does not mean that there are no flaws in other TrustZone implementations. For example, in Keltner and

[9] https://www.qualcomm.com/products/snapdragon-processors-805.

Holmes (2014), the authors describe the procedure to read and write operations on arbitrary memory locations within the SW using the failed memory validation mechanism. Similarly, Rosenberg (2014) observed a faulty SMC memory check mechanism. This flaw enables an attacker with kernel privileges to write into the SW.

4.2.5. Unlocking bootloader attacks

There are other TrustZone attacks that target the bootloader of smartphones, such as the attacks described by Rosenberg (2013, 2014). In the first paper, Rosenberg describes a write vulnerability in Motorola smartphones. This vulnerability affected a specific SMC call whose role was to allow the kernel in the NW to obtain values stored on the memory side of the safe world. However, an attacker can abuse this SMC call to overwrite the memory in the secure region – in particular, the flag responsible for granting the TrustZone kernel permission to blow Qfuses. As a result, the attacker can blow Qfuses through another SMC call, in order to indicate that the bootloader is unlocked. This way, an unsigned image (e.g. a tampered Android firmware) can be loaded.

In the second paper, Rosenberg (2014) identifies a new vulnerable SMC function. The function, known as *qsee_is_ns_memory()*, checks whether a certain memory range belongs to the SW. This function involves an uncontrolled primitive write based on an overflow. This vulnerability enables a chain of attacks that gives the attacker the possibility of circumventing all validation checks and execute any code in safe memory region, unlocking the bootloader in the process.

4.2.6. ROM Extraction attack

There are other attacks, such as Basse (2016) by Basse et al., whose goal is to bypass the TrustZone authentication mechanisms to extract the boot image (BootROM) from a device. In ARM devices, an UART interface is available in the device to give access to a root shell and a high-level debug message interface. Still, the BootROM image is stored in a secure memory area within the SoC to prevent unauthorised access or changes. To bypass the security measure two conditions must be met: i) the MMU tables must be extended to include the BootROM address (thus allowing access to this partition), and ii) the user needs kernel privileges.

Although an attacker can exploit existing overflow errors in the SMC interface to gain kernel privileges, the access to the memory is limited due to the authentication routine that protects the MMU images. However, in some cases, this authentication routine is a mere hash function. Therefore, an attacker can update the MMU table to include the BootROM, recalculate the hash of the MMU table, and write both values in the device. A custom SMC can then be executed, which will access the BootROM partition through the tampered MMU table.

5. Architectural attacks

This section presents the main security issues arising from the architecture of today's TEE systems. We distinguish between attacks made possible by the elements of the architecture dedicated to the isolation between worlds (SW vs NW) and attacks on memory protection mechanisms.

5.1. Isolation focused attacks

Attacks on inter-world isolation include (a) memory exposure due to physical memory mapping in the NW by applications, and (b) information leakage due to TEE debugging mechanisms.

5.1.1. Memory exposure

Certain TAs require an efficient shared memory mechanism with the ability to exchange large volumes of data between worlds, which has led to security holes in some TEE implementations.

Beniamini Beniamini (2016b) describes how an attacker, starting with only TA privileges running in the NW, can get full control of the kernel, which is due to the fact that Qualcomm's TEE implementation allows an arbitrary application to allocate an arbitrary area of the Normal World. For this, it is only necessary to use a call to the SW, which in turn allows the attacker to take control of the operating system. This would enable him to sweep through all the physical addresses of the kernel, manipulate it and introduce backdoors.

Fortunately this is not the case for all implementations. In the case of Trustonic TEE, TAs cannot read from or write to physical memory.

5.1.2. BOOMERANG attack

Boomerang attacks Wagner (1999) exploit flaws that appear in the design of the communication between realms. This type of attack is made possible by the fact that the trusted OS has no restrictions on the memory addresses it can access and the normal OS has no way of checking if the entity performing this action is entitled to do so. The attack starts with an application or user in the NW passing an unauthorized memory address to a SW call. If the address is not filtered out due to the lack of standard memory sanitation mechanisms, the attacker could read and/or write that memory, as detailed in Section 7.1.

Fig. 9 shows an overview of the attack. The attacker's goal is to send a privileged address to the application (4). For this purpose, and in order to circumvent the sanitation process, a filled data structure is transferred – which among other things contains an address pointer without annotating it. There are three possible ways to transfer the data to the existing mode: (1a) by using the Daemon TEE in charge of pointer sanitation with background execution, (1b) by taking advantage of an API that is used by the application, and (1c) by using a library for the aforementioned API. The NW OS kernel makes a call to the SMC with the purpose of switching worlds and transferring the filled data structure to the SW (2). Once the data structure is in the SW OS, a check is made to see if the pointers actually point to memory areas from the SW. As the pointer comes from the NW, it passes the test and the trusted OS passes the structure to the TA (3) without any further checks.

Based on how an attacker bypasses pointer sanitation, Machiry et al. (2017) successfully attacked a wide variety of TEE architectures. Using a static analysis tool, they were able to perform analysis of several TEE implementations (QSEE, Kinibi, OP-TEE (Brand), SierraTEE (SierraWare), and Huawei) and applications on them, searching for BOOMERANG vulnerabilities. The results of the study revealed several vulnerabilities in the analyzed platforms, which affected a very high number of mobile devices. This work has enabled TEE vendors to implement specific fixes in their environments.

5.2. TEE Wide attack surface

Attacks to memory protection mechanisms include certain bugs appearing in software drivers (executed in kernel space), others appearing in the interfaces shared among different TEE components and broad interfaces.

5.2.1. Kernel contains driver execution

Most systems require software drivers to communicate with specific hardware. Some TEE drivers are meant to interact with devices that handle sensitive (e.g. a biometric sensor) and for that reason they are executed in the TEE kernel. Therefore, an attacker

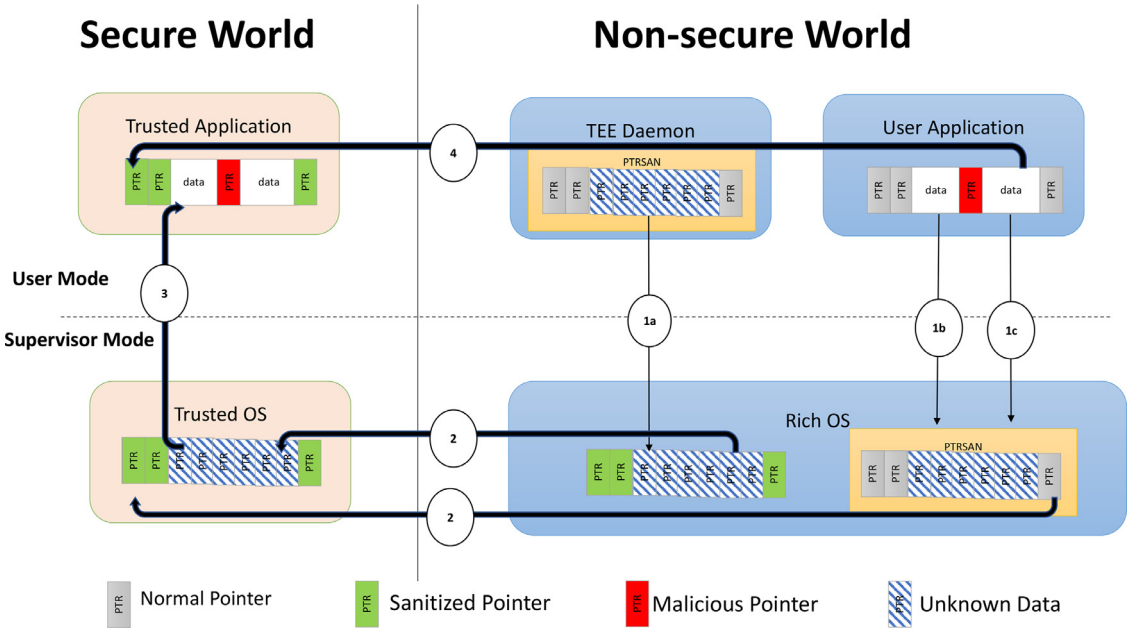

Fig. 9. An attacker bypasses pointer sanitation by hiding it inside the structure to send to applications.

could exploit any error in these drivers in order to access the privileged area of the system. In fact, some implementations like OP-TEE Brand and Snapdragon (Rosenberg, 2014) allow the execution of all the code labelled as privileged within the kernel.

5.2.2. Downgrade attack

Trusted applications are signed using the TEE trusted public key. If the application passes the verification, the system will accept it and execute it. This is exploited by downgrade attacks, which consist of loading old buggy binaries to take control of the system. Chen et al. (2017) demonstrated the effectiveness of this kind of attack.

Nowadays, in order to prevent such attacks, the majority of TEEs implementations include some kind of mechanism to control the application versioning. However, Beniamini (2017) analysed a number of applications and their respective updates and realized that all shared the same version number.

Application developers are therefore urged to make use of the version control mechanisms provided by the TEE vendors. This shows that even when protection mechanisms are in place it is important to make use of them or they are rendered useless thus opening the door to attacks.

5.2.3. Broad interfaces to attack

Opening secure system has always been tricky and dangerous. In order to extend functionalities the number of interfaces offered by TEE is growing and this has led to the development of several exploits. For example, the exploit on the TZ linux driver (Beniamini, 2015a) in Android. Trusted applications are also being provided with more functionality, which is also sensitive from a security point of view.

TEEs should allow developers to minimise the Trusted Computing Base (TCB) of their applications to maintain a proper security/efficiency balance: the larger the size of the TCB, the more error-prone implementations are (Cerdeira et al., 2020). It is worth noting that the size of the TCB varies considerably for TEE each implementation, ranging from 97KB for Tegra's TEE to 1.62MB for Qualcomm's.

6. Side-Channel Attacks

As mentioned above, memory protection mechanisms in TEE implementations are rather weak or lacking. In this section we show how exploiting these mechanisms lead to side-channel attacks (SCA). An SCA is an attack that exploits certain types of information such as power consumption data to leak information about cryptographic material and operations.

Fault-injection is a particular kind of side-channel attack consisting on inducing physical- or software-based faults (also referred to as glitches) in a computation to expose secret information. Due to their relevance, we focus on this type of attacks. This type of attacks include the application of high voltages, temperatures or electromagnetic (EM) pulses in order to expose electronic components to unexpected conditions. Electromagnetic fault injection (EMFI) attacks Maistri et al. (2014) are probably the most relevant and difficult to protect from. These attacks have provided very successful results when implemented on a huge number of commercially available integrated circuits.

Some of the most relevant fault-injection attacks are known as Dynamic Voltage and Frequency Scaling (DVFS), which allow the software to regulate device voltage and frequency based of each CPU execution thread. This makes it possible to modify and monitor the power consumed since this value is directly related to both factors (frequency and operating voltage). Some of them, namely CLKscrew Tang et al. (2017), Plundervolt Murdock et al. (2020b), Platypus attack Lipp et al. (2021) and VoltJockey Qiu et al. (2019a) are based on producing dynamic voltage and frequency scaling, where power traces can be collected by software and there is no need to physically access the device itself. Additionally, Rowhammer Lipp (2016) and BADFET Cui and Housley (2017) are attacks based on the application of electromagnetic pulses.

6.1. CLKscrew

CLKscrew takes advantage of a feature available in modern devices that enables software control of both CPU voltage and frequency for the primary purpose of power administration.

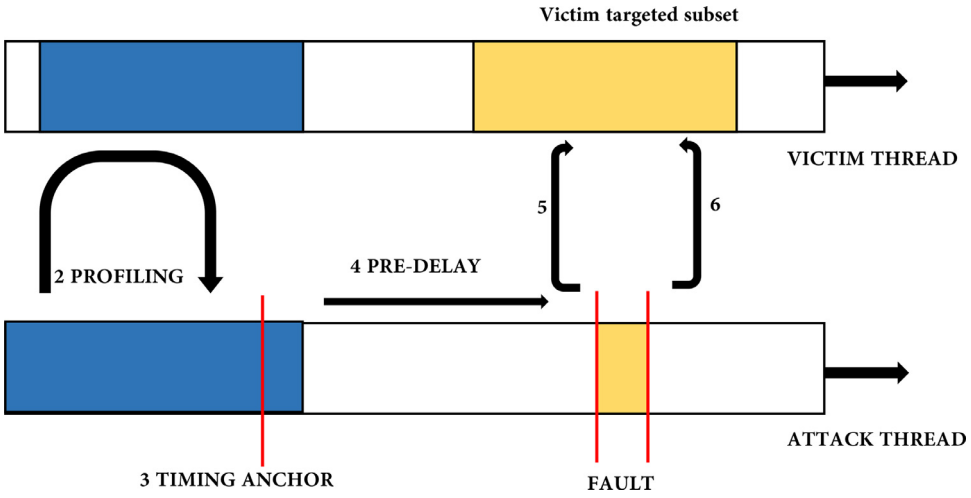

Fig. 10. CLKscrew fault injection Attack.

Tang et al. (2017) show a successful implementation of the attack on an ARM device, namely the Nexus 6 smartphone. This attack consists of inducing failures in certain operations by causing calculation errors in the CPU, allowing the attacker to obtain essential information to deduce secret keys from an ARM TrustZone.

To cause erroneous behaviour, the attacker can overclock and undervolt the CPU, thereby exceeding the CPU fault induction boundaries. There are no protection mechanisms to prevent the CPU from being able to operate at faulty frequency and voltage combinations. Also, since hardware regulators[10] have their operating range precisely at the TEE separation, this opens the possibility that the attack can occur even in the same SW execution.

Once frequency-voltage combinations of faulty behaviour have been identified, the attacker makes use of a manipulated kernel driver that manages to link the victim's thread to a particular kind of kernel, leaving the rest of kernels to other applications. This avoids the threat of possible collateral damage during the attack. In addition, interrupts are disabled during fault injection, which allows circumventing any possible context switching.

A representation of the attack is depicted in Figure 10. The attack requires some preparation: it starts with clearing out any cache residue, since in the following phases of the attack a cache-based profile is used to signal the start of the victim's execution (step 1). Then, the attacker monitors the victim's code execution by inspecting certain execution points, called "Timing Anchor" point, especially in the instant prior to the execution of the target code where the fault is to be injected (steps 2-3). There are some cases where the accuracy of the Timing Anchor is not good enough, thus it is necessary to achieve a more precise synchronization of the attack. To fine-tune the accuracy, the attacking thread remains in a loop for a period of time, after which it will proceed to the next step of the process (step 4). Note that a distinguishing feature of this attack is that the frequency of the victim's CPU kernel undergoes changes while the attack is taking place, raising the frequency value to a specified one and over a specified period – and then restoring normal conditions (steps 5-6).

Using this attack technique, it was possible to unveil the secret key of a previously manipulated implementation of AES executed in the Secure World. The implementation consisted of a simple decryption tool that received encrypted messages as input and returned the plaintext, decrypted with a stored secret key. The attacker was able to unveil the AES secret key by inducing various glitches during the AES decryption phase and applying differential fault analysis (DFA) attack.

The authors also showed a second type of attacks on TZ with CLKscrew, which they call *self-signed application loading*. In this case, CLKscrew can be used to modify the RSA signature chain of firmware images in TZ, which is the method used for verifying their authenticity. Firmware images to be updated contain the updated code, a signature of the firmware's hash to maintain its integrity, and a certificate chain. During the upgrade process, a verification of the signature is performed on the hash of the new firmware to be uploaded, together with a secret key linked to the hardware (this key is stored in the Secure World). Using CLKscrew, the authors are able to crack the signature process to force it to produce a hash that is identical to the hash of a different firmware. Consequently, the verification mechanism accepts to install an illegitimate firmware as if it were correctly signed by a trusted entity.

6.2. PlunderVolt

Plundervolt Murdock et al. (2020b) relies on the inducing changes to the voltage received by the processor, causing the program to change its intended execution path. Pundervolt exploits the lack of a stable power supply voltage.

Plundervolt circumvents the protection limits of the TEE memory encryption engine by abusing an undocumented voltage scaling interface, which allows privileged software adversaries to lower the tension and cause predictable failures in the SW. With this technique, the theft of secrets is achieved, even in the presence of memory encryption technology.

For instance, Plundervolt can break the integrity and (indirectly) the confidentiality of Intel SGX Murdock et al. (2020a). Indeed, as a consequence of Plundervolt it is possible to break the processorüs instruction set specification, making it possible to successfully attack bug-free code, tested code and even formally verified code. Unlike other Intel SGX attacks, which abused architectural design flaws to break the confidentiality of enclave secrets, the authors demonstrated that even the integrity of seemingly secure enclave computations can no longer be trusted. The authors in addition to succeeding in breaking cryptographic code show how Plundervolt can be used to induce memory safety vulnerabilities into bug-free code.

[10] Hardware can include voltage/frequency regulators, which contain a phase-locked loop (PLL) circuit that generates a synchronous and adjustable clock signal for the digital components.

6.3. Platypus Attack

Platypus Lipp et al. (2021) is based on exploiting the mechanism of accessing the interface of Intel's RAPL - Running Average Power Limit, which reveals information about power consumption. The weakness lies in that any user of the system can access this interface.

Platypus shows that by performing a statistical study with a certain number of evaluated data, it is possible to appreciate and identify variations in energy consumption. By assigning different Hamming weights to what is loaded into memory, different code instructions can be identified. This makes it possible to monitor the control flow of applications, which is very valuable to a potential attacker.

Using Platypus, an attacker has also the ability to deduce sensitive information such as secret keys. The authors show how a potential attacker, who starts from an unprivileged state, is capable of obtaining AES new instructions (AES-NI) keys from Intel SGX and the Linux kernel, infer secret instruction streams, break the randomisation of the kernel address space layout (KASLR) and finally achieve the establishment of a time-independent covert channel.

6.4. VoltJockey

VoltJockey Qiu et al. (2019a) is an attack based on dynamic voltage and frequency scaling (DVFS). This attack differs from others (e.g. CLKscrew) in that it performs manipulations on voltages instead of frequencies. This allows the generation of failures in the target hardware. VoltJockey is notable for being more stealthy and therefore more difficult to avoid than similar attacks such as CLKscrew. Some authors Qiu et al. (2019a); Qui et al. (2020) have shown how TrustZone's AES key and RSA-based authentication can be cracked on an Android smartphone using VoltJockey. This is one of the most effective attacks for obtaining protected TrustZone credentials.

VoltJockey is an attack on TrustZone based on hardware flaws using software-controlled voltage manipulation. It exploits the DVFS voltage management vulnerability. In Qiu et al. (2019a); Qui et al. (2020) the authors implement VoltJockey on an ARM-based Krait multicore processor, whose core frequencies can be different but the processor voltage is controlled by a shared hardware regulator. The Trust-Zone protected AES key is achieved and thus guide the RSA-based signature verification to obtain the target plaintexts. An implementation of VoltJockey was used to break Intel SGX in Qiu et al. (2019b) and in an advance scaling based fault injection Qiu et al. (2020).

6.5. Rowhammer

The Rowhammer attack Lipp (2016) exploits the particular design of some modern DRAM memory in which memory cells are getting closer and closer. This complicates isolation and makes DRAM cell capacitors sensitive to electrical interference thus potentially leading to memory corruption. As such, the repeated access to a row of memory can cause bit flipping (shifts from 0 to 1 and vice versa) in adjacent rows.

Consequently, Rowhammer takes advantage of this isolation problem to affect the RAM rows storing TrustZone data, even bypassing the NS bit protection mechanism. The authors of the attack, from Carnegie Mellon University and Intel, tested this phenomenon on Intel and AMD systems using a program that generates multiple accesses to DRAM memory. They managed to cause errors in most of the DRAM modules tested (110 out of 129) from three major manufacturers.

6.6. BADFET

In recent years, electromagnetic fault injection (EMFI) attacks are becoming a major threat. This is as a consequence of the massive increase in CPU speed and the reduction of the size of the components, which hinders other types of injection attacks.

BADFET Cui and Housley (2017) is based on *second-order* EMFI attacks, which do not target the CPU but other components of the system. In fact, this attack can be applied to any arbitrary component (such as memory, buses, controllers, etc.) that the processor makes use of during sensitive operations. This approach can significantly reduce the temporal and spatial resolution requirements of the hardware needed for EMFI injection.

The attack consists of two steps. During startup, BADFET applies electromagnetic radiation on the system's RAM memory. These memory-induced failures trigger a condition that exposes the uBoot's debugging Command Line Interface (CLI) to attackers, which enables to switch between the Normal and Secure worlds. Once the CLI is available, during the second step, a buffer overflow-based vulnerability is exploited in the SW. This allows attackers to obtain write, execute and read privileges and, as a result, the attacker achieves a new CLI that is capable to fully execute commands in the SW.

7. Micro-architectural attacks

The last category of this taxonomy include attacks targeting micro-architectural elements. This section summarizes the attacks considered as micro-architectural as they have been applied to TEEs. These attacks focus on micro-architectural details as caches, Branch Target Buffer (BTB) unit, etc.

7.1. Cache timing attacks

As previously mentioned when the architecture of the TZ was described, cache memory is shared between SW/NW. Since the secure parts of the cache are not accessible from the NW, bidding for the use of the cache lines does not take place, and therefore a substantial improvement in system performance is achieved. However, information leakage through caches is an open avenue for attackers. These attacks are usually performed by extracting hardware information such as timing computations, cache access attempts and even the sound released while the computation is taking place.

In a *cache timing attack*, an adversary is capable of inferring secrets from the secure world by monitoring accesses made by the victim in a shared memory. Generally speaking, a cache timing attack has two phases – timing and correlation, and is typically used for leaking cryptographic keys or another sensitive information. During the timing phase, the attacker sends raw data to a specific (cryptographic) function to measure the time spent on each encryption. The total execution time can be highly affected by the number of cache hits and misses produced during the execution. Once the attacker gathers enough measurements, he is able to match the entries with the execution times, and thus infer the key. These methods rely on active cache manipulation designed to produce data with a higher level of entropy, which in turn results in a fairly smaller data set to perform the attack.

Next, we elaborate on how this type of attack affects TZ with an specific example. The ARM chip is built in such a way that a shared CPU cache is used to improve the performance of data and instructions processing in the SW and NW. This cache integrates a mechanism, known as the TZ NS-bit, dedicated to ensuring separation between the two worlds. Included in this separation are the access rights for the resources available in each world. The operation of this mechanism is simple: the bit is used to tag each cache entry, such that if any NW process attempts to access a SW entry a

miss occurs (Kim et al., 2012). Although this cache tagging mechanism may appear to be secure, recent works have revealed that its design present several flaws that can be exploited using different strategies (Gras et al., 2017; Irazoqui et al., 2015; 2016). Still, a successful implementation of this attack is not trivial among other reasons because the attacker must be able to manipulate the cache in order to monitor the victim's process.

Götzfried et al. (2017) showed a cache-timing attack affecting Intel SGX enclave (Intel, 2014). The authors demonstrated that, in practice, SGX cannot resist its designated attacker model (i.e. attackers gaining root access to the system) when dealing with side-channels. In fact, during the experiments the authors realized that the side-channel attack surface increases significantly in the SGX scenario. This is because without SGX some capabilities are restricted to the kernel. In the presence of Intel SGX the attacker acquire new capabilities, such as the possibility to operate the power management control (PMC).

This type of attacks have also been tested against ARM based CPUs. Weiß et al. (2012) present the implementation of an attack against a *virtualized* ARM system. Based on the conclusions of this work, Spreitzer and Plos (2013) studied the application of this timing attack on different Android smartphones. Later, these authors Spreitzer and Gérard (2014) achieved substantial improvements in the results by reducing the key space. Bogdanov et al. (2010) presented another attack against AES table implementations based on the exploitation of collisions. They used an ARM9 microprocessor for this purpose.

The use of *branch predictor* is another way to implement cache-timing attacks on TrustZone. In the latest processor designs, a component called the branch target buffer unit (BTB) is included. This allows the storage of target addresses obtained from the computation of the forking instructions performed, with subsequent retrieval when the instructions are predicted (Takahashi et al., 2018). As a consequence of BTB being shared between both worlds, it is possible to perform attacks such as Prime+Probe (explained below) to reveal data. The process starts with a priming of the BTB. The victim process is then allowed to start, which will be evict the attacker's BTB entries. Once the attacker acquires control of the execution, he initiates the associated branches in order to detect prediction errors. A relevant aspect in the internal operation of the BTB is related to byte granularity rather than cache line granularity. This enables a new attack vector by significantly increasing the spatial resolution of the probing mechanisms. Using this approach, it is possible to retrieve a private key directly from certain hardware-backed keystores Ryan (2019b). Some examples of memory-based attacks using different techniques are briefly described below.

7.1.1. Prime+Probe

The Prime+Probe attack (Osvik et al., 2006) begins with the attacker filling the cache with data. Subsequently, the attacker monitors how the cache changes while the victim process is running. From the changes detected in the cache, the attacker infers information about the victim's operation and behavior.

From the attacker's perspective, the main advantage of this technique is that there is no need to carry a shared memory map between attacker and victim. This results in a very suitable mechanism for attacking the SW with very few additional resources required.

7.1.2. Evict+Time

This attack (Osvik et al., 2006) is based on the execution time of the victim process. The process is run and then all cache entries that have been used by it are deleted (evicted), in such a way that the execution time is modified in the next execution. The differences between execution times are then analyzed and correlated with all cache changes so as to extract useful information. For example, this type of attack can be launched against a cryptographic algorithm, say AES, to expose the cryptographic material.

7.1.3. Flush(Evict)+Reload

Yarom and Falkner (2014) describe the Flush(Evict)+Reload technique. Flush + Reload works based on an abuse of shared code/data by making use of the clflush cache flush instruction. It is necessary that victim and attacker physically share at least one page of data. This is possible since shared libraries are normally only loaded once physically into memory. Instead, different applications access the same data (physically) since the page tables point to the same physical address. The process is as follows, when the attacker uses the clflush command with an address pointing to this shared data, it is completely flushed from the cache hierarchy. As the data is shared, the attacker can hit on this data in the cache. Repeatedly the attacker empties the shared data with the victims as Fig. 11 depicts, then the attacker remains on standby until the victim executes, at which time it performs the reload of the data. From this moment on, if the attacker gets a cache miss, i.e. the victim has not accessed the data, and therefore has not returned it to the cache. On the other hand, if he gets a cache hit, that is, the victim did. In this way, the attacker can distinguish hits from misses because the memory access time is very different.

The potential of this attack lies in the fact that the attacker can reach a very high level of knowledge of the cached data. As memory is slower than the processor, this fact produces bottlenecks. Recently used lines are stored in the cache, which improves the performance. Since Multi-processors Systems-on-Chip (MPSoCs) components can directly access the hardware information, like communication infrastructure or physical addresses, the Flush+Reload technique on MPSoCs is prone to be implemented in these settings.

7.1.4. Flush+Flush

The Flush+Flush mechanism (Gruss et al., 2016b) could be seen as a variation of the *Flush+Reload* attack implemented in reverse. It begins in a similar way to the one described above: by emptying the cache lines that are shared. Immediately afterwards, the victim program can be executed. The attacker then performs another cache flush while calculating the time taken to perform this flush.

The idea behind this attack is that the time spent in flushing the cache can change depending on the cache lines that have been loaded while the victim was running. This allows the attacker to infer certain information from the victim's process. Although this attack is more complex, this technique has the advantage of going unnoticed more often than previously described ones. The reason is that many attack detection mechanisms rely on the presence of cache misses to identify possible attacks.

7.1.5. Wei' Attack

Weiß et al. (2012) demonstrate that cache timing attacks can bypass virtualization barriers. The experiment made use of replay-resistant authentication by performing all encryption operations in the secure world. The attack targets the authentication scheme, and for this purpose a reduction in the key space is pursued until it can be effectively implemented by brute force.

This attack is structured in two phases: offline and online. During the offline stage, the attacker gathers multiple encryption operations using a known, all-zero key. In the other phase, the attacker's goal is to capture the key that is unknown to him. Once enough synchronization data has been collected, the correlation between the two sets is established, thus obtaining the possible values of each byte of the key. To find the values, a calculation is performed based on a probability threshold. The mechanism is initiated by inserting a value in the list, which contains those pos-

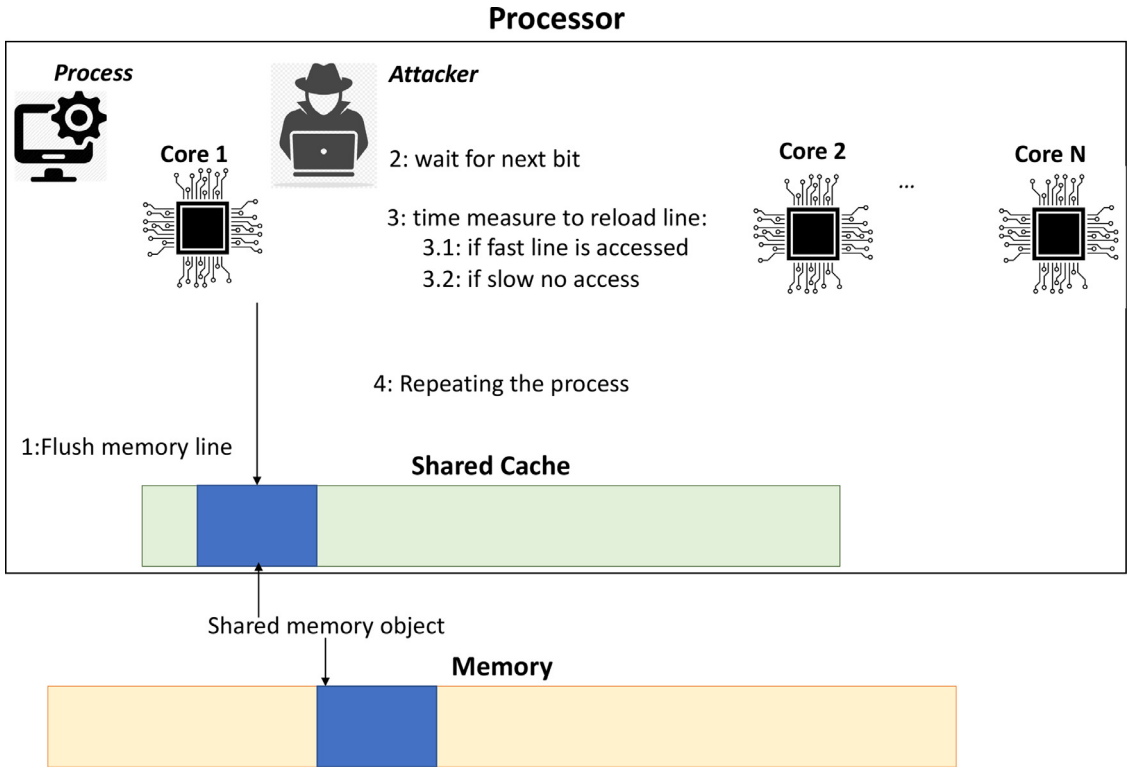

Fig. 11. Flush+Reload attack workflow.

sible values of the key, just at the instant when a byte of the key appears with a probability higher than the established threshold.

This work was developed in 2012 when the TEEs were just beginning to get standardized by GlobalPlatform and deployed in consumer devices. For this reason, rather than on a TEE, Weiß et al. (2012) present an implementation of the attack on virtualized systems. Although this attack was not implemented in TEE, the authors showed that cross-isolation attacks are effective, given both worlds share CPU and cache. This particular implementation was performed on a Beagleboard[11], which is basically an ARM-based development board that integrates an L4 microkernel – which is the virtualization layer. During the experiment, they took measurements of the time spent on each encryption operation using the ARM CCNT register, as well as the total count of CPU clock cycles since the last restart. They took different implementations of the AES to study the weaknesses that appear in general computation and concluded that, to a greater or lesser extent, they were all vulnerable. Two years later, Weiß et al. (2014) reproduced the experiment – but this time in *a multi-core environment* on a development board.

7.1.6. ARMageddon

Lipp et al. (2016) describe the implementation of a cache-timing attack, called ARMageddon, that uses only unprivileged applications and target Android devices based on ARM architectures. To understand the attack we first need to be aware that ARM level 2 caches are *not inclusive* for the most part. This implies that it is not possible to guarantee that there are entries in lower-level cache shared by the CPU cores thus hindering *cross-core attacks*. This is because the last shared cache level is the only way for an attacker to access and modify data from other cores.

The attack is implemented on modern devices employing multi-CPU based designs, namely ARM devices with non-inclusive L2 caches (the last-level ones). A new exploitation of cache coherency protocols and transfers between L1 and L2 is presented, achieving an workaround to the difficulty of last-level cache non-inclusiveness. As mentioned above, devices with multiple CPUs do not share a common cache between them. However, the protocols used to retrieve line cache entries coming from different CPUs follow coherence rules that allow exploiting certain attacks more effectively. Among the different policies, we find LRU (least-recently used) implemented by Intel or a pseudo-LRU variant by ARM processors.

As ARM CPUs make use of a pseudo-random cache replacement policy, this makes it difficult for the attacker to predict which line to replace. This technique lowers overall attack performance because it reduces the effects of erroneous prediction of replaced lines. In this work, the authors present results of the implementation of ARMageddon on three different devices, each one with particular strategies for accurate unprivileged cache timing in the attacks.

7.2. Separation barrier

These are focused on exploiting the separation barrier and since it is a micro-architectural element, they belong to this category.

7.2.1. Prime and count

The Prime and Count technique Cho et al. (2018) aims to reduce the noise caused by TZ's own inter-world switching mechanism and the pseudo-random cache replacement policies. On its own it cannot be used to snoop into the secure world, however, it provides a proof of the existence of a side channel that can be established between both NW and SW. This attack has been used as a precursor of more complex attacks such as privilege escalation.

The technique is implemented with a sender in charge of writing data to the cache to signal a message to a receiver process. There are two strategies for implementing this attack depending

[11] http://beagleboard.org/.

on whether they are applied to single-core or multi-core architectures. The difference lies mainly in the cache level to which it is applied, as the L1 cache is available to each CPU core, without being shared by other cores. Unlike the L2 cache which, being larger, can be shared among all the cores.

In the first phase of the *single-core attack*, the receiver primes the L1 cache filling it entirely. Then, the sender application, which is running in the SW, then takes control and writes new data to the L1 cache for signaling the message. Finally, control is switched back to the NW which can learn how many cache lines have been modified by the sender. After each sender - receiver interactions a piece of the message is covertly transmitted.

In the case of a *multi-core attack*, the difference is that during the first stage both L1 and L2 caches are primed and therefore invalidated. Meanwhile, the sender only writes to the L2 cache. Clearly, this attack is more difficult to implement because the L2 is a global cache that can be accessed by applications executed in parallel by other cores. Nevertheless, messages can be encoded taking into account the accesses made by other process and eliminate noise that may appear in the channel by introducing error correction codes.

7.2.2. TruSpy

The TruSpy technique Zhang et al. (2016b) could be considered the first proof-of-concept of *"cross world"* attacks. A cross-world attack can be defined as one capable of breaking the isolation between the normal and secure worlds. The authors present two types cross-world attacks, one of which requires kernel privileges and is easier to implement, and the other one which can be successful even with user-space privileges alone, but is more difficult to execute.

In the privileged attack, the adversary has access to both the virtual-to-physical memory mapping and the Performance Monitor Unit (PMU), which offers statistics on the operations of the processor and memory. This allows him to perform cache priming and cache probing with ease. The other attack only requires user-space privileges, but is more difficult to execute because it lacks access to the previously mentioned resources. Memory sharing between the attacker and victim processes is not a requirement for the implementation of either attack, since they are based on the Prime+Probe technique.

The attack has five stages, as it is shown in Fig. 12. In step 1, the attacker finds memory addresses for cache priming, if the virtual address space is mapped to the cache sets. Once identified, the attacker performs the priming of the cache (step 2). The victim process then takes control and changes the state of the cache during its execution (step 3). Finally, the attacker probes the cache for cache misses (step 4) thereby identifying the lines that have been modified by the victim. The difference between both states is stored, and returns to the second step to keep iterating – until a sufficient amount of data is recorded. Finally, in step 5, the collected data is analyzed in order to reveal secret information from the victim running in the secure world.

7.3. Speculative execution attacks

Speculative attacks exploit a feature present in most modern processors, called speculative execution, to leak confidential information. In speculative execution, the CPU attempts to anticipate the processing of certain future instructions, which may or may not be necessary, to optimize code execution. In case these instructions are eventually not necessary, the changes are reversed and the results ignored. However, not all changes are reverted (e.g. cache changes) and leave traces that can reveal sensitive data to attackers. Since speculative attacks are mainly focused on fault injection and cache timing techniques, they have been included in Section 7.

This category of attacks has become increasingly prevalent lately and they can hinder the isolation guarantees of TEEs in different implementations. Some important examples are Meltdown (Lipp et al., 2018) and Spectre (Kocher et al., 2019). The basic idea behind Spectre and its different variants is to trick the processor into speculatively executing sequences of instructions that should not have been executed under normal circumstances. By influencing which instructions are speculatively executed, sensitive information is leaked from the victim's memory address space. Kocher et al. (2019) demonstrate the feasibility of Spectre attacks across security domains from both unprivileged native code and portable JavaScript code.

A variant of Spectre for Intel SGX is known as Sgxpectre Chen et al. (2019a). Sgxpectre bases its attack on misusing the branch prediction unit (BPU) to cause the victim to run certain secret leakage instructions. BPU are certain hardware components that collaborate in the prediction of conditional branches, indirect jumps and calls, and function returns. To do so, the attacker must be able to induce speculative access of unwanted data by deviating the execution branch (within the same kernel) beforehand. This enables the possible execution of malicious code on another thread from the main domain – it could even be the same thread – if the execution of the domain itself can be interrupted and the BPU contaminated.

Meltdown Lipp et al. (2020) is a software-based attack that can be considered the precursor to the attacks included in Section 7.4. It exploits out-of-order execution (a type of speculative execution) to allow an unprivileged adversary to read the memory of other processes or virtual machines, which may include personal data and passwords. Meltdown does not require the adversary to exploit any existing vulnerability in the software and is operating system independent.

Meltdown consists of three steps. In the first step, the attack loads the contents of a memory location (inaccessible to the attacker) into a CPU register. This will eventually cause an unauthorized access exception rolling back the execution. In the second step, the attacker defines a sequence of instructions, by taking advantage of out of order execution, that are capable of accessing the secret data loaded into the register. Before the register is cleared due to the exception, this transient instruction sequence will encode the secret into the micro-architectural cache state using the Flush+Reload technique, although it would also be possible to use other similar techniques. In the last step, the attacker recovers the secret data from the cache state. By repeatedly performing these three steps over different memory locations, the attacker can retrieve the entire physical memory.

These attacks have been successfully implemented in the most widespread TEE implementations such as Intel SGX (Brasser et al., 2017; Götzfried et al., 2017; Intel, 2014; Moghimi et al., 2017; Schwarz et al., 2017) and ARM TZ (Lipp et al., 2016; Zhang et al., 2016b).

In addition to Meltdown and Spectre there are other attacks that can be considered speculative. These include the exploitation of the lack of prediction of conditional forks, the poisoning of direct forks, as well as other combinations. Instruction timing can also be exploited, since instructions whose timing depends on operand values can leak information about operands without necessarily involving caches. The efficacy of this type of attacks to infer private information (data, operations) has been proven, as well as the ability to circumvent the barriers imposed by address space layout randomization (ASLR) (Gras et al., 2017; Gruss et al., 2016a).

Finally, another interesting attack vector is due to the inherent leakage caused by latency differences between cache inputs and outputs. This allows to infer keystroke behavior (Gruss

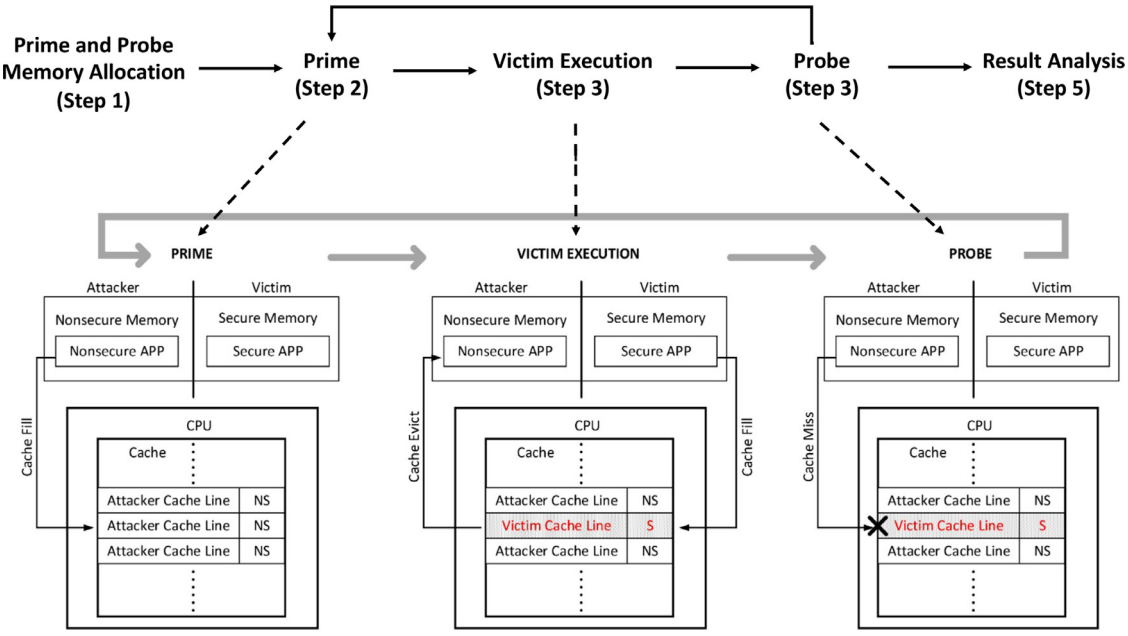

Fig. 12. TruSpy attack workflow. Based on Zhang et al. (2016b).

et al., 2016b; 2015), and even both symmetric AES (Bonneau and Mironov, 2006; Irazoqui et al., 2015) and asymmetric RSA (Liu et al., 2015; Zhang et al., 2012) keys.

7.4. Out-of-order execution attacks

Out-of-order execution is a subtype of speculative execution that allows instructions to be executed as long as the necessary resources to do so are available, even if they do not follow the normal sequence of code execution. Out-of-order attacks exploit the fact that the memory used for the execution of these transient instructions can be accessed by other processes before being freed.

Foreshadow (Weisse et al., 2018), Micro-architectural Data Sampling (Minkin et al., 2019; Schwarz et al., 2019; Van Schaik et al., 2019) and Load Value Injection (LVI) (Van Bulck et al., 2020) are attacks that belong to this category.

7.4.1. Foreshadow attack

Until the publication of Foreshadow (Van Bulck et al., 2018), Intel SGX was thought to be resistant to speculative execution attacks. However, Foreshadow demonstrated it was possible to read the memory protected by SGX and even extract the machine's private attestation key.

Intel analyzed Foreshadow in an attempt to prevent the cause of the attack and they realized that two additional attacks were possible. These attacks, which are referred to as Foreshadow-NG (next generation) Weisse et al. (2018), allow an adversary to read any information contained in the L1 cache. This includes information from other virtual machines running on cloud infrastructures.

Moreover, Foreshadow-NG might be able to bypass some of the countermeasures that were created to prevent other types of speculative attacks, such as Meltdown and Spectre.

7.4.2. Micro-architectural data sampling attack

Micro-architectural Data Sampling (MDS) vulnerabilities allow adversaries to exfiltrate data from different CPU internal buffers, such as the Store Buffer and the (Line) Fill Buffer. They are called *sampling* attacks because the adversary retrieves data being used by another process but has no control over the memory positions the victim is accessing. This is similar to sniffing CPU buffers.

Using this type of attacks, various researchers were able to access the memory of Intel SGX (Minkin et al., 2019; Schwarz et al., 2019; Van Schaik et al., 2019). In addition, some authors (Ragab et al., 2021) showed that, despite existing mitigations against speculative execution attacks, existing CPUs are inadequately protected and sensitive data can still be leaked.

Notable attacks within this category are the Rogue In-Flight Data Load (RIDL) (Van Schaik et al., 2019), Fallout (Canella et al., 2019a) and ZombieLoad (Schwarz et al., 2019), which are described in more detail below.

Rogue In-Flight Data Load RIDL (Van Schaik et al., 2019) can leak data from a victim process even if that process is not speculating (e.g., due to Spectre mitigations) and requires no control over address translation data structures. Attackers running arbitrary unprivileged code manage to leak information across arbitrary security boundaries (JavaScript sandbox, process, kernel, VM, SGX, etc.). In short, RIDL allows the attacker to listen in on all communication between CPU components.

As with other attacks in this category, it originates from optimizations that cause the CPU to serve speculative loads. In this paper, authors present several exploits that allow data leakage by the following steps. First, the victim code loads/stores data, the CPU performs the load/store through internal buffers [12]. Next, the attacker performs a load and the processor uses data from the buffers speculatively. Finally, it makes use of the speculatively loaded data in the buffer to extract the secret value.

Fallout Fallout (Canella et al., 2019a) takes advantage of the internal Store Buffer, which is used to track pending store operations. This attack allows programs with no special privileges to read data recently written by the kernel, as well as to de-randomize the Kernel Address Space Layout Randomization (KASLR).

When a code writes a value to memory, before getting exclusive access to the address, the processor maps the virtual address of the destination to a physical address. However, instead of waiting for the computation to finish, the processor inserts the value and the address into the Store buffer and continues the

[12] The paper primarily focuses on (Line) Fill Buffers, but other buffers can be used such as load ports and store buffers.

execution of the program. The Store buffer then resolves the address and stores the data. The processor must control that obsolete values are not loaded, which is the purpose of the Write Transient Forwarding (WTF) instruction optimization. WTF marks the load as faulty and forwards the partially matched store value, which should not be forwarded. This behavior is exploited by Fallout to obtain the value that WTF sends. As in other cases, it uses a side channel (Flush+Reload) to exfilter the value.

ZombieLoad ZombieLoad (Schwarz et al., 2019) is a transient execution attack that takes advantage of the Fill Buffer present in Intel CPUs. This buffer, which is used during load instructions, retain data from memory load requests until new ones overwrite them. Moreover, it is shared among the logical cores of a physical CPU. Therefore, a malicious thread running on a logical core could access the data of another thread running on a different logical core within the same physical CPU, even if the threads belong to completely different applications.

Under certain conditions, typically a faulty load operation due to erroneous data, speculative execution allows to obtain other data not related to the load memory address from the Fill Buffer. These data can be finally extracted by some sort of side channel, such as those provided by the cache subsystem.

7.4.3. Load value injection attack

Bulck et al. (Van Bulck et al., 2020) present the Load Value Injection (LVI) attack, which is based on the injection of erroneous data into the memory of a victim's program. Once the application detects in-memory data is incorrect, the execution is rolled back. Before the mistake is detected, during this short period of time, an attacker can access the data from the victim, which may include sensitive information from Intel SGX. A limitation of LVI attacks is that the adversary cannot always control certain conditions, such as when a failure occurs, as they take place in the victim's environment.

Unfortunately, LVI is much more difficult to mitigate than previous attacks as it requires compilation patches that insert instructions to limit speculative execution after every potentially vulnerable instruction. This impedes the processor to optimize its execution (i.e., the pipeline is serialized) resulting in a significant decrease of Intel SGX computation performance – up to nearly 20 times slower.

Although the proof-of-concept implementation of the attack targets Intel SGX, the authors argue that LVI attacks are not unique to this enclave but the necessary conditions are harder to be met.

8. Countermeasures

A number of attacks for different TEE implementations have been described so far. To complete the picture, we also review different countermeasures that have appeared in recent years. Since these countermeasures have appeared as a response to attacks, we present them following the proposed taxonomy.

8.1. Countermeasures to software-based attacks

First, we describe the most relevant countermeasures against software-based attacks to mitigate or reduce certain security issues of TEE components and applications.

TEE master key extraction is possible because the disk encryption is based on a software key derived from information stored inside the TrustZone kernel memory. Since the key is inside the software, attackers can extract this key. A countermeasure for this is the use of a secure element with hardware-bound key functionality, such as TPM.

Regarding validation failures, most commercial TEE systems are written in C, which does not provide memory protection mechanisms. As a result, developers introduce memory violation errors, which in turn cause validation failures. As a solution to this, in certain TEE systems such as TLR (Santos et al., 2011) applications are interpreted with.NET managed code – similar to a Java Virtual Machine (JVM). Even if this introduces an extra overhead in the execution of the applications, this approach can be of great help, as it provides certain tools (e.g. run-time memory checks and rubbish collection) that reduce the risk of validation failures.

Other approaches follow the idea of using secure programming languages for developing sensitive components that will be deployed in TrustZone ecosystems. Among them, RustZone (Evenchick, 2018) can be highlighted. RustZone provides an extension of OP-TEE that enables developing applications using the Rust programming language. This language provides memory and thread safety, which help to avoid validation errors and some concurrency errors responsible for application software crashes.

Implementation errors caused by a lack of consistency between the expected requirements of a software component and its actual implementation are often encountered. Techniques such as model checking, symbolic execution and formal methods can be very useful to avoid these mismatches, and are very effective in ensuring that an implementation meets the proposed requirements. Although the application of these methodologies is generally not trivial, significant progress has been made in the use of formal verification techniques to analyze the robustness of TEE components. There are very interesting proposals such as Komodo (Ferraiuolo et al., 2017), which consists of a monitor that implements the Intel SGX enclaves specification, and the memory manager known as MIPE (Chang et al., 2017).

On the other hand, there are different tools for *malware detection*. This is important to consider, as many attacks that target TEEs are deployed as malware. Among such tools, Andrubis (Weichselbaum et al., 2014) combines static and dynamic analysis techniques using unsupervised learning (with clustering). Tools like DroidClone (Alam et al., 2016; Alam and Sogukpinar, 2020) exposes similar code segments ("code clones") in a very accurate manner for the detection of malware variants, while other approaches, such as DIFT (Andriatsimandefitra and Tong, 2015), focus on monitoring the information flow for malware detection by tracking selected data during the application execution. There are other lighter alternatives such as ThinAV (Jarabek et al., 2012), which combines a low footprint on an Android device with the ability to leverage various anti-malware services in the cloud.

There are other software-based countermeasures that focuses on recognition and detection using machine learning techniques. For example, in (Soviany et al., 2018) the authors describe a whole crypto-mining detection and recognition methodology based on machine learning. Another approach, based on a structured heterogeneous information network (HIN), known as Hindroid, is presented by Hou et al. (2017). Authors integrate several machine learning-based tasks with some optimisations that are performed at various processing stages, including the multi-core approach. In addition, techniques such as DroidDream (Kim et al., 2016) can be used for malware family identification, based on malware detection work with dynamic analysis on real devices.

Finally, there are other solutions that pursue to empower the applications themselves such as PrOS (Kwon et al., 2019) and TEEv (Li et al., 2019), which provide a minimalist hypervisor implementation on the SW. This allows applications to work on multiple guest OSs in a secure and isolated way.

8.2. Architecture-based countermeasures

In this section some of the countermeasures already proposed in the literature against architecture-based or micro-architectural

attacks are presented. These countermeasures are presented together, because in many cases they are shared.

Isolation between worlds is a source of different security threats. Several mechanisms have emerged that aim to overcome the existing limitations in the main TEE. Examples of such limitations are the absence or weakness in authentication when accessing TEE resources from the NW and shared memory which as we have argued is potentially insecure for data exchange within the channel. A technique commonly used to reduce the attack surface is known as *multi-isolated environments*. They are different from traditional sandboxes and are particularly useful for protecting TEE systems from a wide variety of attacks. They make it possible to contain the scope of damage that can be caused by a security breach by increasing the granularity of isolation between different TEE components. They also allow limiting the code that can be executed, which directly reduces the possibility of privilege escalation attacks. This technique has been implemented in different ways. Some focus on the creation of compartments of the NW itself, with a strong isolation, in which applications would be assigned. Others focus on protecting the applications, with approaches such as Sanctuary (Brasser et al., 2019) and TrustICE (Sun et al., 2015b) leveraging different features of TZASC. There are mechanisms that explore the implementation of environment isolation with hardware virtualization extensions available in NW (NS-EL2) such as PrivateZone (Jang et al., 2016), OSP (Cho et al., 2016), and vTZ (Hua et al., 2017).

As seen in this paper, some architectural attacks occur because TAs in Trustonic TEE cannot physically read/write to physical memory – this task is performed by *specific driver TAs*. If an application needs to make use of shared memory, it will have to issue a request to the controller. Samsung's TZ, known as TIMA, uses a similar approach, where only the application controller can allocate physical memory – thus mitigating risk. TIMA makes use of a whitelist that limits the applications that can query the application controller. Although this mechanism provides additional security guarantees, it is still not sufficient: the attacker could target the whitelisted applications to successfully compromise the system.

Some implementations aim to mitigate this potential source of vulnerabilities using an architectural design based on *microkernel*, which restricts the execution of drivers to the SW user space only. This approach is being integrated into NVIDIA and Trustonic implementations. Other companies, such as Huawei, focus on introducing a new task to control the TEE lifecycle. To do this, it creates a TEE with certain privileges, which it calls GlobalTask. Another measure is the inclusion of a single non-secure port to perform the centralized connection of all memory-mapped non-sensitive IP cores. This allows their operation to be controlled by memory protection mechanisms such as SMMU (Marchand et al., 2017). Other measures focus on preventing the misuse of hardware voltage regulators, which is solved by applying specific hardware and software performance limiters via drivers Tang et al. (2017).

SeCReT (Jang et al., 2015) provides a *session key* for applications running in the NW to encrypt messages. In more detail, SeCReT proposes a number of input and output mode changes to the kernel, including the elimination of the memory key during kernel mode execution, pursuing the protection of the NW kernel session key – which is untrusted. In the case of TFence (Jang and Kang, 2018), a non-fully privileged process (a shielded part of the NW application process) communicates directly with the TEE, further eradicating this kernel dependency. There are alternatives that implement exclusive shared memory such as TTEEv, Sanctuary and PrivateZone. The latter allows communication, but without memory sharing, since it implements it by means of data copies. There are other alternatives that avoid BOOMERAN attacks by sanitizing the Machiry et al. pointers. In fact, Machiry et al. were in contact throughout the process with the TEE suppliers themselves, with the ultimate goal of being able to develop the relevant corrections for their environments.

COLONY (Xia et al., 2021) proposes a new architecture in which each instance of the design ("COLONY") has grants to *access only* the necessary *system-level semantics*. This approach relies on a secure monitor to implement isolation and capability management. Despite the advantages provided by this approach, which assumes that hardware components are completely reliable, the protection provided is not sufficient – as demonstrated in Section 6. In fact, a compromised "COLONY" can attack the caller by returning a malicious value (Checkoway and Shacham, 2013). Furthermore, COLONY does not take into account side-channel attacks, hardware-based attacks and DoS attacks.

Other solutions use particular techniques such as Keystone (Lee et al., 2020), which aims at *isolating memory* with a *programmable layer* below untrusted components. Keystone provides protection to the TEE against some attacks (Mapping, Syscall Tampering and Side-channel), as well as protection to the host OS against TEE attacks. It also provides protection to the secure monitor, since the entire memory of the secure monitor is isolated and therefore not reachable for all TEEs. In fact, it is not even accesible for OS hosts. EnclaveDom (Melara et al., 2019), implemented in Intel SGX, is a system that provides a separation of privileges for larger TEE applications. The enclave is divided by memory regions which are labeled, and establishes a set of access rules per region with some granularity of the individual functions in the enclave.

Sanctuary (Brasser et al., 2019) proposes an extension of TZ with the use of *user-space enclaves*. This approach is designed to provide hardware-enforced bidirectional isolation, without the need to trust or veto the code of authors called Sanctuary Applications (SAs), since a malicious SA should not be more privileged than normal user space applications. Through bus identity filtering and some additional architectural changes, Sanctuary achieves parallel isolation of individual CPU cores. This allows sensitive code to run without affecting the user experience and with fairly negligible latency in benchmarks.

Many of the existing weaknesses in memory protection of TEEs can be addressed by mechanisms in major operating systems. Still, note that some commercial TEEs provide stronger security mechanisms, either by implementing measures against specific attacks such as *cold boot attacks*, or by integrating tools to provide additional protection such as *memory encryption* (e.g. Intel SGX provides *memory encryption*, yet TrustZone does not provide integrated support for it on the chip itself). Other solutions, such as CaSE (Zhang et al., 2016b), allow applications to run from the cache, thus ensuring that their state remains properly encrypted when writing back to main memory. Also, Ginseng (Yun and Zhong, 2019) performs *variable protection* by tagging the application programmer as "sensitive". Therefore, its information is encrypted at runtime while stored at the CPU registers, thus no unencrypted data will be stored in memory.

Regarding the integrity of the TEE, commercial TEEs have attempted to address this weakness by making use of a *secure boot confidence* to preserve TEE image integrity. Nevertheless, we highlight that only with this mechanism it is not possible for an application client to verify the identity and integrity of both the application binaries and the TEE. For this reason, some of the commercial implementations of TEEs provide certain extra trust primitives. The use of techniques such as remote attestation and sealed storage can be useful in providing such assurances. Thus, TLR (Santos et al., 2011) includes a sealed storage mechanism to protect data from each other by linking them to specific hash values in the TEE-App software stack. Komodo (Ferraiuolo et al., 2017) describes the implementations of the sealed key storage and remote attestation security protocols, as it appeared in the original SGX enclave specification.

Other strategies include *pre-venting the cache side channels* performed by implementing cryptographic algorithms in software (Guanciale et al., 2016; Lipp et al., 2016; Ryan, 2019a; Zhang et al., 2016b) or in specific hardware (e.g., as is the case with specific instructions in ARM such as AESD and AESE) (Lipp et al., 2016) to prevent information leaks in operations. Besides, implementing a reduction of the attack surface by seeking the reduction of the Trusted Computing Base (TCB) (Ying et al., 2019). Truz et al. present as a novelty a proposal based on the use of what they call the delegation model. This model is based on the reuse of almost the entire OS user interface stack in the NW. In this way, they manage to protect the user interface only as a two-dimensional surface, and manage to reduce the size of the TCB considerably.

8.3. Memory protection mechanisms

8.3.1. Lack of address space layout randomisation

Whether due to the lack of Address space layout randomisation (ALSR) implementations, or the poor implementation of existing ones, the fact is that this is an architectural flaw shared by the vast majority of existing TEEs.

Implementations such as OP-TEE Brand, NVIDIA and Huawei do not provide any ALSR mechanism. In Qualcomm's case, an ASLR is provided for all applications, but only makes use of a small physical memory area where the application code is loaded, so that in a small space (about 100MB) all applications are sequentially hosted. It is desirable to achieve high entropy to avoid failures, although in the case of Qualcomm TEE its ALSR is 9 bits, a number that is not enough to provide high entropy.

Despite ASLR, the attacker can be able to figure out where to read and where to write, so other mechanisms are needed. In Section 7.1.6, the insertion of noise while taking measurements of the cache during the attack is described. Other strategies, such as (Lipp et al., 2020), focus on disabling the path predictor if an attempt to exploit the path predictor occurs, and compare the labels of all routes again. Still, so far there is no documented evidence that AMD processors support such advanced strategies in hardware, or even that there is any OS interface for this purpose.

8.3.2. Other memory protection mechanisms

Current OSs integrate memory protection mechanisms such as Guard pages (GP), Stack Cookies (SC) or Execution protection (XP). GPs are used to define the boundaries of the mutable data segments for each process. In other words, it defines the stack, heap and global data in order to avoid a potential attacker from trying to perform an attack based on an overflow of one segment with the aim of corrupting another and resulting in a failure. SC are unique values used for stack smashing detection to allow aborting a running program. Finally, XP delimits certain memory areas in which programs cannot execute. However, this type of mechanism has repeatedly proven to be insufficient. In fact, not all OS integrate these mechanisms. In the case of Trustonic TEE, it has no SC, and it allocates memory to both the global and the stack from the application data segment without putting GP between them. Qualcomm implements SC with random pointer size, yet GP protection mechanisms are not integrated. The ARM implementation of XP makes use of a bit (WXN) of the SCTLR register. This is used to mark write-capable memory regions as "Execute Never" (XN). Other approaches make use of the GP XN attribute (in those implementations that have it) in order to allocate unpriviledge (UXN) and priviledge (PXN) XN, such as NVIDIA (Corporation, 2015) and Linaro Brand implementations that provide both kernel space and user space.

8.3.3. Speculative attacks protection

We consider the case of Spectre (Koruyeh et al., 2020) to be of particular relevance. Firstly, because of the impact it has had. Secondly because, unlike the attacks that have been carried out based on side channels, Spectre highlights the relevance of covert channels, which have often been forgotten. There are two countermeasures to prevent exploitation of Spectre-PHT: memory fences after branches (Canella et al., 2019b), or constraining the index to a valid range using a bitmask (Canella et al., 2019b; Zhang et al., 2022).

The countermeasure KAISER (Lipp et al., 2020), developed initially to prevent side-channel attacks targeting KASLR, inadvertently protects against Meltdown. KAISER prevents Meltdown to a large extent, thus it is highly recommended to deploy KAISER. Intel (Canella et al., 2019a) has proposed certain hardware countermeasures it built into its latest processors Coffee Lake Refresh i9 CPUs to prevent Meltdown. While they certainly make it difficult to implement these attacks they open the door for other attacks such as Fallout.

Still, there are certain countermeasures that manage to mitigate the impact of the attack to a certain extent. These are focused on partitioning, as proposed Lych et al. in 1992 Lynch et al. (1992), (Liedtke et al., 1997) in 1997 and Shi et al. (2011) 2011. Others are based on flushing, as Osvik et al. (2006) and Guanciale et al. (2016) proposed in 2016 and 2013 respectively. However, we should be aware that state partitioning in the kernel will only be possible with additional hardware support as Maña and Muñoz described in 2006 (Maña and Muñoz, 2006) and Dominster et al. in 2012 (Domnitser et al., 2012).

Hyperrace (Chen et al., 2019b) is an alternative designed to detect speculative execution attacks. The authors of this paper propose a mitigation scheme that requires the support of an untrusted operating system. In fact, this alternative design is certainly capable of verifying the behaviour of the operating system.

9. Open challenges

This section outlines some research challenges and open questions that have to be resolved in order to reach an overall improvement of the security of TEE architectures and specific implementations.

One major challenge in the development of secure TEE-based solutions is the protection of **shared resources** between the normal and the secure world. Although some mechanisms have been devised to protect shared resources (e.g., the NS bit), these are not efficient against some attacks. A particularly serious threat is the exploitation of side channels, which could be applied to transfer data between worlds, or to leak sensitive TA data. Therefore, it is paramount to investigate novel mechanisms capable of diminishing this threat while allowing third-party applications to make use of the security mechanisms included and offered by TEE. In fact, side-channel attacks, especially speculative attacks, are currently a hot topic of research due to the drastic consequences of recent attacks.

The use of **dedicated hardware** is also important for solving some of the limitations or complementing the functionalities of TEEs. Dedicated hardware can be used to improve the levels of entropy achieved by current implementations (e.g., QSEE has a 9-bit ASLR with low entropy) but it can also help to preserve the integrity and confidentiality of sensitive data, such as cryptographic keys from side-channel attacks. However, the integration of TPM-type secure elements has some limitations. Not only the addition of new hardware implies increased cost but also applications need to be prepared to use it correctly. A possible alternative to secure hardware in the protection of side-channels is to restrict the number of applications that are allowed to access to the secure world simultaneously but this would limit the performance of the system. Therefore, an important challenge to solve is to find a technol-

ogy with the security of TPM but with the functionality and cost of TEE.

In the absence of any message protection mechanism in TZ, any attacker with privileges to make direct use of the kernel could issue any custom SMC and fuzz the form. This would allow him to successfully implement a man-in-the-middle (MitM) attack with the aim of discovering flaws in the TEE and then exploiting them. In addition, other sorts of attacks, for example denial-of-service attacks, can also be successfully implemented. In fact, at least in none of the existing TEE implementations, there is no message validation mechanism. In fact, even the Universal Unique Identifier (UUID) is susceptible to replication and could be overridden as a security measure. This implies that the TEE has no choice but to act without certainty, making use of information from the unverified message. For all these reasons, we consider that it is essential to elaborate more in-depth studies on the possible integration of validation mechanisms.

The lack of sufficient **validation mechanisms** in exiting TEE implementations is another open problem that needs to be tackled. On the one hand, no TEE solution implements message validation in terms of authentication and integrity. This implies that the TEE has no choice but to act without certainty with information from unverified messages. This would allow, for example, to successfully implement a denial of service attack or a man-in-the-middle attack. It could be argued that the UUID of the message could be used to verify the legitimacy of function calls but since the UUID is part of the SMC it is susceptible to replication and/or impersonation. On the other hand, there is an insufficient validation of the parameters passed to functions. In fact, this is one of the main causes of several of the software-based attacks presented in previous sections. To prevent them, it is necessary to devise more robust sanitation mechanisms to the parameters received by functions before they are used.

A typical problem of many security systems that also affects most TEE implementations is that they are obscured systems. Most existing implementation **designs are closed** and the result is architectures that are not analyzed by security experts prior to their widespread adoption. This security-by-obscurity approach has proven to be wrong on many occasions. Although this trend may be changing with the recent release of the specification of the Qualcomm TEE secure boot procedure, as well as the TA authentication, we are still far from open designs and architectures.

As the IoT matures and the number of interconnected devices continue to grow it is vitally important to protect these devices, which may be part of critical systems. We envision that some of the **IoT devices** in these systems will incorporate some kind of TEE technology for improved security at a cost not as high as that imposed by other hardware solutions. Indeed, some manufactures already provide solutions that can be fitted into some IoT devices such as Infineon's OPTIGA Trust X Infineon, Microchip Technology's ATECC608A Inc, Maxim Integrated's MAXQ106 Integrated, Trusted Objects' TO136 Objects, NXP Semiconductors' proposals SE050 (Semiconductors, 2021) and A71CH (Semiconductors, 2018). Therefore, the research community should investigate how to take advantage of these solutions to establish trust relationships between devices, how these are affected by the integration of different TEE implementations, and so on.

In general, there is an urgent need for **security frameworks** that allow security experts to assess TEE implementations and the code running in them. In fact, the code to be executed inside the TEE is prone to contain vulnerabilities, which can be used to compose attack vectors to corrupt the TEE, compromising the entire system. Security frameworks should help to analyze and verify the security of the code, the appropriateness of the protection mechanisms among trusted environments, in addition to providing methods for monitoring and detecting compromised TEEs and mechanisms for recovering from attacks.

Recall that any application has access to all the resources that a trusted application has. Therefore, an attacker could modify the legitimate OS kernel of a device by exploiting the memory mapping and writing capabilities of the SW and, as a result, the kernel would be infected even if there is no vulnerability in the NW kernel itself. For example, neither QSEE or TrustonIC provide a security mechanism that enables the separation of different memory segments and controls possible heap overflows between different segments.

10. Conclusion

TEE development have been a very prolific field of research and innovation in the last few years. Undoubtedly, this technology provides an improved level of protection during the execution of third-party applications. However, evidence has shown that it has many shortcomings in terms of security.

Throughout this paper, we have presented and analyzed a vast myriad of attacks that can be launched against TEE. These include software-based attacks, side-channel attacks and (micro-)architectural attacks. Although some of these attacks are theoretical, many of them can be realized and have been exploited in practice. What is worse, countermeasures have only been developed for some of them.

In general, we can state that despite the widespread adoption of these technologies, especially in the mobile sector, this is still an immature technology yet with much potential. Much of their problems are due to the fact that their architecture is software-based, resulting in faulty implementations and poor protection against hardware-based attacks. Combining this technology with dedicated secure hardware to complement its security features may be the way forward.

TrustZone, and the various implementations of TEEs that utilize it, are seen as the optimal security providing mechanism in mobile devices, and it is used to provide a vast array of integrity and confidentiality functionalities to the platform. Nevertheless, cryptographic primitives capable of providing the appropriate root of trust to the persistent sealing and attestation mechanisms are not included.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Antonio Muñoz: Conceptualization, Methodology, Writing – original draft. **Ruben Ríos:** Conceptualization, Methodology, Writing – review & editing. **Rodrigo Román:** Methodology, Writing – review & editing. **Javier López:** Supervision.

Data availability

No data was used for the research described in the article.

Acknowledgements

This work has been partially supported by the Spanish Ministry of Science and Innovation through the SecureEDGE project (PID2019-110565RB-I00), and by the by the Andalusian FEDER 2014–2020 Program through the SAVE project (PY18-3724).

References

Ahmad, Z., Francis, L., Ahmed, T., Lobodzinski, C., Audsin, D., Jiang, P., 2013. Enhancing the security of mobile applications by using tee and (u) sim. In: 2013 IEEE 10th International Conference on Ubiquitous Intelligence and Computing and 2013 IEEE 10th International Conference on Autonomic and Trusted Computing. IEEE, pp. 575–582.

Alam, S., Riley, R., Sogukpinar, I., Carkaci, N., 2016. Droidclone: Detecting android malware variants by exposing code clones. In: 2016 Sixth International Conference on Digital Information and Communication Technology and its Applications (DICTAP). IEEE, pp. 79–84.

Alam, S., Sogukpinar, I., 2020. Droidclone: attack of the android malware clones-a step towards stopping them. Computer Science and Information Systems (00). 35–35

AMD, 2021. Secure encrypted virtualization (sev). Accessed on 08.11.2022. https://developer.amd.com/sev/.

Andriatsimandefitra, R., Tong, V.V.T., 2015. Detection and identification of android malware based on information flow monitoring. In: 2015 IEEE 2nd international conference on cyber security and cloud computing. IEEE, pp. 200–203.

Arfaoui, G., Gharout, S., Traoré, J., 2014. Trusted execution environments: A look under the hood. In: 2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering. IEEE, pp. 259–266.

Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'keeffe, D., Stillwell, M.L., et al., 2016. SCONE: Secure linux containers with intel SGX. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 689–703.

Asokan, N., Ekberg, J.-E., Kostiainen, K., Rajan, A., Rozas, C., Sadeghi, A.-R., Schulz, S., Wachsmann, C., 2014. Mobile trusted computing. Proc. IEEE 102 (8), 1189–1206.

Azab, A.M., Ning, P., Shah, J., Chen, Q., Bhutkar, R., Ganesh, G., Ma, J., Shen, W., 2014. Hypervision across worlds: real-time kernel protection from the arm trustzone secure world. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 90–102.

Azab, A.M., Ning, P., Zhang, X., 2011. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 375–388.

Azab, A.M., Swidowski, K., Bhutkar, R., Ma, J., Shen, W., Wang, R., Ning, P., 2016. Skee: A lightweight secure kernel-level execution environment for arm. In: NDSS, Vol. 16, pp. 21–24.

Basse, F., 2016. Amlogic s905 sytem on chip: bypassing the (not so) secure boot to dump the bootrom. Accessed on 27.07.2021. https://fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html.

Baumann, A., Peinado, M., Hunt, G., 2015. Shielding applications from an untrusted cloud with haven. ACM Transactions on Computer Systems (TOCS) 33 (3), 1–26.

Beaupre, S., 2015. Trustnone.

Beniamini, G., a. Exploring qualcomm's secure execution environment. http://bits-please.blogspot.gr/2016/04/exploring-qualcomms-secure-execution.html.

Beniamini, G., b. Qsee privilege escalation vulnerability and exploit (cve-2015-6639), may 2016. URL https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html 64.

Beniamini, G., c. Trustzone kernel privilege escalation (cve-2016-2431), 2016 5. https://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html.

Beniamini, G., 2015a. Android linux kernel privilege escalation vulnerability and exploit (cve-2014-4322).

Beniamini, G., 2015b. Full trustzone exploit for msm8974. URL http://bits-please.blogspot.co.il/2015/08/full-trustzone-exploit-for-msm8974.html.

Beniamini, G., 2016a. Extracting qualcomm's keymaster keysbreaking android full disk encryption. https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html.

Beniamini, G., 2016b. War of the worlds-hijacking the linux kernel from qsee.

Beniamini, G., 2017. Trust issues: Exploiting trustzone tees. Accessed on 27.07.2021. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html.

Bogdanov, A., Eisenbarth, T., Paar, C., Wienecke, M., 2010. Differential cache-collision timing attacks on aes with applications to embedded cpus. In: Cryptographers' Track at the RSA Conference. Springer, pp. 235–251.

Boivie, R., Williams, P., 2012. Secureblue++: cpu support for secure execution. IBM, IBM Research Division, RC25287 (WAT1205-070) 1–9.

Bonneau, J., Mironov, I., 2006. Cache-collision timing attacks against aes. In: International Workshop on Cryptographic Hardware and Embedded Systems. Springer, pp. 201–215.

Brand, P.,. Op-tee. Accessed on 08.11.2022. https://github.com/OP-TEE.

Brasser, F., Gens, D., Jauernig, P., Sadeghi, A.-R., Stapf, E., 2019. Sanctuary: Arming trustzone with user-space enclaves. NDSS.

Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.-R., 2017. Software grand exposure:SGX cache attacks are practical. 11th USENIX Workshop on Offensive Technologies (WOOT 17).

Busch, M., Westphal, J., Mueller, T., 2020. Unearthing the trustedcore: A critical review on huawei's trusted execution environment. 14th USENIX Workshop on Offensive Technologies (WOOT 20).

Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., et al., 2019. Fallout: leaking data on meltdown-resistant cpus. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 769–784.

Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., Von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D., 2019. A systematic evaluation of transient execution attacks and defenses. In: 28th USENIX Security Symposium (USENIX Security 19), pp. 249–266.

Cerdeira, D., Santos, N., Fonseca, P., Pinto, S., 2020. SOK: understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1416–1432.

Chakraborty, D., Hanzlik, L., Bugiel, S., 2019. SIMPTM: User-centric TPM for mobile devices. In: 28th USENIX Security Symposium (USENIX Security 19), pp. 533–550.

Chang, R., Jiang, L., Chen, W., Xiang, Y., Cheng, Y., Alelaiwi, A., 2017. Mipe: a practical memory integrity protection method in a trusted execution environment. Cluster Comput 20 (2), 1075–1087.

Checkoway, S., Shacham, H., 2013. Iago attacks: why the system call api is a bad untrusted rpc interface. ACM SIGARCH Computer Architecture News 41 (1), 253–264.

Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., Lai, T.H., 2019. Sgxpectre: Stealing intel secrets from SGX enclaves via speculative execution. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, pp. 142–157.

Chen, G., Li, M., Zhang, F., Zhang, Y., 2019. Defeating speculative-execution attacks on sgx with hyperrace. In: 2019 IEEE Conference on Dependable and Secure Computing (DSC). IEEE, pp. 1–8.

Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R., 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. ACM SIGOPS Operating Systems Review 42 (2), 2–13.

Chen, Y., Zhang, Y., Wang, Z., Wei, T., 2017. Downgrade attack on trustzone. Arxiv:1707.05082

Cho, H., Zhang, P., Kim, D., Park, J., Lee, C.-H., Zhao, Z., Doupé, A., Ahn, G.-J., 2018. Prime+ count: novel cross-world covert channels on arm trustzone. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 441–452.

Cho, Y., Shin, J., Kwon, D., Ham, M., Kim, Y., Paek, Y., 2016. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In: 2016 USENIX Annual Technical Conference (USENIX ATC 16), pp. 565–578.

Cooijmans, T., de Ruiter, J., Poll, E., 2014. Analysis of secure key storage solutions on android. In: Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, pp. 11–20.

Corporation, N., 2015. Tlk repository. Accessed on 27.07.2021. http://nv-tegra.nvidia.com/gitweb/?p=3rdparty/otepartner/tlk.git.

Costan, V., Lebedev, I., Devadas, S., 2016. Sanctum: Minimal hardware extensions for strong software isolation. In: 25th USENIX Security Symposium (USENIX Security 16), pp. 857–874.

Criswell, J., Dautenhahn, N., Adve, V., 2014. Virtual ghost: protecting applications from hostile operating systems. ACM SIGARCH Computer Architecture News 42 (1), 81–96.

Cui, A., Housley, R., 2017. BADFET: defeating modern secure boot using second-order pulsed electromagnetic fault injection. 11th USENIX Workshop on Offensive Technologies (WOOT 17).

Dautenhahn, N., Kasampalis, T., Dietz, W., Criswell, J., Adve, V., 2015. Nested kernel: an operating system architecture for intra-kernel privilege separation. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 191–206.

Dietrich, K., Winter, J., 2009. Implementation aspects of mobile and embedded trusted computing. In: International Conference on Trusted Computing. Springer, pp. 29–44.

Diffie, W., Hellman, M., 1976. New directions in cryptography. IEEE Trans. Inf. Theory 22 (6), 644–654.

Domnitser, L., Jaleel, A., Loew, J., Abu-Ghazaleh, N., Ponomarev, D., 2012. Non-monopolizable caches: low-complexity mitigation of cache side channel attacks. ACM Transactions on Architecture and Code Optimization (TACO) 8 (4), 1–21.

Drozdovskyi, T.A., Moliavko, O.S., 2019. Mtower: trusted execution environment for MCU-based devices. Journal of Open Source Software 4 (40), 1494.

Ekberg, J.-E., Afanasyeva, A., Asokan, N., 2012. Authenticated encryption primitives for size-constrained trusted computing. In: International Conference on Trust and Trustworthy Computing. Springer, pp. 1–18.

Ekberg, J.-E., et al., 2007. Mobile trusted module (MTM)–an introduction.

Elenkov, N., 2013. Credential storage enhancements in android 4.3. URL-nelenkov. blogspot. co. uk/2013/08/credential-storage-enhancements-android-43. html.

Evenchick, E., 2018. Rustzone: Writing trusted applications in rust.

Felton, D.,. Trustonic, trusted executed environment(tee). Accessed on 08.11.2022. https://www.trustonic.com/technology/trusted-execution-environment.

Ferraiuolo, A., Baumann, A., Hawblitzel, C., Parno, B., 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 287–305.

Feske, N., 2015. Genode operating system framework.

Fitzek, A., Achleitner, F., Winter, J., Hein, D., 2015. The andix research os - arm trustzone meets industrial control systems security. In: 2015 IEEE 13th International Conference on Industrial Informatics (INDIN), pp. 88–93. doi:10.1109/INDIN.2015.7281715.

Ge, X., Vijayakumar, H., Jaeger, T., 2014. Sprobes: Enforcing kernel code integrity on the trustzone architecture. arXiv:1410.7747

GlobalPlatform,. Globalplatform specifications. Accessed on 27.09.2022. http://www.globalplatform.org/.

González, J., Bonnet, P., 2013. Towards an open framework leveraging a trusted execution environment. In: International Symposium on Cyberspace Safety and Security. Springer, pp. 458–467.

Google,. Google. (n.d.). trusty tee. Accessed on 08.11.2022. https://source.android.com/security/trusty/index.html.
Götzfried, J., Eckert, M., Schinzel, S., Müller, T., 2017. Cache attacks on intel sgx. In: Proceedings of the 10th European Workshop on Systems Security, pp. 1–6.
Götzfried, J., Eckert, M., Schinzel, S., Müller, T., 2017. Cache attacks on intel sgx. In: Proceedings of the 10th European Workshop on Systems Security. Association for Computing Machinery, New York, NY, USA doi:10.1145/3065913.3065915.
Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C., 2017. Aslr on the line: Practical cache attacks on the mmu. In: NDSS, Vol. 17, p. 26.
Gruss, D., Maurice, C., Fogh, A., Lipp, M., Mangard, S., 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp. 368–379.
Gruss, D., Maurice, C., Wagner, K., Mangard, S., 2016. Flush+ flush: a fast and stealthy cache attack. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 279–299.
Gruss, D., Spreitzer, R., Mangard, S., 2015. Cache template attacks: Automating attacks on inclusive last-level caches. In: 24th USENIX Security Symposium (USENIX Security 15), pp. 897–912.
Guan, L., Liu, P., Xing, X., Ge, X., Zhang, S., Yu, M., Jaeger, T., 2017. Trustshadow: Secure execution of unmodified applications with arm trustzone. In: Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, pp. 488–501.
Guanciale, R., Nemati, H., Baumann, C., Dam, M., 2016. Cache storage channels: Alias-driven attacks and verified countermeasures. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 38–55.
Hofmann, O.S., Kim, S., Dunn, A.M., Lee, M.Z., Witchel, E., 2013. Inktag: Secure applications on an untrusted operating system. In: Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems, pp. 265–278.
Holding, A., 2009. Arm security technology: Building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.
Hou, S., Ye, Y., Song, Y., Abdulhayoglu, M., 2017. Hindroid: an intelligent android malware detection system based on structured heterogeneous information network. In: Proceedings of the 23rd ACM SIGKDD International conference on knowledge discovery and data mining, pp. 1507–1515.
Hua, Z., Gu, J., Xia, Y., Chen, H., Zang, B., Guan, H., 2017. VTZ: Virtualizing ARM trustzone. In: 26th USENIX Security Symposium (USENIX Security 17), pp. 541–556.
Hussin, W.H.W., Coulton, P., Edwards, R., 2005. Mobile ticketing system employing trustzone technology. In: International Conference on Mobile Business (ICMB'05). IEEE, pp. 651–654.
Hussin, W.H.W., Edwards, R., Coulton, P., 2006. E-pass using DRM in symbian v8 os and trustzone: Securing vital data on mobile devices. In: 2006 International Conference on Mobile Business. IEEE. 14–14
Inc, M. T.,. Atecc608a, secure element to secure authentication. Accessed on 27.10.2022. https://www.microchip.com/en-us/product/ATECC608A.
Infineon, T.,. Optigatmtrust x sls 32aia. Accessed on 08.11.2022. https://www.infineon.com/cms/en/product/security-smart-card-solutions/optiga-embedded-security-solutions/optiga-trust/optiga-trust-x-sls-32aia.
Integrated, M.,. Maxq1061, deep cover cryptographic controller for embedded devices. Accessed on 27.07.2021 https://www.maximintegrated.com/en/products/microcontrollers/MAXQ1061.html.
Intel, 2014. Intel software guard extensions programming reference. Accessed on 08.11.2022 https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.
Irazoqui, G., Eisenbarth, T., Sunar, B., 2015. S$a: A shared cache attack that works across cores and defies VM sandboxing–and its application to aes. In: 2015 IEEE Symposium on Security and Privacy. IEEE, pp. 591–604.
Irazoqui, G., Eisenbarth, T., Sunar, B., 2016. Cross processor cache attacks. In: Proceedings of the 11th ACM on Asia conference on computer and communications security, pp. 353–364.
Jang, J., Choi, C., Lee, J., Kwak, N., Lee, S., Choi, Y., Kang, B.B., 2016. Privatezone: providing a private execution environment using arm trustzone. IEEE Trans Dependable Secure Comput 15 (5), 797–810.
Jang, J., Kang, B.B., 2018. Retrofitting the partially privileged mode for tee communication channel protection. IEEE Trans Dependable Secure Comput 17 (5), 1000–1014.
Jang, J.S., Kong, S., Kim, M., Kim, D., Kang, B.B., 2015. Secret: Secure channel between rich execution environment and trusted execution environment. NDSS.
Janjua, H., Ammar, M., Crispo, B., Hughes, D., 2019. Towards a standards-compliant pure-software trusted execution environment for resource-constrained embedded devices. In: Proceedings of the 4th Workshop on System Software for Trusted Execution, pp. 1–6.
Jarabek, C., Barrera, D., Aycock, J., 2012. Thinav: Truly lightweight mobile cloud-based anti-malware. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 209–218.
Ji, D., Zhang, Q., Zhao, S., Shi, Z., Guan, Y., 2019. Microtee: designing tee os based on the microkernel architecture. In: 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE). IEEE, pp. 26–33.
Keltner, N., Holmes, C., 2014. Here be dragons: Vulnerabilities in trustzone.
Kim, Y., Lee, J., Mai, T.X., Paek, Y., 2012. Improving performance of nested loops on reconfigurable array processors. ACM Transactions on Architecture and Code Optimization (TACO) 8 (4), 1–23.
Kim, Y., Liszka, K.J., Chan, C.-C., 2016. Using droiddream android malware behavior for identification of other android malware families. In: Proceedings of the International Conference on Security and Management (SAM). The Steering Committee of The World Congress in Computer Science, Computer, p. 286.
Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al., 2019. Spectre attacks: Exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1–19.
Komaromy, D., 2018. Unbox your phone part iAccessed on 08.11.2022 https://medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c.
Koruyeh, E.M., Shirazi, S.H.A., Khasawneh, K.N., Song, C., Abu-Ghazaleh, N., 2020. Speccfi: Mitigating spectre attacks using cfi informed speculation. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 39–53.
Kostiainen, K., Ekberg, J.-E., Asokan, N., Rantala, A., 2009. On-board credentials with open provisioning. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, pp. 104–115.
Kwon, D., Seo, J., Cho, Y., Lee, B., Paek, Y., 2019. Pros: light-weight privatized se cure oses in arm trustzone. IEEE Trans. Mob. Comput. 19 (6), 1434–1447.
Kwon, Y., Dunn, A.M., Lee, M.Z., Hofmann, O.S., Xu, Y., Witchel, E., 2016. Sego: pervasive trusted metadata for efficiently verified untrusted system services. ACM SIGARCH Computer Architecture News 44 (2), 277–290.
Lammens, L.,. Code aurora forum security bulletin. Accessed on 27.10.2022 https://www.codeaurora.org/security-bulletin.
Lapid, B., Wool, A., 2018. Navigating the samsung trustzone and cache-attacks on the keymaster trustlet. In: European Symposium on Research in Computer Security. Springer, pp. 175–196.
Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., Song, D., 2020. Keystone: An open framework for architecting trusted execution environments. In: Proceedings of the Fifteenth European Conference on Computer Systems, pp. 1–16.
Lee, U., Park, C., 2020. Softee: software-based trusted execution environment for user applications. IEEE Access 8, 121874–121888.
Li, W., Xia, Y., Lu, L., Chen, H., Zang, B., 2019. Teev: virtualizing trusted execution environments on mobile platforms. In: Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 2–16.
Liedtke, J., Hartig, H., Hohmuth, M., 1997. Os-controlled cache predictability for real–time systems. In: Proceedings Third IEEE Real-Time Technology and Applications Symposium. IEEE, pp. 213–224.
Lipp, M., 2016. Cache attacks and rowhammer on arm.
Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S., 2016. Armageddon: cache attacks on mobile devices. In: 25th USENIX Security Symposium (USENIX Security 16), pp. 549–564.
Lipp, M., Hažić, V., Schwarz, M., Perais, A., Maurice, C., Gruss, D., 2020. Take a way: Exploring the security implications of amd's cache way predictors. In: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, pp. 813–825.
Lipp, M., Kogler, A., Oswald, D., Schwarz, M., Easdon, C., Canella, C., Gruss, D., 2021. Platypus: Software-based power side-channel attacks on x86. IEEE Symposium on Security and Privacy (SP).
Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., et al., 2018. Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 973–990.
Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B., 2015. Last-level cache side-channel attacks are practical. In: 2015 IEEE symposium on security and privacy. IEEE, pp. 605–622.
Liu, H., Saroiu, S., Wolman, A., Raj, H., 2012. Software abstractions for trusted sensors. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, pp. 365–378.
Lynch, W.L., Bray, B.K., Flynn, M.J., 1992. The effect of page allocation on caches. ACM SIGMICRO Newsletter 23 (1–2), 222–225.
Machiry, A., Gustafson, E., Spensky, C., Salls, C., Stephens, N., Wang, R., Bianchi, A., Choe, Y.R., Kruegel, C., Vigna, G., 2017. Boomerang: Exploiting the semantic gap in trusted execution environments. NDSS.
Maistri, P., Leveugle, R., Bossuet, L., Aubert, A., Fischer, V., Robisson, B., Moro, N., Maurine, P., Dutertre, J.-M., Lisart, M., 2014. Electromagnetic analysis and fault injection onto secure circuits. In: 2014 22nd International Conference on Very Large Scale Integration (VLSI-SoC). IEEE, pp. 1–6.
Maña, A., Muñoz, A., 2006. Protected computing vs. trusted computing. In: 2006 1st International Conference on Communication Systems Software & Middleware. IEEE, pp. 1–7.
Marchand, C., Aubert, A., Bossuet, L., et al., 2017. On the security evaluation of the arm trustzone extension in a heterogeneous soc. In: 2017 30th IEEE International System-on-Chip Conference (SOCC). IEEE, pp. 108–113.
McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A., 2010. Trustvisor: Efficient tcb reduction and attestation. In: 2010 IEEE Symposium on Security and Privacy. IEEE, pp. 143–158.
McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H., 2008. Flicker: An execution infrastructure for tcb minimization. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, pp. 315–328.
McGill, K.N., 2013. Trusted mobile devices: requirements for a mobile trusted platform module. Johns Hopkins APL Tech Dig 32 (2), 544–554.

McGillion, B., Dettenborn, T., Nyman, T., Asokan, N., 2015. Open-tee–an open virtual trusted execution environment. In: 2015 IEEE Trustcom/BigDataSE/ISPA, Vol. 1. IEEE, pp. 400–407.
Melara, M. S., Freedman, M. J., Bowman, M., 2019. Enclavedom: privilege separation for large-tcb applications in trusted execution environments. Arxiv:1907.13245
Meng, H., Thing, V.L., Cheng, Y., Dai, Z., Zhang, L., 2018. A survey of android exploits in the wild. Computers & Security 76, 71–91.
Minkin, M., Moghimi, D., Lipp, M., Schwarz, M., Van Bulck, J., Genkin, D., Gruss, D., Piessens, F., Sunar, B., Yarom, Y., 2019. Fallout: Reading kernel writes from user space. Arxiv:1905.12701
Moghimi, A., Irazoqui, G., Eisenbarth, T., 2017. Cachezoom: How sgx amplifies the power of cache attacks. In: International Conference on Cryptographic Hardware and Embedded Systems. Springer, pp. 69–90.
Murdock, K., Oswald, D., Garcia, F.D., Van Bulck, J., Gruss, D., Piessens, F., 2020. Plundervolt: software-based fault injection attacks against intel sgx. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1466–1482.
Murdock, K., Oswald, D., Garcia, F.D., Van Bulck, J., Piessens, F., Gruss, D., 2020. Plundervolt: how a little bit of undervolting can create a lot of trouble. IEEE Security & Privacy 18 (5), 28–37.
Ngabonziza, B., Martin, D., Bailey, A., Cho, H., Martin, S., 2016. Trustzone explained: architectural features and use cases. In: 2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC). IEEE, pp. 445–451.
Objects, T.,. To136 secure element. Accessed on 27.07.2021 https://www.trusted-objects.com/webtest/index.php?page=en-TO136-secure-element.
Oh, S.-C., Koh, K., Kim, C.-Y., Kim, K., Kim, S., 2012. Acceleration of dual os virtualization in embedded systems. In: 2012 7th International Conference on Computing and Convergence Technology (ICCCT). IEEE, pp. 1098–1101.
Oliveira, D., Gomes, T., Pinto, S., 2021. uTango: an open-source tee for the internet of things. Arxiv:2102.03625
Osvik, D.A., Shamir, A., Tromer, E., 2006. Cache attacks and countermeasures: the case of aes. In: Cryptographers' track at the RSA conference. Springer, pp. 1–20.
Pinto, S., Garlati, C., 2020. Multi zone security for arm cortex-m devices. In: Proc. Embedded World Conf..
Pinto, S., Gomes, T., Pereira, J., Cabral, J., Tavares, A., 2017. Iioteed: an enhanced, trusted execution environment for industrial iot edge devices. IEEE Internet Comput 21 (1), 40–47.
Pinto, S., Santos, N., 2019. Demystifying arm trustzone: a comprehensive survey. ACM Computing Surveys (CSUR) 51 (6), 1–36.
Pirker, M., Slamanig, D., 2012. A framework for privacy-preserving mobile payment on security enhanced arm trustzone platforms. In: 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications. IEEE, pp. 1155–1160.
Pirker, M., Slamanig, D., Winter, J., 2012. Practical privacy preserving cloud resource–payment for constrained clients. In: International Symposium on Privacy Enhancing Technologies Symposium. Springer, pp. 201–220.
Qiu, P., Wang, D., Lyu, Y., Qu, G., 2019. Voltjockey: Breaching trustzone by software–controlled voltage manipulation over multi-core frequencies. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 195–209.
Qiu, P., Wang, D., Lyu, Y., Qu, G., 2019. Voltjockey: breaking SGX by software-controlled voltage-induced hardware faults. In: 2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST). IEEE, pp. 1–6.
Qiu, P., Wang, D., Lyu, Y., Tian, R., Wang, C., Qu, G., 2020. Voltjockey: a new dynamic voltage scaling-based fault injection attack on intel sgx. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 40 (6), 1130–1143.
Qualcomm, 2018. Qualcomm product security - security advisories. https://www.qualcomm.com/company/product-security/security-advisories.
Qui, P., Wang, D., Lyu, Y., Qu, G., 2020. Voltjockey: abusing the processor voltage to break arm trustzone. GetMobile: Mobile Computing and Communications 24 (2), 30–33.
Ragab, H., Milburn, A., Razavi, K., Bos, H., Giuffrida, C., 2021. Crosstalk: Speculative data leaks across cores are real. IEEE Symposium on Security and Privacy. Institute of Electrical and Electronics Engineers Inc..
Rosenberg, D., 2013. Unlocking the motorola bootloader. Azimuth Security Blog.
Rosenberg, D., 2014. Reflections on trusting trustzone. BlackHat USA.
Roth, T., 2013. Next generation mobile rootkits. Hack in Paris.
Ryan, K., 2019. Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 181–194.
Ryan, K., 2019. Return of the hidden number problem. IACR Transactions on Cryptographic Hardware and Embedded Systems 146–168.
Sabt, M., Achemlal, M., Bouabdallah, A., 2015. Trusted execution environment: what it is, and what it is not. In: 2015 IEEE Trustcom BigDataSE ISPA, Vol. 1. IEEE, pp. 57–64.
Santos, N., Raj, H., Saroiu, S., Wolman, A., 2011. Trusted language runtime (tlr) enabling trusted applications on smartphones. In: Proceedings of the 12th Workshop on Mobile Computing Systems and Applications, pp. 21–26.
Santos, N., Raj, H., Saroiu, S., Wolman, A., 2014. Using arm trustzone to build a trusted language runtime for mobile applications. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 67–80.
Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D., 2019. Zombieload: Cross-privilege-boundary data sampling. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 753–768.
Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S., 2017. Malware guard extension: Using sgx to conceal cache attacks. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 3–24.
Semiconductors, N., 2018. A71ch, plug & trust secure element. Accessed on 27.09.2022 https://www.nxp.com/docs/en/data-sheet/A71CH-SDS.pdf.
Semiconductors, N., 2021. Se050 plug & trust secure element. Accessed on 08.11.2022 https://www.nxp.com/docs/en/data-sheet/SE050-DATASHEET.pdf.
Shah, J. H., et al., 2012. Armithril: A secure os leveraging arm's trustzone technology.
Shen, D., 2015. "attacking your trusted core exploiting trustzone on android". Accessed on 08.11.2022 https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android.pdf.
Shi, J., Song, X., Chen, H., Zang, B., 2011. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W). IEEE, pp. 194–199.
Shin, J., Kim, Y., Park, W., Park, C., 2012. Dfcloud: a tpm-based secure data access control method of cloud storage in mobile devices. In: 4th IEEE International Conference on Cloud Computing Technology and Science Proceedings. IEEE, pp. 551–556.
Shinde, S., Le Tien, D., Tople, S., Saxena, P., 2017. Panoply: Low-tcb linux applications with sgx enclaves. NDSS.
SierraWare,. Sierratee for arm trustzone. Accessed on 08.11.2022 https://www.sierraware.com/open-source-ARM-TrustZone.html.
Solacia,. Securitee. Accessed on 27.07.2021 http://www.sola-cia.com/en/securiTee/product.asp.
Soviany, S., Scheianu, A., Suciu, G., Vulpe, A., Fratu, O., Istrate, C., 2018. Android malware detection and crypto-mining recognition methodology with machine learning. In: 2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC). IEEE, pp. 14–21.
Spreitzer, R., Gérard, B., 2014. Towards more practical time-driven cache attacks. In: IFIP International Workshop on Information Security Theory and Practice. Springer, pp. 24–39.
Spreitzer, R., Plos, T., 2013. On the applicability of time-driven cache attacks on mobile devices (extended version).
Sun, H., Sun, K., Wang, Y., Jing, J., 2015. Trustotp: Transforming smartphones into secure one-time password tokens. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 976–988.
Sun, H., Sun, K., Wang, Y., Jing, J., Wang, H., 2015. Trustice: Hardware-assisted isolated computing environments on mobile devices. In: 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, pp. 367–378.
Takahashi, A., Tibouchi, M., Abe, M., 2018. New bleichenbacher records: practical fault attacks on qdsa signatures. IACR Cryptol. ePrint Arch. 2018, 396.
Takei, C., Takada, H., Yamamoto, M., Honda, S., 2009. Integrated software platform for automotive systems. In: 2009 International SoC Design Conference (ISOCC). IEEE, pp. 377–379.
Tamrakar, S., Ekberg, J.-E., Asokan, N., 2011. Identity verification schemes for public transport ticketing with nfc phones. In: Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing, pp. 37–48.
Tang, A., Sethumadhavan, S., Stolfo, S., 2017. CLKSCREW: exposing the perils of security-oblivious energy management. In: 26th USENIX Security Symposium (USENIX Security 17), pp. 1057–1074.
TCG, 2013. Tpm 2.0 mobile trusted module use cases. Accessed on 08.11.2022 https://trustedcomputinggroup.org/resource/mobile-trusted-module-2-0-use-cases/.
Tögl, R., Winter, J., Pirker, M., 2013. A path towards ubiquitous protection of media. In: Proceedings Workshop on Web Applications and Secure Hardware, ser. CEUR Workshop Proceedings, Vol. 1011. Citeseer, pp. 32–38.
TrustKernel,. T6. Accessed on 27.07.2021 https://www.trustkernel.com/.
Trustonic, 2017. Not just droning on! the rise of kinibi-m. https://www.trustonic.com/news/blog/not-just-droning-rise-kinibi-m/.
Tsai, C.-C., Porter, D.E., Vij, M., 2017. Graphene-sgx: A practical library OS for unmodified applications on SGX. In: 2017 USENIX Annual Technical Conference (USENIATC 17), pp. 645–658.
Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R., 2018. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 991–1008.
Van Bulck, J., Moghimi, D., Schwarz, M., Lippi, M., Minkin, M., Genkin, D., Yarom, Y., Sunar, B., Gruss, D., Piessens, F., 2020. Lvi: Hijacking transient execution through microarchitectural load value injection. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 54–72.
Van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C., 2019. Ridl: rogue in-flight data load. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 88–105.
Wagner, D., 1999. The boomerang attack. In: International Workshop on Fast Software Encryption. Springer, pp. 156–170.
Weichselbaum, L., Neugschwandtner, M., Lindorfer, M., Fratantonio, Y., Van Der Veen, V., Platzer, C., 2014. Andrubis: android malware under the magnifying glass. Vienna University of Technology, Tech. Rep. TR-ISECLAB-0414-001 1–10.
Weiß, M., Heinz, B., Stumpf, F., 2012. A cache timing attack on AES in virtualization environments. In: International Conference on Financial Cryptography and Data Security. Springer, pp. 314–328.

Weiß, M., Weggenmann, B., August, M., Sigl, G., 2014. On cache timing attacks considering multi-core aspects in virtualized embedded systems. In: International Conference on Trusted Systems. Springer, pp. 151–167.
Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T. F., Yarom, Y., 2018. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution.
Xia, Y., Hua, Z., Yu, Y., Gu, J., Chen, H., Zang, B., Guan, H., 2021. Colony: a privileged trusted execution environment with extensibility. IEEE Trans. Comput.
Yarom, Y., Falkner, K., 2014. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 719–732.
Ying, K., Thavai, P., Du, W., 2019. Truz-view: Developing trustzone user interface for mobile os using delegation integration model. In: Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, pp. 1–12.
Yun, M.H., Zhong, L., 2019. Ginseng: Keeping secrets in registers when you distrust the operating system. NDSS.
Zhang, N., Sun, K., Lou, W., Hou, Y.T., 2016. Case: Cache-assisted secure execution on arm processors. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 72–90.
Zhang, N., Sun, K., Shands, D., Lou, W., Hou, Y.T., 2016. Truspy: cache side-channel information leakage from the secure world on arm devices. IACR Cryptol. ePrint Arch. 2016, 980.
Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T., 2012. Cross-Vm side channels and their use to extract private keys. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 305–316.
Zhang, Z., Barthe, G., Chuengsatiansup, C., Schwabe, P., Yarom, Y., 2022. Breaking and fixing speculative load hardening. Cryptology ePrint Archive.

Antonio Muñoz is an assistant professor at the University of Malaga, where he obtained his Ph.D. and M.Sc. degrees in computer engineering and computer science, respectively, in 2010 and 2005. He holds his Ph.D. an MSc degree in Computer Science and a Postgraduate Master degree in Software Engineer and Artificial Intelligence, both of them from the University of Malaga. His principal research interests are in the area of Agent technology, Digital Content Protection, Cryptographic Hardware based Systems, Security Patterns and Security Engineering.

Ruben Rios is assistant professor at the University of Malaga, Spain. He received the Ph.D. degree in Computer Science in 2014. His main research activities are centred on the design and development of solutions for the protection of digital privacy and anonymity in scenarios with resource-constrained devices. Dr. Rios was awarded the FPU fellowship from the Spanish Ministry of Education and received the prize to the most outstanding Ph.D. thesis in the University of Malaga. He is also one of the authors of the book ⁂Location Privacy in Wireless Sensor Networks𝗀 from CR Press.

Rodrigo Roman is an assistant professor at the University of Malaga, where he obtained his Ph.D. and M.Sc. degrees in computer engineering and computer science, respectively, in 2008 and 2003. Previously, he worked for the Institute of Infocomm Research (I2R) in Singapore in the areas of sensor network security and cloud security. Working to make security simple and usable, his research is fo- cused on the development of protection mechanisms for the Internet of Things and related paradigms, such as cloud computing and fog computing.

Javier Lopez is a full professor and head of the Net- work, Information and Computer Security (NICS) Lab at the University of Malaga, Malaga, 29071, Spain. His research activities are mainly focused on network security, security protocols, and critical information infrastructures, leading a number of national and international research projects in those areas. He is Senior Member of IEEE.