The Interdisciplinary Center, Herzlia

Efi Arazi School of Computer Science

M.Sc. program - Research Track

Attacking ARM TrustZone using Hardware vulnerability

by

Ron Stajnrod

M.Sc. dissertation, submitted in partial fulfillment of the requirements for the M.Sc. degree, research track, School of Computer Science
The Interdisciplinary Center, Herzliya

February 2021

This work was carried out under the supervision of Dr. **Nezer Zaidenberg** and assistance of **Raz Ben Yehuda** from the University of Jyväskylä, Finland.

Abstract

ARM TrustZone offers trusted execution environment (TEE) embedded into the processor cores. ARM TrustZone has become widely used in ARM processor devices such as Smartphones, IoT devices and Embedded devices. Due to the increase of security awareness ARM implements the TrustZone mechanism which enables device manufacturers to add secure storage and secure applications to perform cryptographic logic like saving user fingerprint or DRM protection to prevent data leakage.

While ARM TrustZone can improve the overall security of the device it depends on the vendor (Who manufacture the System-on-chip) to fully comply with ARM TrustZone specification. In the case where a vendor does not fully comply with ARM TrustZone specification, vulnerabilities may occur [1] (Few vulnerable devices are shown in Table 2).

The contribution of this work is threefold. First, we present how ARM TrustZone works and what are the main hardware components which assure that ARM TrustZone will be secure.

Second, we evaluate and demonstrate a vulnerability caused by hardware implementation which does not fully comply with ARM TrustZone hardware specifications. Third, we argue that software configuration bugs in the low-level vendor implementation may introduce vulnerabilities to the ARM TrustZone. Our attack is based on DMA Transaction so it can be executed from a peripheral device like a controller, network card, etc… or executed on the device itself using the DMA controller.

In our research we present how we gain full access to the secure world from the normal world.

Table of Contents

List of Figures

List of Tables

1 Introduction

The development of the Internet of Things (IoT) is hailed as the third wave of world information development after computers and the Internet [2], with embedded systems as the driving force for technological development in many domains, such as automotive, healthcare, and industrial control in the emerging post-PC era. As an increasing number of computational and networked devices are integrated into all aspects of our lives in a pervasive and 'invisible' way, security becomes critical for the dependability of all smart or intelligent systems built upon these embedded systems [3]. Embedded IoT products are increasingly wireless. By their nature, such products are constrained in terms of computing and memory capacity and what can be done given cost realities [4]. The constrained nature of such devices means we are trying to 'build a fortress from pebbles', so to speak. Therefore, we must take the very best security measures to prevent malicious activity on those devices given the limited conditions, which often means cutting corners compared with other resource-rich areas of computing (personal computers, servers, etc.). ARM TrustZone [5] was introduced as part of the ARMv6 architecture and is widely used in smartphones, tablets, wearables and other devices.

As TrustZone is becoming a popular hardware security architecture for mobile devices and IoT, it is important to ensure the security of TrustZone itself [6]. Even though ARM TrustZone is a great way to implement security mechanisms across IoT-embedded devices, it is still prone to bad hardware and software implementations; thus, the hardware of different companies like Google, Samsung, Huawei, etc. might still be affected by severe vulnerabilities that compromise the entire security suite [7], [8], [9], [10]. Some ARM modules lack AMBA AXI [11] support, which leads to insecure memory separation between the Normal and Secure Worlds. In this paper, we present Direct Memory Access

(DMA) attack [12] on ARM TrustZone Trusted Applications (TA) running in Open Portable Trusted Execution Environment (OP-TEE) [13], [14]. This allows an attacker to execute arbitrary code in the Secure World or read arbitrary data from the secure world into the rich OS. Our attack is a control-flow attack [15], [16] on the OP-TEE kernel.

In this paper, we show a hardware vulnerability on SoC that compromises ARM TrustZone. Using DMA attack, we gain the ability to replace trusted applications with malicious ones. We demonstrate an attack on a Raspberry PI computer and explain how this method affects other platforms. This paper also provides measures to mitigate this vulnerability.

2 TrustZone Background

2.1 ARM Permission Model

ARM has a unique approach to security and privilege levels. In ARMv7, ARM introduced the concept of secured and non-secured worlds through the implementation of TrustZone and starting from ARMv7a. ARM presents four exceptions (permission) levels as follows.

Exception Level 0 (EL0) Refers to the user-space code. Exception Level 0 is analogous to ”ring 3” on the x86 platform.

Exception Level 1 (EL1) Refers to operating system code. Exception Level 1 is analogous to ”ring 0” on the x86 platform. Our attack demonstration starts from EL1 and escalating to EL3.

Exception Level 2 (EL2) Refers to HYP mode. Exception Level 2 is analogous to ”ring -1” or ”real mode” on the x86 platform.

Exception Level 3 (EL3) Refers to TrustZone as a special security mode that can monitor the ARM processor and may run a real-time security OS. There are no direct analogous modes, but related concepts in x86 are Intel’s ME or SMM.

Each exception level provides its own set of special purpose registers and can access these registers at the lower levels, but not higher levels. The general purpose registers are shared; therefore, moving to a different exception level on the ARM architecture does not require the expensive context switch associated with the x86 architecture.

2.2 ARM TrustZone

ARM TrustZone technology is aimed at establishing trust in ARM-based platforms. In contrast to a TPM (Trusted Platform Module), which is designed as a fixed-function

device with a predefined feature set, TrustZone represents a much more flexible approach by leveraging the CPU as a freely programmable trusted platform module. To do that, ARM introduced a special CPU mode called 'secure mode' in addition to the regular normal mode, thereby establishing the notions of a 'Secure World' and a 'Normal World' (Figure 1). The distinction between these worlds is completely orthogonal to the normal ring protection between user-level and kernel-level code, and hidden from the operating system running in the Normal World [17].

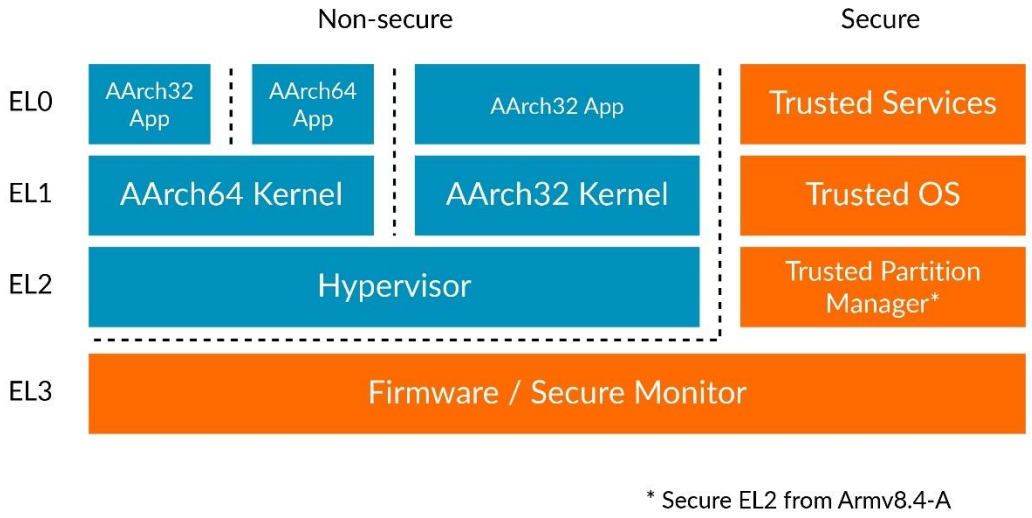

Figure 1: Normal and Secure World

As an example, the Linux kernel runs in EL1 and the user-space processes execute in EL0. The separation of Secure and Normal World secures certain RAM ranges and peripherals, which are only accessible by the Secure World. This means that a compromised Normal World code (in the user-space or the kernel) cannot access these memory ranges or devices. This separation is completely artificial. The same cores are used to run both Secure and Normal Worlds and they use the same RAM (Figure 2).

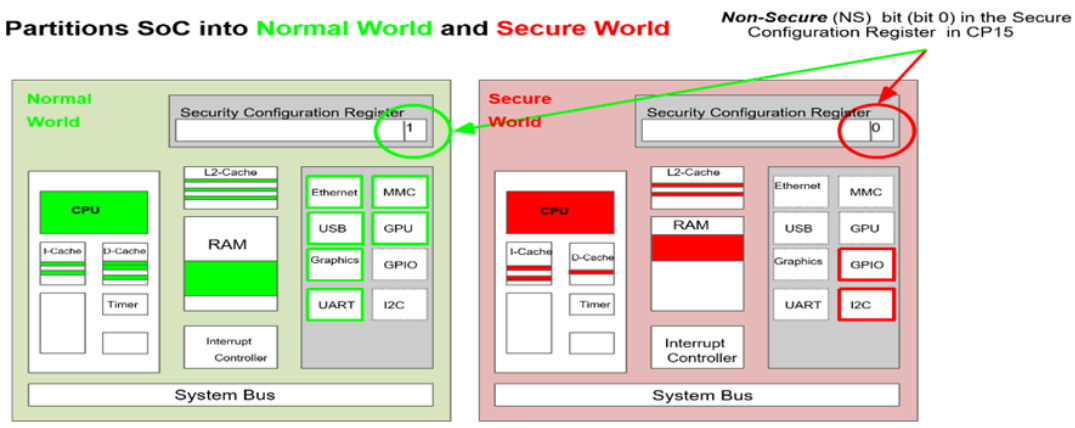

Figure 2: NS bit

The Non-Secure (NS) bit is used to determine whether the CPU executes in Normal or Secure World context to create a separation in memory. TrustZone technology extends beyond the processor into the SoC peripherals connected with the SoC, such as the DRAM controller (Figure 2), the DMA (Direct Memory Access), the secure boot ROM, the GIC (Generic Interrupt Controller), the DMA (Direct Memory Access), the secure boot ROM, the GIC (Generic Interrupt Controller), the TrustZone Address Space Controller (TZASC), the TrustZone Protection controller (TZPC) and the Dynamic Memory Controller (DMC).

The above components communicate through the AXI bus and the SoC communicates with peripherals through the AXI to APB bridge. The SoC peripherals are implemented by third-party companies; therefore, to reduce costs, some vendors choose not to comply entirely with TrustZone specifications. It is possible to access the entire memory from the Secure World but not vice versa. The Secure Monitor Call (SMC) instruction is used to traverse to the Monitor in EL3. The SMC depends on the manufacture implementation and, thus, is prone to bugs and other vulnerabilities [10]. This paper focuses on the physical level of memory isolation. TrustZone enables memory partitions between Normal and Secure Worlds by using the TZASC and the TZPC. This provides a secure

I/O to peripherals over standard interfaces. For instance, the SPI or GPIO route interrupts to the TEE kernel (Secure World kernel) through the TZPC. The NS bit is used to secure on-chip peripherals from the Rich Execution Environment (REE, Normal World) [18]. TZASC utilises the NS bit for a memory-mapped device like DRAM. These two devices require support from the AXI bus, which is vendor-specific. TrustZone use cases include building a root-of-trust for the system with everything needed for a secure boot and system recovery.

Secure World trusted applications may be used for secure PIN and biometric checks to ensure details are safe from hacking. Another trusted application use case is Digital Right Management (DRM) for online media, where the private information is kept within the Secure World so hackers cannot access the keys required to reverse-engineer the system.

Many more use cases of TrustZone can be found for IoT and mobile devices [19].

2.3 OP-TEE

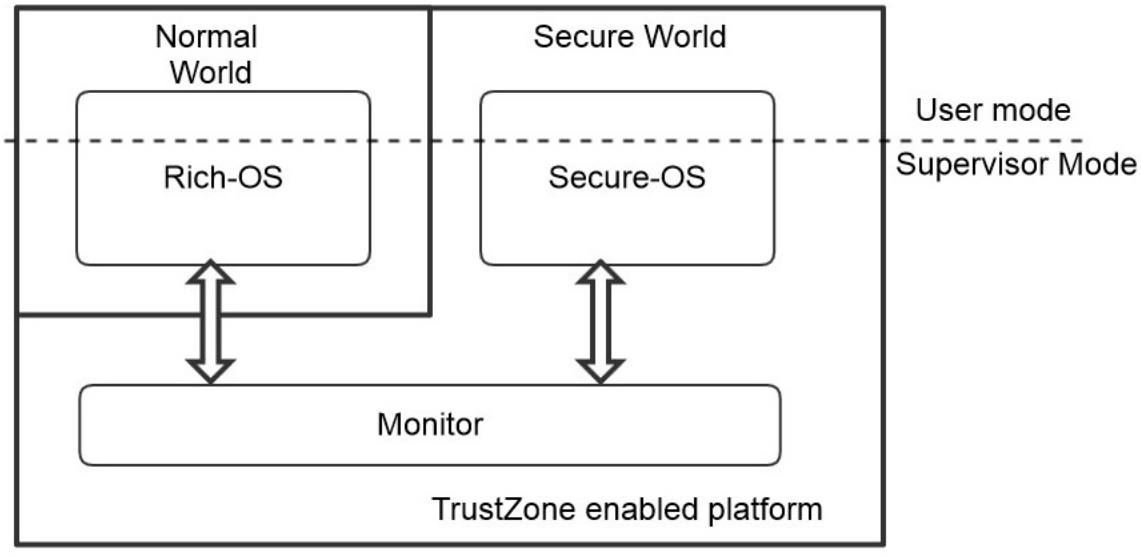

Figure 3: Outline of ARM TrustZone

OP-TEE [13] is a Trusted Execution Environment (TEE) designed as a companion to a non-secure Linux kernel running on ARM Cortex-A cores using the TrustZone

technology. OP-TEE implements TEE Internal Core API v1.1.x, which is the API exposed to Trusted Applications and the TEE Client API v1.0, which is the API describing how to communicate with a TEE. These APIs are defined in the GlobalPlatform API specifications. The non-secure OS is referred to as the Rich Execution Environment (REE) in TEE specifications.

OP-TEE is designed primarily to rely on the ARM TrustZone technology as the underlying hardware isolation mechanism. However, it has been structured to be compatible with any isolation technology suitable for the TEE concept and goals, such as running as a virtual machine or on a dedicated processor core. The main design goals for OP-TEE are:

• **Isolation** - OP-TEE provides isolation from the non-secure OS and protects the loaded Trusted Applications (TAs) from each other by using underlying hardware support.

• **Small footprint** - OP-TEE should remain small enough to reside in a reasonable amount of on-chip memory as found on ARM-based systems.

• **Portability** - OP-TEE is aimed to be pluggable to different architectures and must support various setups such as multiple client OSs or multiple TEEs.

OP-TEE offers threads and shared memory among the REE to the secured OS, Secured interrupts, RPC from the secured to the REE and communication from the REE to the Secured World via the SMC interface where some are possible attack vectors. For instance, consider an attack on the SMC interface. It is possible to replace the SMC interface from the REE side with malicious code that hijacks the SMC requests in the non-secure side, for example, by manipulating the kernel code itself by a DMA attack. It is also possible to attack the shared-memory in cases when it is used. TrustZone-protected DRAM or non-secure DRAM is used as the backing store. The data in the backing store are protected with a hash. However, read-only pages are not encrypted

because the OP-TEE binary itself is not encrypted. Therefore, a DMA attack on the OP-TEE kernel is easier than on TA-encrypted programs as it bypasses the MMU permissions model as well as the need to encrypt the code. Each TA is encrypted with a private key. The vendor creates a public key that is used to decrypt the TA. The decryption takes place in OP-TEE in the TrustZone. Thus, the program in its decrypted form is only visible in the Secured RAM and the processor's EL3 cache. It is, therefore, sensible to attack in the decryption area.

3 Related Work

Many words were written on side-channel attacks and other vulnerable targets in ARM architecture in prior research. In the area of ARM, [8] et al. describe a downgrade or rollback attack. A trusted application is encrypted for security purposes by public and private keys that originate from the hardware. In cases when the system is updated, old TAs can still be executed on the new system. A downgrade attack is when an attacker exploits a vulnerability in the old TA version by patching the old version onto the new TA version. According to [8], the above applies to the OP-TEE and QSEE (Qualcomm's Secure Execution Environment). [8] et al. describe a simple procedure for mobile phones: root the device, remount the 'system' partition in READ-WRITE mode, replace the current trustlet with an old vulnerable trustlet and use the trustlet. [8] et al. describe another possible rollback attack on the chain of trust and proves it possible to downgrade the bootloader successfully.

Armageddon [20] et al. explore attacks on ARM caches, concentrating on cross-core cache attacks in non-rooted arm mobile devices and showing a novel approach to exploit the coherence protocols. Although most smartphones have multiple processors that do not share caches, cache coherence protocols allow processors to fetch cache lines. By exploiting the lack of ‘cache flush’ on 'old' ARM cores (before ARMv8), a novel technique that analyses cache eviction strategies and another approach on how to perform cycle-timing without root access. Armageddon [20] et al. provide a technique to gain sensitive information such as inter-keystroke timings or the length of a swipe action requiring significantly higher measurement accuracy. As for TrustZone vulnerability, Armageddon [20] shows a cache attack used to monitor cache activity caused within the ARM TrustZone from the Normal World.

Flush and Reload attack [21] et al. take advantage of the coherence protocol in a multiprocessor computer. In most ARM processors, the last level cache is inclusive (i.e. it includes low-level cache lines); therefore, examining the content of the last-level cache may provide the contents of low-level cache lines of another core. However, the AutoLock [22] tool assesses the real risk in cache attacks, prevents cross-cache evictions, and highlights the intricacies of cache attacks in ARM. AutoLock [22] et al. claim that unlike Intel processors, many ARM caches are both inclusive and exclusive, and therefore hardens the LLC (last-level cache) attacks. In their work, Demme et al. [23] demonstrate that small changes to the cache architecture have a considerable impact on side-channel vulnerability. Like cache attacks, DMA attacks are continuously under research. [24] et al. show that by dumping memory frequently enough using DMA transactions, write patterns can be examined, and some algorithms, such as the RSA Montgomery ladder [25] , may leak secrets. DAGGER [26], a DMA-based keystroke logger, exfiltrates captured data to an external entity and cannot be detected by anti-virus. [26] shows how DAGGER can steal cryptographic keys, target OS kernel structure, and copy files from the file cache on Linux and Windows through DMA malware, even if the memory addresses are random. [26] et al. also offer countermeasures to detect DMA attacks. [27] et al. integrate DMA attacks through FireWire into Metasploit [28] for payload selection, session control, etc. and attack via DMA over Firewire.

TRESOR-HUNT [29] relies on the insight that DMA-capable adversaries are not restricted to simply reading physical memory but can write arbitrary values to memory as well. Hard disk encryption keys were considered safe if not saved on the RAM, but TRESOR-HUNT [29] injects malicious code to the kernel using DMA attack and then extracts disk encryption keys from the CPU into the target system's memory from which they can be retrieved using a normal DMA transfer. [30] et al show that an adversary

with physical access to a device, could impersonate the device's memory controller, by attaching a malicious memory controller to the exposed pins of each DIMM socket of RAM and, by doing so, an attacker would have full access (READ/WRITE) to the target memory. Duflot et al. [31] introduce the vulnerability of remote code execution on a network adapter and how it could compromise the system-running kernel using DMA attack. BROADPWN [32] is a novel approach of privilege escalation. From exploiting a bug in Broadcom WiFi chip into DMA attack on the main processor of the device. The emerging of cache, DMA, and hardware attacks shows that not only software bugs can impose security risks but also hardware implementation bugs are becoming more common, specifically when new features rely on old security assumptions. [1] et al.show that because ARMv7 (the ARM debugging model) requires no physical access, a low-privilege host can use ARM debugging features to gain read/write access to TrustZone Secure World. Because there is no hardware privilege access control, a low-privilege host can initiate a debug session with a high-privilege target using ARM debugging features. [1] et al.use ARM debugging features to leak private keys from the Secure World, thus compromising ARM TrustZone security. The hardware implementation bugs of ARM debugging features affect development boards, IoT devices, and mobile devices. Defense against these vulnerabilities requires hardware and software solutions like the vulnerability we found. [1] et al.suggest that ARM should add restrictions in the interprocessor debugging model to enforce permission between host and target. SoC vendors should refine debug signal management, and add support to disable only inter-processor debugging. OEMs should add software-based access control to go with the hardware permission model. Matt Spisak et al. [33] describe another processor feature-based attack using ARM CoreSight debug features. [33] et al.leverage ARM PMU (Performance Monitoring Unit) to create a rootkit that cannot be detected by the kernel

monitor because it does not change the kernel syscall but rather attaches through the PMU to any syscall. Thus, every syscall will raise a PMU event, and the rootkit would be able to modify the input and output data of the syscall. This attack is possible due to a hardware implementation bug of a debug signal authorization that enables debug features in the hardware. [34] et al.suggest a different approach where the code and data that need to be protected are kept only in EL2 [35] (HYP mode) instead of in the TrustZone, where there is a strong coupling between vendor-specific code and hardware implementation; in which case, EL1 and EL0 will not have access to this code. Cloaker [36] et al.leverage ARM architecture System Control Register (SCTLR) to move the exception vector table (EVT) from high to low address so that mapping a malicious EVT at address 0x0 would intercept all exceptions.

Much is found in the literature on control-flow integrity (CFI). [37] et al.present the kernel CFI used to protect the kernel's stack and heap. A flaw in the kernel may allow user processes to write to kernel-space. Therefore, processor vendors presented the NX (Never Execute) bit that thwarts execution from the kernel's data portions. However, the execution segments were still writable and vulnerable to exploits. This led to making the kernel execution part read-only. But this also was not enough, as all of the user-space portion could be both written to and executed via a kernel exploit. To probity this, Intel created the supervisor mode execution prevention (SMEP) and ARM privileged execute never (PXN) bit. These features restrict the kernel from executing user-space memory while in kernel mode. This led attackers to target the stack, mainly manipulating the return addresses kept on the stack. This type of attack is referred to as 'return-oriented programming' (ROP) attacks. ROP attacks manipulate indirect calls, i.e. function pointers. These attacks concentrate on the calling (forward edge) and returning (backward edge) of a function. Thus, the main purpose of CFI is to try to ensure that forward edges

go to the expected addresses and that the backward edges are not changed. CFI is implemented through the Clang compiler extensions and utilizes link-time optimization (LTO) to examine the entire kernel code. Functions are classified according to their signature and checked in runtime. Another mechanism is kCFI, which narrows the classification of the edges. Thus, to use this feature, OP-TEE must be compiled with Clang and then apply kFCI on it. Unfortunately, none of these defenses thwart a DMA attack. In the area of thwarting hypervisor CFI attacks, [38] et al. offer Hypersafe. Hypersafe is used to protect the hypervisor from CF hijack attack through a memory lockdown and restricts pointer indexing, a layer of indirection that converts the control data into pointer indexes. These pointer indexes are restricted such that the corresponding call/return targets strictly follow the hypervisor control flow graph, hence expanding protection to control-flow integrity. This mitigation reduces the ease of performing a DMA attack on the hypervisor and, combined with IOMMUs, DMA attacks can be entirely mitigated.

4 The Attack – Return of the DMA

4.1 DMA Attack

Direct Memory Access (DMA) allows I/O devices to access the memory. DMA has evolved since its inception, when a single DMA controller was set in a computing system. Following the introduction of many high-speed I/O peripherals, devices started to incorporate DMA engines that enabled them to initiate DMA transactions without the coordination of a central DMA controller. ARM implements the advanced microcontroller bus architecture (AMBA), an open standard for on-chip interconnect specification. DMA transactions connect through the DMA controller to the on-SOC AMBA AXI Bus (AMBA advanced extensible interface) and the AMBA AXI Bus supports TrustZone NS-bit. The DMA controller can handle secure and non-secure events simultaneously, with full support for interrupts and peripherals. Examples of DMA devices are graphic cards, network adapters, FireWire, ThunderBolt, etc.

Although DMA is essential for fast I/O transactions, it also opens new vulnerabilities to DMA attacks [12], [29], [35]. This paper demonstrates a DMA attack on a poorly implemented TrustZone hardware architecture; without an SMMU (System Memory Management Unit) or ARM TZASC/TZPC and AXI-bus NS bit support, the system cannot prevent a DMA-capable device like Firewire/Thunderbolt from accessing the RAM.

4.2 Attack Goal

The secured memory is accessible through DMA transactions. Through this vulnerability, the TrustZone can be exploited. We escalate privileges by reading data from the Secure World.

Through this attack, we inject code to the Monitor in EL3, thus executing malicious programs in the Secure World operating system (the Secure World kernel). This offers us to bypass any validation of the secure operating system and also makes it possible to patch the EL1 kernel and execute arbitrary code.

4.3 The Attack – “Trusted” Arbitrary Code Execution

Attack primitive is based on Write What Where vulnerability achieved using DMA transactions and overriding the code in the RAM. We use this vulnerability to show that we can gain access to execute arbitrary code in the OP-TEE OS, thereby bypassing OP-TEE OS trusted application signature validation and gaining control of every trusted application in the system. Our approach is to change the return values of key functions without changing the stack. This technique impedes CFI tools such as gcc stack guard [36] or Clang [32] kFCI to detect our attack. Trusted applications are located on the REE file-system because this file-system usually contains more memory; by using this file-system, it is easier to update those applications. The trusted applications are built separately from the trusted operating system (similar to Linux kernel and user-space applications in the Normal World) and are signed with a private key from the manufacturer of the device application (e.g. Samsung sign their trusted applications with their private key). Common usages of trusted applications are DRM validations, HMAC (keyed-hash message authentication code) based one-time password, AES encryption and more. Using the trusted applications, the manufacturer of the device can make sure a compromised user or kernel will not break the integrity of the device. When the manufacturer wants to update a trusted application, they sign the new version with the same private key and distributes it to the users. When the Secure World OS executes a trusted application, if the signature is invalid, then a security error will occur and the program will not run. In our attack, we first use a DMA attack in order to read memory pages from the RAM. DMA attacks can be initiated from peripheral devices such as FireWire, PCI-connected devices (Network cards, GPU, etc.) as demonstrated by [28], [29], [12]. DMA attacks can also be initiated from the CPU if the CPU can access the

DMA controller. After reading memory pages from the RAM, we analyse the memory and compare it to ARM Trusted Firmware in order to locate similar functions. (Most of TrustZone software implementations are based on ARM Trusted Firmware, which makes reverse-engineering of the code simpler.) Moreover, there are some major vendors' secure OS (Trusted Execution Environment) in the market (QSEE, OPTEE) and we compare our memory dump to the compiled versions of those; by doing so, we can find the functions that validate trusted application signatures. Because none of the widely used TEE OS uses Address Space Layout Randomisation (ASLR) [37] we can use the address from our memory dump to override trusted applications signature validations with a DMA attack. After doing so, we can just replace any TA with our own malicious TA. Even though we do not know the correct signature private key, the TEE OS will succeed to validate our malicious TA.

4.4 Attack summary and vulnerable devices

Using the above method, it is possible to bypass the TA validation by replacing the signature key itself or by patching the validation function so there is no need to sign the TA at all. Using DMA attacks on the TrustZone gives a wide range of attack possibilities. In this paper, we show the usage of DMA attack to perform ACE (Arbitrary Code Execution); however, it is also possible to use this method to read arbitrary code from physical memory whereby a malicious user can access sensitive data. We also show that hardware implementation bugs are common even on security features like ARM TrustZone.

Table 1 List of vulnerable SoC

Manufacturer	**SoC**	**Device**
Texas Instrument	CC2538	Sensibo
HISILICON	Hi3518EV200	Security Cameras
HISILICON	Hi3519V101	Security Cameras
Amlogic	Meson3	Wifi Network Speaker
ST	STM32	Smart Vaccums

Table 2 presents a few SoCs of real IoT devices that lack some TrustZone hardware, but support TrustZone in the ARM Core, thus making the TrustZone 'untrusted'. One can claim the devices using this SoC do not use TrustZone at all; However, if this were the case, then those devices would not be using all the security options given to them, thus introducing architecture security flaws [38] [39] [40].

5 Attack Evaluation

5.1 Raspberry PI

Raspberry PI is a low-cost computer with wide range of uses, from its original target market of education to robotics, research projects and IoT devices (weather monitoring, smart home, smart camera, WIFI range extender and many more).

We use a Raspberry PI3 Model B to demonstrate the attack. The Raspberry PI3 Model B's main specifications are shown in Table 1.

Table 2: Raspberry PI 3 Specifications

SoC	Broadcom BCM2837
CPU	4 cores, ARM Cortex A53, 1.2GHz, (clocked to 700MHz)
RAM	1GB LPDDR2 (900 MHz)
Clock	19.2 MHz

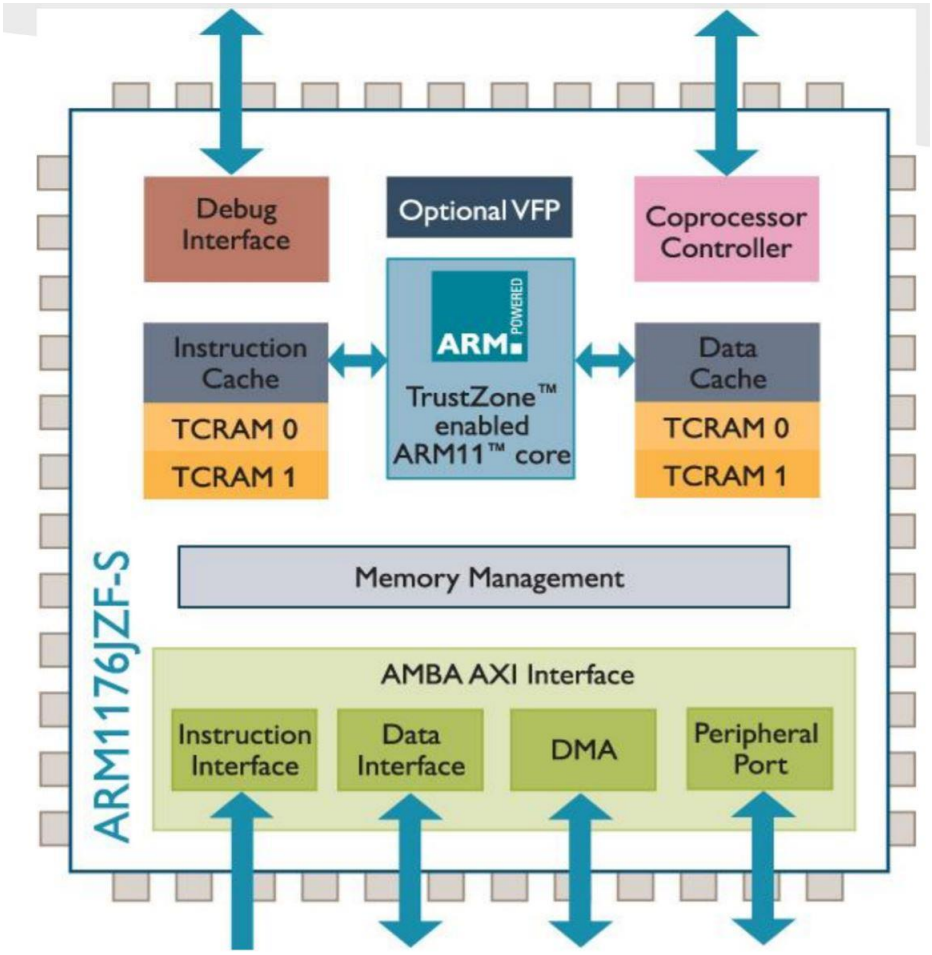

Figure 4: BCM2837 overview

Figure 4 presents the BCM2837 chip. This Broadcom SOC supports TrustZone and DMA transactions through the AMBA (Advanced Microcontroller Bus Architecture) AXI (Advanced Extensible Interface). As mentioned earlier, not all TrustZone cores comply with the entire hardware specifications. Figure 4 shows that the BCM2837 has the correct AXI bus, but it lacks the TZASC and TZPC, making it vulnerable to DMA attacks.

5.2 OP-TEE OS for Raspberry PI

OP-TEE supports Raspberry PI 3 Model B. The ARM Trusted Firmware is the basis for implementing Secure World software for the ARM A-Profile architectures (ARMv8-A and ARMv7-A), including an Exception Level 3 (EL3) Secure Monitor.

The (SMC) Secure Monitor Call instruction is used to invoke functions between the Normal World and the Secure World through the Secure Monitor (Figure 3). ARM Trusted Firmware for the Raspberry PI provides a suitable starting point for the productization of Secure World boot and runtime firmware [41]. When a vendor uses OP-TEE on any hardware in general, and on Raspberry PI specifically, they will most likely use a trusted application in order to implement hardware security measures and secure their devices [42].

All real-world environment file-system trusted applications need to be signed. The signature is verified by OP-TEE OS upon loading of the TA. Within the OP-TEE OS source is a directory key. The public part of the key (public key) will be compiled into the OP-TEE OS binary and the signature of each TA will be verified against this key upon loading using an RSA signature scheme [43]. A vendor must sign his trusted application with a private key; thus, if a malicious party tries to change the trusted applications on the file-system, the OP-TEE OS will return a security error and will not execute the malicious trusted application.

Without this mechanism, a malicious party would be able to alter trusted applications code, which will gain them access to the TrustZone security storage.

When a vendor updates a trusted application, they sign the new TA with their private key. OP-TEE OS contains the public key, thereby validating the TA. In this paper, we present a way to bypass this mechanism and execute our own 'trusted' applications.

5.3 Raspberry PI DMA

The DMA controller can be configured through the CPU as well as an external device. Therefore, we chose to perform this attack through the CPU. We authored a Linux kernel module to perform the DMA transactions. This module maps the DMA controller and configures the DMA control block to initiate DMA transactions.

In OP-TEE's Linux kernel, the DMA controller address space is not available to the user-space. However, it is plausible to assume that an IoT device, for example, will enable this device for peripherals access.

We argue an attack is possible in many IoT devices and we will show the following scenarios:

1. Some IoT devices mapping physical memory to the user-space to increase performance and save kernel access that may lead to DMA controller access.
2. Linux-based devices (IoT devices, routers, etc.) do not update their kernel versions very often due to compatibility issues and the large number of devices. Thus 'one day's' vulnerability can be used to exploit the device and gain root access to perform actions on the DMA controller [44] [45].
3. Attack peripheral device (Bluetooth/WIFI chip, SSD controller, etc…) to perform malicious DMA transactions [46], [47], [48].

All those scenarios may lead to a DMA attack and, on some devices, to a TrustZone vulnerability.

[49] presents the Control Block structure of a DMA and the DMA Controller registers in the Raspberry PI. To initiate a DMA transaction, we first set the Control Block structure and then set *CONBLK_AD* in the DMA controller structure. We perform two types of DMA transactions:

1. Set *SOURCE_AD* to the Secure World physical address in order to read data of the Secure World.
2. Set *DEST_AD* to the Secure World physical address in order to write malicious code to the Secure World, thereby achieving arbitrary code execution.

5.4 The Attack – Evaluation on Raspberry PI

In OP-TEE environment. Trusted applications are signed with the key from the build of the original OP-TEE core blob. Trusted applications consist of a signed ELF header, named from the UUID of the trusted application (set during compilation time) and the suffix ".ta". When a trusted application is replaced in the REE file-system with the new one, the signatures and UUID are validated by the OP-TEE OS (Figure 5).

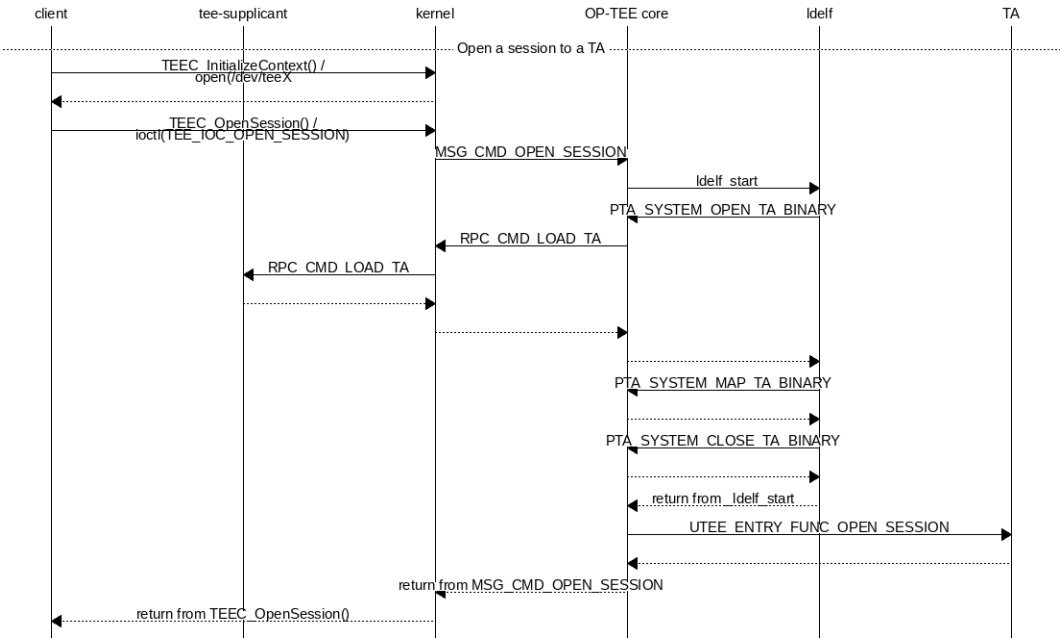

Figure 5:Open Session function Flow

Invoking a trusted application function from the Normal World requires the use of SMC (Secure Monitor Call). SMC is used to communicate between the Normal World and Secure World. SMC is initiated by the kernel (EL1) to reach the EL3 monitor.

OP-TEE provides a Linux kernel driver to interact with the OP-TEE in TrustZone. For instance, *PTA_SYSTEM_OPEN_TA_BINARY* function is accessed by this driver to the OP-TEE OS in the Secure World. *PTA_SYSTEM_OPEN_TA_BINARY* calls system *open_ta_binary*, which looks for the user-trusted application ELF by the UUID in the storage (file-system).

After finding the trusted application ELF in the REE file-system, the OP-TEE OS loads the ELF header and maps the trusted application sections into the secure memory using *PTA_SYSTEM_MAP_TA_BINARY*. After loading the trusted application, the user is able to invoke the trusted application functionality through the OP-TEE Linux kernel driver. We focus on two functions: *ree_fs_ta_open* and *ree_fs_ta_read* called by PTA *SYSTEM_OPEN_TA_BINARY* and *PTA_SYSTEM_MAP_TA_BINARY* respectively. Trusted applications binaries contain a signed header so that a malicious user cannot replace the trusted applications. If a malicious user replaces a trusted application, then OP-TEE OS returns a security error when it tries to execute those trusted applications.

In order for OP-TEE OS to validate those signatures, as a trusted application executes, the function *ree_fs_ta_open* loads the trusted application header, validates the application header signature (Figure 6), and validates its size (Figure 7).

```
1. /* Validate header signature */
2. res = shdr_verify_signature(shdr);
3. if (res != TEE_SUCCESS)
4.     goto error_free_payload;
```

Figure 6: ree_fs_ta_open Header signature validation

```
1. if (ta_size != offs + shdr->img_size) {
2.         res = TEE_ERROR_SECURITY;
3.         goto error_free_hash;
4. }
```

Figure 7: ree_fs_open TA size validation

When OP-TEE OS maps the TA into the secure memory, it loads the application to the memory using *ree_fs_ta_read*, which validates the encrypted trusted application signature (Figures 8 and 9).

```
1. if          (handle->shdr->img_type          ==
   SHDR_ENCRYPTED_TA) {
2.     /*
3.      *   Last   read:   time   to   finalize
   authenticated
4.      * decryption.
5.      */
6.     res     =     tee_ta_decrypt_final(handle-
   >enc_ctx,handle->ehdr, NULL, NULL, 0);
7.     if (res != TEE_SUCCESS)
8.         return TEE_ERROR_SECURITY;
9. }
```

Figure 8: ree_fs_ta_read decrypts a TA header

```
1.  /*
2.  * Last read: time to check if our digest
   matches the expected
3.  * one (from the signed header)
4.  */
5. res = check_digest(handle);
6. if (res != TEE_SUCCESS)
7.     return res;
```

Figure 9: ree_fs_read validates the encrypted header against the hash of the plain header

In the first step, we reverse-engineered OP-TEE OS (using radare2 [50]) in order to find key opcodes of both functions to exploit (Figures 6 - 9). We need to perform this step one time, because there is no ASLR we know the address of the opcode in each boot of the system. On different SoC we need to perform reverse-engineering method again just to find the relevant addresses. We used DMA transactions to read chunks of physical RAM in order to find the opcodes that match the functions above. Once we located the opcodes in the memory and noticed that these functions load in the same location in physical memory every time. We used DMA transactions to override the return values of the validations mentioned above (Figure 6 - 9), thereby gaining the ability to compile our own trusted application, sign it with an arbitrary key and execute it on the machine. We replaced two types of opcodes: the comparison opcode of *w0* register was replaced with *cmp w0,w0* so it always returned true and, when moving the return value of the function to *w0* register, we replace this command with

eor w0,w0,0 so the value of *w0* register would be 0, again having the return values of the validation functions equal true.

We were able to perform this replacement using just a simple DMA transaction with the control block *DEST_AD* set to the physical address of the opcodes we found; all of which constitute in the Secure World memory. Because Raspberry PI does not support physical bus Secure World separation, we used a DMA transaction to replace the correct opcodes with our malicious opcodes.

The trusted applications binaries are on the REE file-system and, thus, can be over-written; however, the OP-TEE OS will never execute a replaced TA when the signature does not match.

Faking a matching signature requires finding a private key of 2048-bit that matches the public key; therefore, only the owner of the key would be able to replace those applications. In our case, we compiled a new TA with the same UUID of the original one and put it in the file-system location. By executing our malicious TA, we gained the ability to manipulate ARM TrustZone to execute invalid signed binaries.

For example, we compiled a fake AES TA (given in the examples of the OP-TEE suite) that encrypts data with our malicious key. Thus, every time the user uses this TA to perform AES, the data will not be truly encrypted with a secret key.

6 Mitigations

When choosing a SoC for a device you must compare the device requirements to the features the SoC contains. In our case when choosing a SoC we want to make sure the SoC architecture has all the chips required for ARM TrustZone to work properly (TZASC, TZPC, supported bus, etc…). The process of checking the SoC architecture is not always easy and surely not automatic, because not all vendors publish their SoC architecture.

We suggest for SoC vendors to be more transparent about their architecture when it comes to security features. We also suggest manufacturers to ensure that their SoC hardware supports not only TrustZone ARM Core but also TrustZone specification.

In cases where a fully compatible TrustZone is not available (Lack of hardware on the SoC which makes the TrustZone secure), we list other protection techniques:

• Using SMMU (similar to IOMMU on Intel x86) to configure specific addresses for DMA controllers. SMMU works as MMU for BUS access, so any memory access through the BUS would have to be matched to the permission configured to the accessed address. With SMMU and a correct configuration a DMA attack through peripheral will not be possible. It is also important to note a kernel attacker could change this configuration.

• In the case of Raspberry PI, by disabling the DMA controller, a non-privileged user or peripheral would not be able to use DMA transactions.

• Set the Secure World on a different RAM without DMA controller mapping so there is no physical interface between the Normal and Secure Worlds.

A software protection would be to encrypt parts of the OP-TEE code itself, mainly the TA decipher functions. Only when these functions are used, OP-TEE will decrypt the

functions into the cache, validate the TA, and evict the cache. This method was introduced by [24]. Using this method an attacker would have to time his attack in order to get the code from the RAM. Combine this method with ASLR and HALT the other cores and this attack will be mitigated.

7 Summary and Conclusions

7.1 Summary

We live in an era of ever-growing IoT devices and connectivity demands. Connectivity between our home, cars, smartphones, etc… impose privacy risk in case of vulnerabilities. Those privacy risks impose a great threat to our day to day life, from digital impersonating, data/money stealing, and more.

To mitigate those risks IoT/Smartphone devices vendors invest more and more in security mechanisms. In particular, hardware-supported security mechanism, for instance, ARM TrustZone, which allows running secure applications and using secure storage so normal world vulnerability (kernel or user space) won't cause leakage of secure information, for instance, encryption keys of encrypted on-device-data.

Since ARM TrustZone is not a single chip but rather an entire architecture, many ARM SoC manufacturers (Huawei, Broadcom, Qualcomm, etc…) may introduce vulnerability to their SoC which will compromise the entire TrustZone security suite. Zhenyu Ning and Fengwei Zhang, et al. [1] showed how manufacturers fault implementation of debug signals impose vulnerability which gives access from normal world to the secure world and compromises ARM TrustZone security. In our approach we assembled a malicious DMA transaction to read data from the secure world such as encryption keys so encrypted on-device-data become compromise. Also using DMA transaction, we gain privilege to execute malicious code in the secure world, so we bypass trusted OS signature validations and by doing so we obtain full control on the trusted applications.

Our attack let us create a malicious trusted application that can impersonate to a valid one, even though our application signature will not match, by changing 4 opcodes. Using

the DMA transaction, we manage to bypass those validations. We evaluate our attack on Raspberry PI 3 model B, and we show how we maliciously replace AES key generator trusted application to return a key. Such that every use of this trusted application is compromise and we could maliciously decrypt all the data encrypted with this key. Our main contribution is to show that even though we have an architecture which supports hardware security features it is still responsibility of the developers to make sure this architecture meets the security standard of ARM and making sure they configure the secure memory regions correctly. It is important to note that our attack can work on devices such as i.MX53 even though the TZASC component is part of the architecture because both peripherals IPU and GPU share the same DMA channel so if the IPU can access the secure world, the GPU can access the secure world also.

Making sure a device does not have TrustZone hardware related vulnerabilities requires comprehensive design review. We call SoC manufacturer to invest more in their design reviews and developers to make sure to configure the secure world properly.

7.2 Future Work

In this work, we present an attack vector on ARM TrustZone secure world caused by hardware architecture vulnerability or misconfiguration of secure memory regions. This work can be further extended in several directions:

1. **Scope expansion.** In this work, we focus on hardware architecture vulnerability but due to lack of software configuration validations in the TrustZone it is common to find software misconfiguration of the secure world memory.
2. **Hardware Vulnerabilities in ARM TrustZone architecture.** In this work we mainly focus on the lack of *TZASC* component. Relying on previous work of Zhenyu Ning and Fengwei Zhang et al. [1] and Matt Spisak et al. [34] we can use

different approaches by leveraging ARM debug features to enable DMA attack vectors on the TrustZone on different platforms. Those attack vectors will not require a lack of component in the hardware design.

8 Bibliography

[1] Z. Ning and F. Zhang, "Understanding the security of arm debugging features," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[2] M. Zhang, Q. Zhang, S. Zhao, Z. Shi and Y. Guan, "Softme: A software-based memory protection approach for tee system to resist physical attacks," *Security and Communication Networks*, vol. 2019, 2019.

[3] D. Papp, Z. Ma and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," in *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, 2015.

[4] J. Leonard, *Why TrustZone Matters for IoT.*

[5] "ARM TrustZone," [Online]. Available: https://developer.arm.com/ip-products/security-ip/trustzone.

[6] S. Zhao, Q. Zhang, Y. Qin, W. Feng and D. Feng, "Minimal Kernel: An Operating System Architecture for {TEE} to Resist Board Level Physical Attacks," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, 2019.

[7] D. Shen, "Exploiting TrustZone on android," *Black Hat USA*, 2015.

[8] Y. Chen, Y. Zhang, Z. Wang and T. Wei, "Downgrade attack on trustzone," *arXiv preprint arXiv:1707.05082*, 2017.

[9] S. Makkaveev, *The Road to Qualcomm TrustZone Apps Fuzzing.*

[10] J. Guilbon, *Attacking the ARM's TrustZone.*

[11] "About the AXI TrustZone memory adapter," [Online]. Available: https://developer.arm.com/docs/dto0017/a/about-the-axi-trustzone-memory-adapter.

[12] G. Kupfer, D. Tsafrir and N. Amit, "IOMMU-resistant DMA attacks," 2018.

[13] "Open Portable Trusted Execution Environment," [Online]. Available: http://www.op-tee.org/.

[14] C. Göttel, P. Felber and V. Schiavoni, "Developing secure services for IoT with OP-TEE: a first look at performance and usability," in *IFIP International Conference on Distributed Applications and Interoperable Systems*, 2019.

[15] L. Davi, P. Koeberl and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014.

[16] M. a. S. R. Zhang, "Control flow integrity for COTS binaries," *22nd USENIX Security Symposium USENIX Security 13*, pp. 337-352, 2013.

[17] "A Technical report on TEE and ARM TrustZone," [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-technical-report-on-tee-and-arm-trustzone.

[18] O. Blazy and C. Y. Yeun, Information Security Theory and Practice: 12th IFIP WG 11.2 International Conference, WISTP 2018, Brussels, Belgium, December 10–11, 2018, Revised Selected Papers, Springer International Publishing, 2019.

[19] B. Ngabonziza, D. Martin, A. Bailey, H. Cho and S. Martin, "TrustZone Explained: Architectural Features and Use Cases," in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, 2016.

[20] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016.

[21] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014.

[22] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl and T. Eisenbarth, "AutoLock: Why Cache Attacks on {ARM} Are Harder Than You Think," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017.

[23] J. Demme, R. Martin, A. Waksman and S. Sethumadhavan, "Side-channel vulnerability factor: A metric for measuring information leakage," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[24] R. B. Yehuda and N. J. Zaidenberg, "Protection against reverse engineering in ARM," *International Journal of Information Security*, vol. 19, p. 39–51, 2020.

[25] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon and P. Sewell, "Modelling the ARMv8 architecture, operationally: concurrency and ISA," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[26] G. Irazoqui, T. Eisenbarth and B. Sunar, "Cross processor cache attacks," in *Proceedings of the 11th ACM on Asia conference on computer and communications security*, 2016.

[27] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[28] M. van Dijk, S. K. Haider, C. Jin and P. H. Nguyen, "Advanced Power Side Channel Cache Side Channel Attacks DMA Attacks".

[29] P. Stewin and I. Bystrov, "Understanding DMA malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2012.

[30] R. Breuk and A. Spruyt, "Integrating DMA attacks in Metasploit," in *Sebug: http://sebug. net/paper/Meeting-Documents/hitbsecconf2012ams D*, 2012.

[31] D. Kennedy, J. O'gorman, D. Kearns and M. Aharoni, Metasploit: the penetration tester's guide, No Starch Press, 2011.

[32] J. Moreira, S. Rigo, M. Polychronakis and V. P. Kemerlis, "DROP THE ROP fine-grained control-flow integrity for the Linux kernel," *Black Hat Asia*, 2017.

[33] Z. Wang and X. Jiang, "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," in *2010 IEEE Symposium on Security and Privacy*, 2010.

[34] M. Spisak, "Hardware-Assisted Rootkits: Abusing Performance Counters on the ARM and X86 Architectures," *10th USENIX Workshop on Offensive Technologies WOOT 16*, 2016.

[35] E.-O. Blass and W. Robertson, "TRESOR-HUNT: attacking CPU-bound encryption," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[36] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.," in *USENIX security symposium*, 1998.

[37] K. Cook, "Kernel address space layout randomization," *Linux Security Summit*, 2013.

[38] D. H. J. W. Christian Lesjak, "Hardware-Security Technologies for Industrial IoT:," *IECON 2015-41st Annual Conference of the IEEE Industrial Electronics Society*, pp. 002589--002595, 2015.

[39] P. L. X. X. X. G. S. Z. M. Y. T. J. Le Guan, "TrustShadow: Secure Execution of Unmodified," *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 488--501, 2017.

[40] T. G. J. P. J. C. A. T. Sandro Pinto, "IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices," *IEEE Internet Computing*, vol. 21,

pp. 40-47, 2017.

[41] "ARM Trusted Firmware," [Online]. Available: https://github.com/ARM-software/arm-trusted-firmware.

[42] A. Nehal and P. Ahlawat, "Securing IoT applications with OP-TEE from hardware level OS," in *2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA)*, 2019.

[43] R. L. Rivest, A. Shamir and L. M. Adleman, *Cryptographic communications system and method*, Google Patents, 1983.

[44] N. Hampton, "The working dead: The security risks of outdated linux kernel," 6 March 2017. [Online]. Available: https://www2.computerworld.com.au/article/615338/working-dead-security-risk-dated-linux-kernels/.

[45] J. Wallen, "Most IoT devices are an attack waiting to happen, unless manufactureres update their kernels," 28 June 2017. [Online]. Available: https://www.techrepublic.com/article/most-iot-devices-are-an-attack-waiting-to-happen-unless-manufacturers-update-their-kernels/.

[46] Y.-A. P. G. V. O. L. Loic Duflot, "Can you still trust your network card?," *CanSecWest*, pp. 24--26, 2010.

[47] R.-P. Weinmann, "Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular," *WOOT*, pp. 12--21, 2012.

[48] Intel, "The latest security information on Intel products," 6 9 2020. [Online]. Available: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00266.html.

[49] in *BCM2837 ARM Peripherals*, 2012, pp. 40-41.

[50] "Libre and Portable Reverse Engineering Framework," [Online]. Available: https://radare.gitbooks.io/radare2book/content/.

[51] J. Criswell, N. Dautenhahn and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *2014 IEEE Symposium on Security and Privacy*, 2014.

9 תקציר

ARM TrustZone מציע סביבת ריצה בטוחה כחלק מהחומרה של המעבד. ARM TrustZone נמצא בשימוש רחב ברכיבים עם מעבדי ARM כגון: טלפונים סלולאריים, רכיבי IoT, ורכיבי embedded. עקב גדילה במודעות לאבטחת מידע ובחשיבות הפרטיות, ARM מימשו את ה-TrustZone המאפשר ליצרניות להוסיף זיכרון מאובטח ואפליקציות מאובטחות למערכת, זאת כדי לבצע פעולות צופן כגון יצירת מפתחות הצפנה, שמירת טביעת האצבע של המשתמש בזיכרון מאובטח, הגנת DRM כדי למנוע גניבת מידע עם זכויות יוצרים וכו'...

למרות ש-ARM TrustZone יכול לשפר את רמת האבט

חה במערכת ולהגן על מידע מתוקף עם הרשאות גבוהות, המימוש החומרתי של ה-TrustZone תלוי ביצרן ה-SoC ועליו להטמיע את ה-TrustZone לפי הארכיטקטורה של ARM עם כל הרכיבים הנלווים לכך כגון TZASC, AXI-BUS תומך ועוד... במקרים בהם היצרן מחסיר אחד או יותר מהרכיבים הנלווים ל-TrustZone עלולות להיווצר פגיעויות ב-TrustZone שיפגעו בתכונות האבטחה שלו.

התרומה של עבודה זו היא כדלקמן. ראשית אנו מציגים איך ARM TrustZone עובד ומה רכיבי החומרה המרכזיים שדואגים לכך שתכולות האבטחה של ARM TrustZone ישמרו. שנית אנחנו מבצעים הדגמה של פגיעויות שנגרמות עקב יצרניות שלא עמדו במפרט של ARM למימוש מאובטח של ה-TrustZone. התקיפה שלנו מבוססת על DMA כך שניתן לבצע אותה מרכיבים היקפיים למערכת כגון: כרטיס רשת, כרטיס מסך וכו'..., אנחנו מראים גם שניתן לבצע אותה מהמעבד עצמו בעזרת גישה ל-DMA Controller. התקיפה מורכבת מזיהוי הקוד הרלוונטי ב-RAM שאחראי על ווידוא החתימה של האפליקציות הבטוחות (Trusted Applications), זיהוי הקוד דורש פעולת reverse engineer אך כיוון שרוב הרכיבים משתמשים באותם מערכות הפעלה ב-Secure World, פעולת ה-RE הופכת פשוטה. לאחר מכן ניתן בעזרת טרנזקצית DMA לדרוס את הקוד שמוודא את חתימת האפליקציות המאובטחות.

במחקר זה נציג איך להשיג שליטה מלאה על הזיכרון והאפליקציות המאובטחות תוך כדי ריצה בהרשאות נמוכות, נראה עד כמה החולשה נפוצה, נציג ניצול פשוט של החולשה על חומרות שונות ונציין רשימה של רכיבי IoT פגיעים.

עבודה זו בוצעה בהדרכתו של דר' נצר זידנברג ועזרתו של רז בן יהודה מאוניברסיטת יובסקולה, פינלנד.

המרכז הבינתחומי בהרצליה
בית-ספר אפי ארזי למדעי המחשב
התכנית לתואר שני (M.Sc.) - מסלול מחקרי

תקיפה על ARM TrustZone על ידי ניצול חולשה חומרתית

מאת
רון שטיינרוד

עבודת תזה המוגשת כחלק מהדרישות לשם קבלת תואר מוסמך M.Sc.
במסלול המחקרי בבית ספר אפי ארזי למדעי המחשב, המרכז הבינתחומי הרצליה

פברואר 2021