Journal of Computer Virology and Hacking Techniques
https://doi.org/10.1007/s11416-021-00413-y

ORIGINAL PAPER

Attacking TrustZone on devices lacking memory protection

Ron Stajnrod[1] · **Raz Ben Yehuda**[2] · **Nezer Jacob Zaidenberg**[2,3]

Received: 11 August 2021 / Accepted: 26 November 2021

Abstract
ARM TrustZone offers a Trusted Execution Environment (TEE) embedded into the processor cores. Some vendors offer ARM modules that do not fully comply with TrustZone specifications, which may lead to vulnerabilities in the system. In this paper, we present a DMA attack tutorial from the insecure world onto the secure world, and the design and implementation of this attack in a real insecure hardware.

Keywords TrustZone · Security

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 [56], with embedded systems as the driving force for technological development in many domains in the emerging post-PC era. As an increasing number of computational devices integrate into our lives in a pervasive and invisible way, security becomes critical for the dependability of all intelligent systems built upon these embedded systems [40].

Embedded IoT products are increasingly not connected to the power grid. Therefore, such devices are constrained in terms of computing power due to limited electric power [29]. The constrained nature of such devices means we are trying to "build a fortress from pebbles." Therefore, we must take the 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.).

✉ Nezer Jacob Zaidenberg
scipio@scipio.org

Ron Stajnrod
ronst12@gmail.com

Raz Ben Yehuda
raziebe@gmail.com

[1] Interdisciplinary Center, Herzliya, Israel

[2] University of Jyväskylä, Jyväskylä, Finland

[3] College of Management Academic Studies, Israel University of Jyväaskylä, Rishon LeZion, Israel

ARM TrustZone [5] was introduced as part of the ARMv6 architecture and is widely used in smartphones, tablets, wearables, and other devices. As TrustZone gains popularity in hardware security architecture for mobile devices and IoT, it is vital to ensure the security of TrustZone itself [57].

Though ARM TrustZone is a great way to implement security mechanisms across IoT-embedded devices, it is still prone to inadequate 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 [11,21,33,43].

One of the key features of the AMBA (Advanced Microcontroller Bus Architecture) AXI (Advanced Extensible Interface) [3] is address space separation. Thus, lacking them creates insecure memory separation between the normal world and the secure world. In this paper, we present a Direct Memory Access (DMA) attack [26] on OP-TEE (Open Portable Trusted Execution Environment) [18,39]. OP-TEE is one of the common operating systems that run on TrustZone. OP-TEE is a popular, open-source, TrustZone operating system. We used OP-TEE as a reference TrustZone OS to demonstrate the attack presented in this paper, though the attack is not due to an OP-TEE bug but rather a missing hardware feature.

Our attack 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 [14,55] on the OP-TEE kernel.

Also in the paper, we show a hardware vulnerability on SoC [10] that compromises ARM TrustZone. Using the

Published online: 20 December 2021

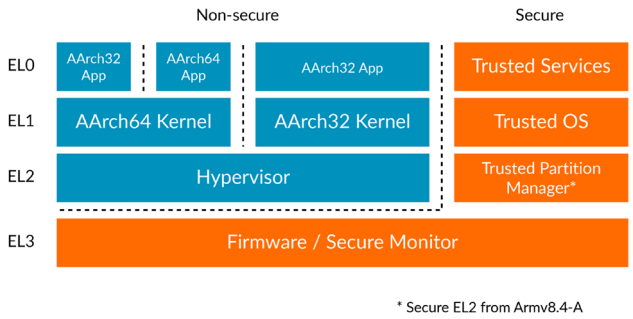

Fig. 1 Normal and secured worlds ©Arm

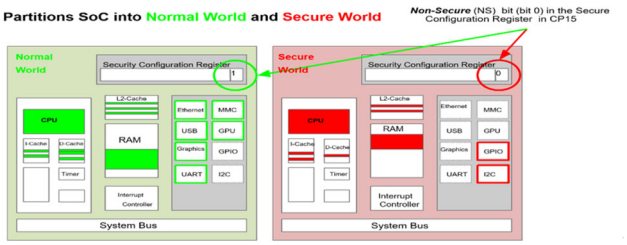

Fig. 2 NS Bit ©Linaro

DMA attack, we gain the ability to replace trusted applications with malicious ones. Furthermore, 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. The attack was not possible when AMBA AXI was present. Unfortunately, the AMBA AXI is not present on a few modern hardware devices, including Raspberry Pi 3,4 and Jetson Nano.

2 Background

2.1 ARM TrustZone

ARM TrustZone technology aims to establish 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' (Fig. 1). The distinction between these worlds is entirely orthogonal to the standard ring protection between user-level and kernel-level code, and hidden from the operating system running in the normal world [1].

As an example, the Linux kernel runs in EL1 and the userspace processes execute in EL0. The separation of the secure and normal worlds protects specific RAM ranges and peripherals only accessible by the secure world. This separation means that a compromised normal world code (in the userspace or the kernel) cannot access these memory ranges or devices. However, this separation is entirely artificial. The same cores run both secure and normal worlds, and they use the same RAM (Fig. 2). The Non-Secure (NS) bit determines whether the CPU executes in the normal world or in the 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 (Fig. 2), 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. Third-party companies implement the SoC peripherals; therefore, some vendors do not comply entirely with TrustZone specifications to reduce costs.

It is possible to access the entire memory from the secure world but not vice versa. To traverse to EL3, we use the Secure Monitor Call (SMC) instruction. Unfortunately, the SMC implementation depends on the manufacturer and, thus, is prone to bugs and other vulnerabilities [21]. 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. In addition, these controllers provide a secure I/O to peripherals over standard interfaces. For instance, the TZPC routes the SPI access to the secure world. Furthermore, the NS bit secures on-chip peripherals from accessing from the Rich Execution Environment (REE) [8]. TZASC utilizes the NS bit for a memory-mapped device like DRAM. These two devices require support from the AXI bus, which is vendor-specific.

Examples of the secure world trusted applications are secure PIN and biometric checks. Another trusted application use case is Digital Right Management (DRM) for online media. Again, private information is kept within the secure world so hackers cannot access the keys required to reverse-engineer the system. [36] describes many more use cases of TrustZone for IoT and mobile devices.

2.2 OP-TEE

OP-TEE [39] 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. The GlobalPlatform API defines

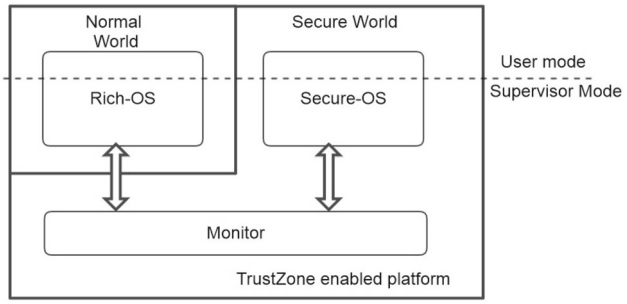

Fig. 3 Outline of TrustZone ©Miledropedia

these specifications [22]. The non-secure OS is referred to as the Rich Execution Environment (REE) in TEE specifications.

OP-TEE is widespread in Snapdragon, IMX7, Hikey, DragonBoard, and many other products.

OP-TEE is designed primarily to rely on the ARM TrustZone technology as the underlying hardware isolation mechanism. However, it is compatible with other isolation technologies 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 between the REE to the secured OS, secured interrupts RPC from the secure world to the REE and the SMC interface communication from the REE to the secure world.

OP-TEE main components are the:

- OP-TEE binary OS executing in secure EL1 (TrustZone);
- OP-TEE Linux kernel driver;
- Linux userspace libraries; and
- a Linux userspace daemon (tee-supplicant) that performs services on behalf of the OP-TEE OS. It is responsible for passing the secured part of the TA into the secure world.

There are several types of TAs. The early TAs hard-link to OP-TEE core binary and, therefore, are available before the REE runs. The second type is an REE File system TA, which is available once the REE OS runs. The TA is composed of two parts: secured and non-secured. It is signed and may be encrypted. The key used to sign the TA is the same key used to sign the original OP-TEE binary core blob used when built.

Each TA is composed of two parts: a Linux userspace application and a secure world application. TA execution follows the next steps:

1. *Initialize Context* and *Open Session*: Creates a context and loads the TA secure part into the OP-TEE core in the TrustZone.
2. *Invoke command*. The non-secure part sends commands to the secured TA receiver.
3. *Close session* and *Finalize context*.

OP-TEE secure storage manager implements a secure file system in two ways: REE-FS and protected memory (if possible). When a TA writes data to the secured storage, the trusted storage invokes TEE file system operations to store the data. Then the TEE file system encrypts the data and passes the data to tee supplicant, keeping the data in the REE file system. The TEE file system is visible in the Linux file system as a directory. Each object within the TEE is assigned an internal identifier in addition to the TA objects.

Kept in the TEE file system, the key-manager responsibilities are data encryption and decryption. It uses three types of keys:

- **SSK** - Secure storage keys
- **TSK** - TA storage keys
- **FEK** - File system key

The FEK derives from TSK, which derives from the SSK. The SSK derives from the unique hardware key (HUK). The HUK may not be accessible from the REE and it is up to the manufacturer to provide it, and provide access to it.

The OP-TEE kernel is not encrypted. Therefore, a DMA attack on the OP-TEE kernel is more straightforward than on a TA-encrypted program as it bypasses the MMU permissions model and the need to encrypt the code.

Each TA is signed and optionally encrypted with a private key. 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 The DMA attack

Direct Memory Access (DMA) allows I/O devices to access the memory. DMA has evolved since its inception and after introducing many high-speed I/O peripherals, vendors started to incorporate DMA engines to initiate DMA transactions without coordinating with the 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 the TrustZone NS-bit. Thus, 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 [7,26,46].

3.1 Attack goal

On a SOC lacking an SMMU (System Memory Management Unit) or TZASC, running OP-TEE without NS-bit support, the secured memory is accessible through DMA transactions. Through this vulnerability, we can exploit TrustZone. We escalate privileges by reading data from the secure world. In this attack, we inject code to the Monitor in EL3, thus executing malicious programs in the secure world OS. This injection lets us bypass any validation of the secure operating system and makes it possible to patch the EL1 kernel and execute arbitrary code.

3.2 The attack - 'trusted' arbitrary code execution

We base the attack primitive on *Write What Where* vulnerability achieved using DMA transactions. We use this vulnerability to show that we can gain access to execute arbitrary code in the OP-TEE OS. We bypass OP-TEE OS TA signature validation and gain control of every trusted application in the system, which we present later in the paper. Our approach is to change the opcodes that return error values of key functions without changing the stack. This technique impedes CFI tools such as gcc compiler stack guard [13], Clang CFI [2], or kFCI [34] to detect our attack.

Trusted applications are located on the REE file system because it usually contains more memory; using this file system makes it easier to update those applications. The trusted applications are built separately from the trusted operating system (similar to Linux kernel and userspace 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). Typical 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 device's manufacturer can ensure a compromised user or kernel will not break the device's integrity. When the manufacturer wants to update a trusted application, they sign the new version with the same private key and distribute it to the users. When the secure world OS executes a trusted application, a security error will occur if the signature is invalid and the program will not run.

Table 1 PI3 specifications

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

In our attack, we first use a DMA attack to read memory pages from the RAM. Peripheral devices such as FireWire, PCI-connected devices (Network cards, GPU, etc.) can initiate a DMA attack, as demonstrated by [46,49], and [26]. The CPU can also initiate DMA attacks by activating the DMA controller. After reading memory pages from the RAM, we analyse the memory and compare it to ARM Trusted Firmware to locate similar functions. (Most TrustZone software implementations are based on ARM Trusted Firmware, making reverse-engineering the code simpler.) Moreover, some significant vendors' secure OS (Trusted Execution Environment) is in the market (QSEE, OP-TEE). 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 in some cases, some of the widely used TEE OS uses Address Space Layout Randomisation (ASLR) [12], 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. Thus, even though we do not know the correct signature private key, the TEE OS will succeed to validate our malicious TA.

4 Attack evaluation

4.1 Raspberry Pi platform

We use a Raspberry PI3 Model B to demonstrate the attack. Table 1 presents the Raspberry PI3 Model B's main specifications.

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 vendors' implementations comply with the entire hardware specifications. For example, Fig. 4 shows that the BCM2837 has the correct AXI bus, but it lacks the TZASC and TZPC, making it vulnerable to DMA attacks.

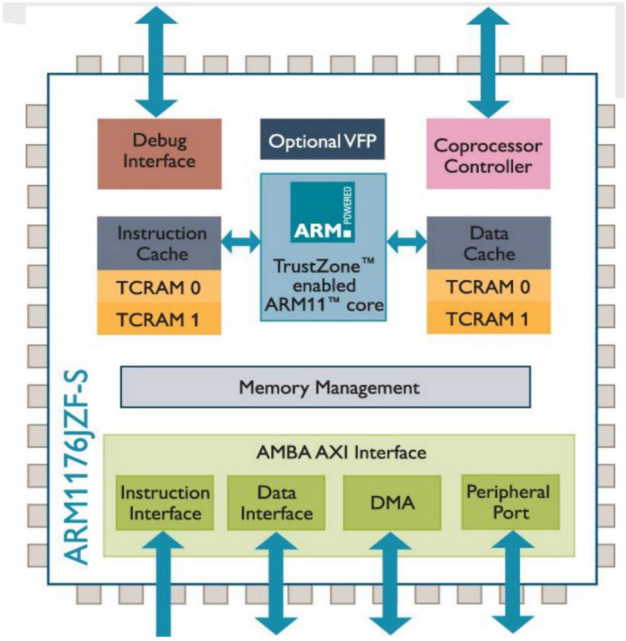

Fig. 4 BCM2387 Overview ©Premier Farnell Ltd

Table 2 DMA Control Block Data Structure

32-bit Word Offset	Description	Associated Read-only Register
0	Transfer Information	TI
1	Source Address	SOURCE_AD
2	Destination Address	DEST_AD
3	Transfer Length	TXFR_LEN
4	2D Mode Stride	STRIDE
5	Next Control Block Address	NEXTCONBK
6–7	Reserved - set to zero	N/A

4.2 OP-TEE for PI

OP-TEE supports *Raspberry Pi 3 Model B*. In addition, 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. ARM Trusted Firmware for the Raspberry Pi provides a suitable starting point for the productisation of secure world boot and runtime firmware [4]. When a vendor uses OP-TEE on any hardware in general and on Raspberry Pi specifically, they will most likely use a trusted application to implement hardware security measures and secure their devices [35].

Table 3 DMA Controller

32-bit Address offset	Register name	Description
0	CS	DMA Channel Control and Status
1	CONBLK_AD	DMA Channel Control Block Address
2	TI	DMA Channel Transfer Information
3	SOURCE_AD	DMA Channel Source Address
4	DEST_AD	DMA Channel Destination Address
5	TXFR_LEN	DMA Channel Transfer Length
6	STRIDE	DMA Channel 2D Stride
7	NEXTCONBK	DMA Channel Next CB Address
8	DEBUG	DMA Channel Debug

4.3 Raspberry Pi DMA

As noted earlier, the CPU can access the DMA controller. 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 userspace. 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 map physical memory to the userspace to increase performance and save kernel access, leading 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 many devices. Thus one day's vulnerability can be used to exploit the device and gain root access to perform actions on the DMA controller [37,50].
3. Attack peripheral device (Bluetooth/WIFI chip, SSD controller, etc.) to perform malicious DMA transactions [17,47,52].

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

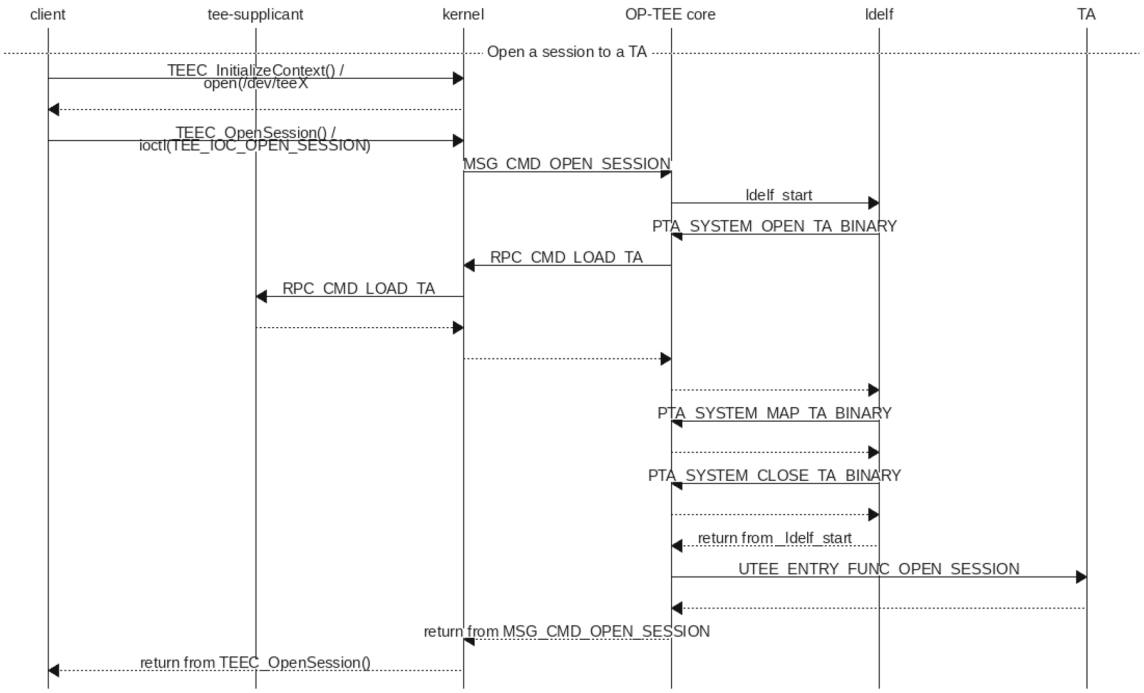

Fig. 5 Open session flow

Table 2 presents the Control Block structure of a DMA in the Raspberry Pi. Table 3 shows the DMA Controller registers. 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 to read data of the secure world.
2. Set *DEST_AD* to the secure world physical address to write malicious code to the secure world, thereby achieving arbitrary code execution.

5 The attack - evaluation on raspberry Pi

In the 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 (Fig. 5).

OP-TEE provides a Linux kernel driver to interact with the OP-TEE in TrustZone. For instance, the *PTA_SYSTEM _OPEN _TA_BINARY* function access the OP-TEE OS. *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 TA 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 executing 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 (Fig. 6) and validates its size (Fig. 7). 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 (Figs. 8 and 9).

In the first step, we reverse-engineered OP-TEE OS (using radare2 [31]) in order to find *key* opcodes of both functions to exploit (Figs. 6, 7, 8, 9). 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

```
/* Validate header signature */
res = shdr_verify_signature(shdr);
if (res != TEE_SUCCESS)
  goto error_free_payload;
```

Fig. 6 ree_fs_ta_open Header signature validation

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

Fig. 7 ree_fs_open TA size validation

```
if (handle->shdr->img_type ==
SHDR_ENCRYPTED_TA) {
  /*
   * Last read: time to finalise
authenticated
   * decryption.
   */
  res = tee_ta_decrypt_final(handle->
enc_ctx,handle->ehdr, NULL, NULL, 0);
  if (res != TEE_SUCCESS)
    return TEE_ERROR_SECURITY;
}
```

Fig. 8 ree_fs_ta_read decrypts a TA header

same location in physical memory every time. We used DMA transactions to override the return values of the validations mentioned above (Figs. 6, 7, 8, 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 replaced this command with *eor w0,w0,w0* 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*, which contains the physical address of the opcodes we found, all of which are located in the secure world memory. In our case, we compiled a new TA with the same UUID as 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 run invalidly signed binaries. For instance, 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, it will not encrypt the data with the secret key.

```
/*
 * Last read: time to check if our
digest matches the expected
 * one (from the signed header)
 */
res = check_digest(handle);
if (res != TEE_SUCCESS)
  return res;
```

Fig. 9 ree_fs_read validates the encrypted header against the hash of the plain header

5.1 Other attack possibilities

Using DMA attacks on the TrustZone gives a wide range of attack possibilities. In this paper, we show the usage of DMA attacks 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.

6 Mitigation

When choosing an SoC, you must compare the device requirements to the SoC features. In our case, when selecting an SoC, we want to make sure the SoC architecture has all the chips required for ARM TrustZone to work correctly (TZASC, TZPC, supported bus, etc.). Unfortunately, checking the SoC architecture is not always easy and not automatic because not all vendors publish their SoC architecture. We suggest that SoC vendors be more transparent about their architecture when it comes to security features. We also recommend that manufacturers ensure their SoC hardware supports TrustZone ARM Core and TrustZone specifications. In cases where a fully compatible TrustZone is not available (lack of hardware on the SoC that 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 is matched to the permission configured to the accessed address. With SMMU and a correct configuration, a DMA attack through peripherals 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 technique would encrypt parts of the OP-TEE code itself, mainly the TA decipher functions. Then, when OP-TEE runs these functions, it decrypts them into the cache, validates the TA, and evicts the processors' cache. Using this method [54], an attacker would have to time his attack to get the RAM code. However, combining this method with ASLR impedes the attack.

7 Related work

In the area of ARM, [11] 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 [11], the above applies to the OP-TEE and QSEE (Qualcomm's Secure Execution Environment). [11] 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. [11] et al. describe another possible rollback attack on the chain of trust and proves it possible to downgrade the bootloader successfully.

Many words have been written on side-channel attacks and other vulnerable targets in ARM architecture in prior research. For example, Armageddon [32] 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 to perform cycle-timing without root access, the Armageddon [32] et al. provide a method to gain sensitive information such as inter-keystroke timings or the length of a swipe action. As for TrustZone vulnerability, Armageddon [32] et al. shows a cache attack used to monitor cache activity caused within the ARM TrustZone from the normal world.

Flush and Reload attack [53] 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 [19] tool assesses the actual risk in cache attacks, prevents cross-cache evictions and highlights the intricacies of cache attacks in ARM. [19] et al. claim that unlike Intel processors, many ARM caches are both inclusive and exclusive and, therefore, harden the LLC (last-level cache) attacks. In their work, Demme et al. [16] demonstrate that small changes to the cache architecture have a considerable impact on side-channel vulnerability. Finally, [28] et al. present in their work a side-channel cache attack against Samsung TrustZone via the Android's Keymaster cryptographic functions.

Like cache attacks, DMA attacks are continuously under research. [49] et al. show that by dumping memory frequently enough using DMA transactions, write patterns can be examined. Some algorithms, such as the RSA Montgomery ladder [23], may leak secrets. DAGGER [46], a DMA-based keystroke logger, exfiltrates captured data to an external entity and cannot be detected by anti-virus software. [46] 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. [46] et al. also offer countermeasures to detect DMA attacks. [9] et al. integrate DMA attacks through FireWire into Metasploit [24]. Thus, an attacker could use Metasploit [24] for payload selection, session control, etc. and attack via DMA over Firewire.

TRESOR-HUNT [7] 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. Still, TRESOR-HUNT [7] injects malicious code to the kernel using a 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. [48] 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; by doing so, an attacker would have full access (READ/WRITE) to the target memory. Duflot et al. [17] introduce the vulnerability of remote code execution on a network adapter and how it could compromise the system-running kernel using DMA attack. BROADPWN [6] is a novel approach of privilege escalation from exploiting a bug in Broadcom WiFi chip into a DMA attack on the main processor of the device.

There are also hardware tools that perform attacks, such as PCI leech [41] that performs DMA attacks, Lan turtle [27] that performs a man-in-the-middle attack, and kon-boot [25] that bypasses Windows password protection [45].

The emerging cache, DMA and hardware attacks demonstrate that software bugs can impose security risks, and weak hardware implementation is becoming more common, specifically when new features rely on old security assumptions. For example, in the Raspberry PI case, CVE-2018-18068 is a privilege escalation vulnerability of non-authorised memory access via inter-processor debugging. This vulnerability is also demonstrated by [38] et al. who 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. Furthermore, 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. [38] 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. Defence against these vulnerabilities requires hardware and software solutions like the vulnerability we found. [38] et al. suggest that ARM should add restrictions in the interprocessor debugging model to enforce permission between host and target. OEMs should add software-based access control to go with the hardware permission model.

Matt Spisak et al. [44] describe another processor feature-based attack using ARM CoreSight debug features. [44] 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 modify the syscall's input and output data. This attack is possible due to a hardware implementation bug of a debug signal authorisation that enables debug features in the hardware. Cloaker [15] et al. leverage the 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). [34] et al. present the kernel CFI used to protect the kernel's stack and heap. However, 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 userspace portions could be 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 userspace memory while in kernel mode. Thus, attackers started 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 exploit indirect calls, i.e. function pointers. These attacks concentrate on a function's calling (forward edge) and returning (backwards edge). Thus, the primary 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 optimisation (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, OP-TEE must be compiled with Clang and then kFCI applied to use this feature. Unfortunately, none of these defences thwarts a DMA attack.

In the area of thwarting hypervisor CFI attacks, [51] et al. offer Hypersafe. Hypersafe protects the hypervisor from CFI hijack attacks through a memory lockdown and restricts pointer indexing, a layer of indirection that converts the control data into pointer indexes. These pointers indexes are limited such that the corresponding call/return targets strictly follow the hypervisor control flow graph, expanding protection to control-flow integrity.

This mitigation reduces the ease of performing a DMA attack on the hypervisor and, combined with IOMMUs, it is possible to impede the attack.

Table 4 List of vulnerable SoCs

Manufacturer	SoC	Device	Missing
TI	CC2538	Sensibo	TSASC
HISILICON	Hi3518EV200	Security	
		Cameras	TZASC
HISILICON	Hi3519V101	Security	
		Cameras	TZSAC

8 Conclusions

Using DMA attacks on the TrustZone yields 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 4 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 [20,30,42].

Funding Open Access funding provided by University of Jyväskylä (JYU).

Declarations

Conflict of interest The authors declare that they have no conflicts of interest.

Research involving human participants and/or animals Not applicable.

Informed consent Not applicable.

References

1. A technical report on tee and arm trustzone. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-technical-report-on-tee-and-arm-trustzone. Accessed: 2020-04-16
2. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. **13**(1), 4 (2009)
3. About the axi trustzone memory adapter. https://developer.arm.com/docs/dto0017/a/about-the-axi-trustzone-memory-adapter. Accessed: 2020-04-15
4. Arm trusted firmware. https://github.com/ARM-software/arm-trusted-firmware. Accessed: 2020-04-15
5. Arm trustzone. https://developer.arm.com/ip-products/security-ip/trustzone. Accessed: 2020-04-15
6. Artenstein, N.: Broadpwn: Remotely compromising android and ios via a bug in broadcom's wi-fi chipsets. Black Hat USA (2017)
7. Blass, E.-O., Robertson, W.: Tresor-hunt: attacking cpu-bound encryption. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 71–78 (2012)
8. Blazy, O., Yeun, C.Y.: Information Security Theory and Practice: 12th IFIP WG 11.2 International Conference, WISTP 2018, Brussels, Belgium, December 10–11, 2018, Revised Selected Papers. Lecture Notes in Computer Science. Springer International Publishing (2019)
9. Breuk, R., Spruyt, A.: Integrating dma attacks in metasploit. In: Sebug, D., volume 2 (2012). http://sebug.net/paper/Meeting-Documents/hitbsecconf2012ams
10. Cerdeira, D., Santos, N., Fonseca, P., Pinto, S.: Sok: understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 1416–1432. IEEE (2020)
11. Chen, Y., Zhang, Y., Wang, Z., Wei, T.: Downgrade attack on trustzone. arXiv preprint arXiv:1707.05082 (2017)
12. Cook, K.: Kernel address space layout randomization. Linux Security Summit (2013)
13. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security Symposium, vol. 98, pp. 63–78. San Antonio, TX (1998)
14. Davi, L., Koeberl, P., Sadeghi, A.-R.: 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), pages 1–6. IEEE (2014)
15. David, F.M., Chan, E.M., Carlyle, J.C., Campbell, R.H.: Cloaker: hardware supported rootkit concealment. In: 2008 IEEE Symposium on Security and Privacy (sp 2008), pp. 296–310. IEEE (2008)
16. Demme, J., Martin, R., Waksman, A., Sethumadhavan, S.: Side-channel vulnerability factor: a metric for measuring information leakage. In 2012 39th Annual International Symposium on Computer Architecture (ISCA), pp. 106–117. IEEE (2012)
17. Duflot, L., Perez, Y.-A., Valadon, G., Levillain, O.: Can you still trust your network card. CanSecWest/core10, pp. 24–26 (2010)
18. Göttel, C., Felber, P., Schiavoni, V.: 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, pp. 170–178. Springer (2019)
19. Green, M., Rodrigues-Lima, L., Zankl, A., Irazoqui, G., Heyszl, J., Eisenbarth, T.: Autolock: Why cache attacks on {ARM} are harder than you think. In: 26th {USENIX} Security Symposium ({USENIX} Security 17), pp. 1075–1091 (2017)
20. Guan, L., Liu, P., Xing, X., Ge, X., Zhang, S., Yu, M., Jaeger, T.: 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 (2017)
21. Guilbon, J.: Attacking the arm's trustzone. https://blog.quarkslab.com/attacking-the-arms-trustzone.html. Accessed: 2020-04-16
22. https://globalplatform.org/specs-library/tee-internal-core-api-specification . Accessed 2021-8-05
23. Joye, M., Yen, S.-M.: The montgomery powering ladder. In: International Workshop on Cryptographic Hardware and Embedded Systems, pp. 291–302. Springer (2002)
24. Kennedy, D., O'gorman, J., Kearns, D., Aharoni, M.: Metasploit: The Penetration Tester's Guide. No Starch Press (2011)
25. kon-boot. https://kon-boot.com/. Accessed: 2021-10-21
26. Kupfer, G., Tsafrir, D., Amit, N.: IOMMU-resistant DMA attacks. PhD thesis, Computer Science Department, Technion (2018)
27. lan-turtle. https://hak5.org/products/lan-turtle/. Accessed: 2021-10-21
28. Lapid, B., Wool, A.: Cache-attacks on the arm trustzone implementations of aes-256 and aes-256-gcm via gpu-based analysis. In: International Conference on Selected Areas in Cryptography, pp. 235–256. Springer (2018)
29. Leonard, J.: Why trustzone matters for iot. https://blog.nordicsemi.com/getconnected/why-trustzone-matters-in-iot. Accessed: 2020-04-15
30. Lesjak, C., Hein, D., Winter, J.: Hardware-security technologies for industrial iot: Trustzone and security controller. In: IECON 2015-41st Annual Conference of the IEEE Industrial Electronics Society, pp. 002589–002595. IEEE (2015)
31. Libre and portable reverse engineering framework. https://rada.re/n/. Accessed: 2020-04-17
32. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: Armageddon: Cache attacks on mobile devices. In 25th {USENIX} Security Symposium ({USENIX} Security 16), pp. 549–564 (2016)
33. Makkaveev, S.: The road to qualcomm trustzone apps fuzzing. https://research.checkpoint.com/2019/the-road-to-qualcomm-trustzone-apps-fuzzing/. Accessed: 2020-04-16
34. Moreira, J., Rigo, S., Polychronakis, M., Kemerlis, V.P.: Drop the rop fine-grained control-flow integrity for the linux kernel. Black Hat Asia (2017)

35. Nehal, A., Ahlawat, P.: Securing iot applications with op-tee from hardware level os. In: 2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA), pp. 1441–1444 (2019)
36. Ngabonziza, B., Martin, D., Bailey, A., Cho, H., Martin, S.: Trustzone explained: architectural features and use cases. In: 2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC), pp. 445–451 (2016)
37. Nikolai, Hampton: (Computerworld. The working dead: The security risks of outdated linux kernels. https://www2.computerworld.com.au/article/615338/working-dead-security-risk-dated-linux-kernels/. Accessed: 2020-04-16
38. Ning, Z., Zhang, F.: Understanding the security of arm debugging features. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 602–619. IEEE (2019)
39. Open portable trusted execution environment. http://www.op-tee.org/. Accessed: 2020-09-30
40. Papp, D., Ma, Z., Buttyan, L.: Embedded systems security: threats, vulnerabilities, and attack taxonomy. In: 2015 13th Annual Conference on Privacy, Security and Trust (PST), pp. 145–152. IEEE (2015)
41. pcileech. https://github.com/ufrisk/pcileech. Accessed: 2021-10-21
42. Pinto, S., Gomes, T., Pereira, J., Cabral, J., Tavares, A.: Iioteed: an enhanced, trusted execution environment for industrial iot edge devices. IEEE Internet Comput. **21**(1), 40–47 (2017)
43. Shen, D.: Exploiting trustzone on android. Black Hat USA (2015)
44. Spisak, M.: Hardware-assisted rootkits: Abusing performance counters on the {ARM} and x86 architectures. In: 10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16) (2016)
45. Steal windows password. https://arstechnica.com/gadgets/2021/08/how-to-go-from-stolen-pc-to-network-intrusion-in-30-minutes
46. Stewin, P., Bystrov, I.: Understanding dma malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 21–41. Springer (2012)
47. The latest security information on intel$\mathring{R}$ products. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00266.html. Accessed: 2020-04-15
48. Trikalinou, A., Lake, D.: Taking dma attacks to the next level. BlackHat USA (2017)
49. van Dijk, M., Haider, S.K., Jin, C., Nguyen, P.H.: Advanced power side channel cache side channel attacks dma attacks (2017)
50. Wallen, J.: Most iot devices are an attack waiting to happen, unless manufacturers update their kernels. https://www.techrepublic.com/article/most-iot-devices-are-an-attack-waiting-to-happen-unless-manufacturers-update-their-kernels/. Accessed: 2020-04-16
51. Wang, Z., Jiang, X.: Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In: 2010 IEEE Symposium on Security and Privacy, pp. 380–395 (2010)
52. Weinmann, R.-P.: Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks. In: WOOT, pp. 12–21 (2012)
53. Yarom, Y., Falkner, K.: Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In: 23rd {USENIX} Security Symposium ({USENIX} Security 14), pp. 719–732 (2014)
54. Yehuda, R.B., Zaidenberg, N.J.: Protection against reverse engineering in arm. Int. J. Inf. Secur. **19**(1), 39–51 (2020)
55. Zhang, M., Sekar, R.: Control flow integrity for {COTS} binaries. In: Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13), pp. 337–352 (2013)
56. Zhang, M., Zhang, Q., Zhao, S., Shi, Z., Guan, Y.: Softme: a software-based memory protection approach for tee system to resist physical attacks. Security and Communication Networks, 2019 (2019)
57. Zhao, S., Zhang, Q., Qin, Y., Feng, W., Feng, D.: 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), pp. 105–120 (2019)