Breaking Cross-world isolation on ARM TrustZone through EM Faults, Coredumps, and UUID Confusion

Nimish Mishra* Anirban Chakraborty *
Debdeep Mukhopadhyay*

August 21, 2022

Abstract

In this work, we break the promised normal world to secure world isolation on ARM TrustZone. We attack the OP-TEE implementation of TrustZone in three phases. Firstly, we load a malicious, self-signed Trusted Application (TA) on the secure world side. To do this, we propose a register-cleaning fault attack through electromagnetic (EM) pulse injections, following a completely non-invasive EM triggering mechanism. In the second phase, we take advantage of an empirical observation- OP-TEE prefers opening sessions with non-persistent TA than it does with persistent TA, if both share the same Universally Unique Identifier (UUID). From the malicious TA installed in the first phase, the adversary forces the UUID to be that of another TA already installed in the secure world, thereby effectively mounting a Man-in-the-middle attack. Lastly, we use carefully timed EM injections to force `SIGSEGV` signals to be sent to otherwise correct executions, forcing new execution paths causing coredumps which leak encryption and signing keys (assuming encryption and source-authentication in place for normal world to secure world communication). We demonstrate the entire end-to-end attack to gain unauthorized access to a Machine-Learning-as-a-Service (MLaaS) server running inside the ARM TrustZone. Finally, we propose entirely software based countermeasures to prevent all three attack vectors.

1 Introduction

The Internet-of-Things (IoTs) has become an essential part of modern industry, catering to millions of users with critical services. Naturally, the question of security in the IoT hemisphere has risen in the past years. With the increasing ability of attackers to compromise systems, even the most privileged

*Indian Institute of Technology, Kharagpur

of all softwares (i.e. the kernel) has come to be viewed with doubt. To counter this, the security community has been gradually shifting its root-of-trust to hardware backed mechanisms. Examples of such hardware backed security solutions from major chip manufacturers include the ARM TrustZone, Intel's SGX (Software Guard Extensions) and TDX (Trusted Domain Extensions), AMD's PSP (Platform Security Processor). While specifics of these solutions vary, the core idea is to partition resources on a System-on-Chip (SoC) into a trusted and an untrusted execution environment and shifting critical executions to the trusted environment. And doing this isolation with the help of the hardware adds another layer of security: if an adversary compromises the kernel on the untrusted part, there is still a line of defence before the security-critical data is compromised. In this paper, we focus exclusively on ARM TrustZone [1]. ARM Trustzone has been an integral part of ARM chipsets since ARMv6 [1], including Cortex-A (for mobile devices) and Cortex-M (for IoT devices) family of processors. The TrustZone provides an execution context for security-critical applications such as user authentication, mobile payment, etc. It essentially partitions the System-on-Chip hardware and software into two virtual execution environments: secure world or **Trusted Execution Environment (TEE)** and normal world or **Rich Execution Environment (REE)**. Applications running in REE are called *client applications* (CAs), while the ones running in TEE are called *trusted applications* (TAs). The REE supports a complex software stack and thus can be prone to severe software bugs leading to leakage sources. In this context, TEE provides the necessary isolation guarantee such that the integrity and confidentiality of sensitive data are not compromised. Shared hardware resources such as storage, peripherals, etc. are made private to each world through hardware-powered mechanisms [2, 3].

1.1 Protections for TAs in ARM TrustZone

Hardware backed isolation such as ARM TrustZone is also a natural solution to the problem of protecting security critical computation in the context of IoT edge nodes running on SoCs [4, 5]. We have identified several features- both inherent to ARM TrustZone as well as belonging to third-party extensions- which provide sound security to a TA's operations. Briefly, such features come in two dimensions along which security is desirable: ① securing the communication channel(s) between a TA and a CA, and ② ensuring the resource partitioned to a TA actually remains inaccessible to other TAs/CAs.

1.1.1 Security in dimension ①

Communication flow in the setting we are exploring is: end-user $\rightleftarrows$ CA(s) $\rightleftarrows$ TA(s). There are a few problems related to this communication channel's privacy, which we list next. We also list solutions to these problems. Note that since none of the proposed solutions are inherent to ARM TrustZone, we rely on third-party security extensions when required.

[1] The current ARM version used in a majority of processors is ARMv8.

Achieving source authentication. We assume presence of a *Gatekeeper* TA, in line with the proposal in [6]. Our experimental platform is OP-TEE, which we introduce later in Sec. 2. For now, it suffices to note that the reason of choosing this design is because it was put forth by the very team developing OP-TEE. Concretely, a *Gatekeeper* TA stands at the *gate* between the REE and TEE on ARM TrustZone (refer Sec 5.1 for details on *Gatekeeper* architecture). Only if a CA authenticates itself to this *Gatekeeper* TA, it is allowed to communicate with a TA of its choosing.

Achieving granular access control. In a production environment, a TA should not be accessible by CA(s) which it does not, to some degree, *trust*. The *Gatekeeper* TA [6] does not have this ability of choosing which CA can communicate with it. Therefore, we rely on another third-party security extension *SeCReT* [7] (or *SeCReT*'s production-system friendly sibling [8]). Among many features, one ability of *SeCReT* is that a TA can define a static access control list (ACL), listing out all CAs in a system which it wants to allow to communicate with itself. It is not a far-fetched assumption that, for maximizing security, a TA should keep this ACL as small as possible and restricted to CA(s) which it, to some degree, trusts.

Achieving communication privacy. A straightforward solution to protect communication channels is symmetric encryption. Moreover, to reliably achieve source authentication, signing the encrypted messages is also desirable. Although the *Gatekeeper* TA is able to achieve the second goal, there is still one question: is *the signing key safe with the CA*? As CA(s) do not belong to a TA's root-of-trust, it is possible for a compromised REE kernel to simply extract the encryption/signing key from the CA(s) and compromise the communication channel. To this end, *SeCReT* [7] (or alternatively its sibling [8]), proposes a solution. Instead of letting a CA handle its keys, *SeCReT* handles them for the CA. *SeCReT* monitors accesses to the CA's memory page where the keys are stored, and blocks illegitimate processes from reading that memory page. For every context switch, *SeCReT* performs register level verification (to prevent control-flow manipulations) and memory flush operations to prevent any residue of the keys from being leaked to an adversary. Furthermore, *SeCReT* itself resides as a kernel module in both worlds. We assume that attacks on *SeCReT* itself are outside the scope of this work.

1.1.2 Security in dimension ②

While the previous dimension secured the communication channel, here we discuss secure partitioning of resources- an easily achievable goal through virtualization. However, ARM TrustZone provides no inherent support for virtualization. Hence, we consider third-party extensions. Hua *et. al.* [9] assumes an attacker targeting the virtualization hypervisor. They develop defences against a compromised hypervisor breaking TAs' isolation (referred to as *guest TEE*s) through manipulating the boot sequence, CPU states, and secure memory/peripherals. Liu *et. al.* [10] is another such virtualization defence targeting secure peripherals.

1.2 Goals

In this paper, we ask the following question: *given the defences in these two dimensions, is it possible to break secure isolation on an implementation of ARM TrustZone*? The entire communication is encrypted, and all defences we mentioned so far are in place. Our goal is to find a way for an adversary to access a TA in an execution context where it shouldn't. Hereafter, we call the target TA as *victim TA*. To put into perspective how such an attempt is impossible so far, we note the following points:

1. The adversary cannot deploy a CA which directly communicates with the victim TA, since *SeCReT* maintains an access control list of CAs allowed to access the victim TA. Since this list's owner is the victim TA itself, we assume it does not intentionally expose its own service to anyone other than the *innocent CA*.

2. Because the TAs are signed by device manufactures and the signing keys are off-device, an adversary cannot load a modified TA in the system.

3. The adversary cannot access resources belonging to the victim TA because of the advanced virtualization defences in place.

4. The adversary cannot spy on the victim TA's communication since it is encrypted. Moreover, no man-in-the-middle attack can be mounted since source authentication occurs through the *Gatekeeper* TA.

5. The adversary cannot retrieve the encryption/signing keys from the *innocent CA* since key management is done by *SeCReT*. Since attacks on *SeCReT* itself are outside the scope of this work, we assume complete integrity of *SeCReT*'s functioning.

To motivate this goal in more concrete terms, we assume a TA running a MLaaS (machine learning as a service) server and show an end-to-end attack in Sec. 7.4. In the context of our work, we assume that this MLaaS TA has a corresponding *innocent CA* which is responsible for receiving requests from end-users and get the MLaaS TA to process them. Under these assumptions, this paper chalks out an end-to-end attack wherein an adversary can hijack the incoming encrypted and signed communication to the MLaaS TA, decrypt it, change it, re-encrypt it, re-sign it, and fool the MLaaS TA into believing that the communication was untampered. This leads to an adversary using the services of a MLaaS TA without the latter having any knowledge of the compromise.

Generality of the MLaaS example. Although we speak of a MLaaS TA as our target, our attack vector is in no way specific to it. Our attacks are applicable on any TA; nevertheless we choose MLaaS as a demonstration example since it is a pay-per-use service and typically has privacy concerns accompanying it [11–13]. Moreover, quite some recent interest has risen in adapting MLaaS in an IoT environment [14]. MLaaS is a perfect example because of its critical security-centric nature, the need to prevent unauthorized access to it, and the economic loss accumulated should MLaaS start servicing unauthorized requests.

1.3 Comparison with prior works

In this subsection, we summarize prior works and motivate our choice of the platform, the hardware, and the specific methodology we undertake in this work.

1.3.1 Related works

Fault injection attacks refer to an umbrella of techniques designed for one purpose: to influence the operational conditions of the victim device to introduce unexpected execution paths in a program. Due to this *faulty* intermediate execution state, an adversary is able to do something creative: steal cryptographic keys, bypass verification checks, and so on. Fault injection attacks have evolved into several different subcategories: voltage glitches (manipulating the power supply of the victim device), clock glitches(manipulating the clock input to the victim device), temperature manipulations, and shooting electromagnetic/optical pulses onto the system.

The body of research into fault attacks and their consequences is large. To keep this section focused, we only consider works pertaining to a threat model wherein the adversary has physical access to a device. There have been works [15–17] which assume a remote adversary manipulating the dynamic voltage-frequency scaling (DVFS) interfaces of the victim device to mount fault injections. However, as explained in Sec. 1.3.2, our target does not expose DVFS interfaces thereby rendering these attack vectors useless. Similarly, as discussed in Sec. 2, since secure-world receives no interrupts from the normal world, attacks like [18] do not work either. Similarly, we do not consider works using laser fault injections [19, 20] because of ① higher acquisition and assembly cost for the setup, and ② more time is required to find faults than electromagnetic fault injections.

Even assuming physical access to the victim device, the large body of works on the topic is too large. Therefore, we only consider works with some degree of implications for Trusted Execution Environments (ARM TrustZone, Intel Software Guard Extensions, AMD's Secure Processor etc). Flynn *et. al.* [21] deploy an electromagnetic fault injection technique to fault an integer pointer and read more data than they are allowed, thereby leaking secrets. Nashimoto *et. al.* [22] used a clock glitch to bypass RISC-V's physical memory protection, with direct implications for trusted execution environments. Chen *et. al.* [23] introduce a voltage glitching through faults on the bus between the CPU and the voltage regulator on the motherboard. The attack targets Intel's Software Guard Extensions (SGX) enclaves: Intel's counterpart of ARM TrustZone. Moreover, Buhren *et. al.* [24] use voltage glitches to load malicious firmware that decrypts virtualized memory and fake attestations.

1.3.2 Motivations for attack methodology

In this section, we pinpoint gaps in the attack vectors proposed by prior literature and how none of them are practical on the platform (both hardware

and software) we are targeting. As we mention in Sec. 2 and Sec. 7, our choice of ARM TrustZone implementation is OP-TEE and our hardware platform is Raspberry Pi 3 model B (because of their increasing popularity as the SoCs of the IoT hemisphere [4, 5]).

We first note that remote faulting is not possible in our work's context. For remote fault injections, an adversary needs an exposed dynamic voltage-frequency scaling (DVFS) interface. There are two ways to do this on our target: ① through Linux's scaling governors [25], and ② through `config.txt` that a Raspberry Pi initializes from during boot time [26]. Since ① no longer allows software-based overclocking post similar attacks [16] and ② is not editable from OP-TEE's stripped down Linux kernel in the normal world, remote fault injections like [15, 16] are not possible in our target.

We do, however, consider adversaries with physical access to the device. We borrow two critical objectives for such an adversary from [21]:

1. Attack without physical modifications to the device, preventing future detection of the attack. Therefore, decapsulation is not within scope of our work.

2. Attack in a *reasonable* time-frame of 0.5 – 2 hour window [21]. The *reasonable* time-frame is considered as the duration in which the attack can be mounted, before the device owner notices the device being missing and revokes credentials.

We note, however, that none of the aforementioned works can be directly translated to our setting. For works like [21, 27–29], the problem is their consideration of the victim device being far simpler than a System-on-Chip (SoC) like Raspberry Pi, housing an application grade processor running a production level operating system (which requires radically different fault techniques [30]). Bukasa *et. al.* [28] does consider a similar verification bypass as we do in Sec. 4; however, we believe the high success rate is due to faulting one bit value on a non SoC device. Similarly, works like [22, 31–33] do not apply to our setting because of Raspberry Pi 3's lack of an external clock/voltage interface.

In addition to running on low-end devices like FGPAs and deploying external voltage/clock glitches, works like [22] assume the ability of an attacker to introduce code-based changes allows a *trigger* signalling the start of an execution of interest where the fault must be injected. Although an acceptable fault model in academic settings, we don't agree with its practical implications. From an adversarial point of view, the victim device is running an already compiled code. To be able to introduce a code-based *trigger*, an adversary has to decompile the running binaries (this step is easier if the victim codebase is open-source), introduce a code-based trigger, recompile the codebase building all dynamic libraries (if any), and rerun everything. After this, the fault injection starts. However, the complexity of this entire attack makes its practicality questionable. In this work, we therefore explore less invasive *triggering* mechanisms: triggers that indicate start of an execution of interest without needing code-based triggers. We note that while [23] also requires software based triggers for glitching, the

authors mention that this triggering from the target system is not a strict requirement, rather a matter of convenience.

Although explained in detail later in Sec. 4.2, we briefly mention our departure from established approaches. We also use a *trigger* to begin fault injection. But unlike previous works where changes to the target system were required to generate this trigger, we rather rely on the open-source nature of OP-TEE and techniques of side-channel power trace acquisitions to generate triggers not requiring any modifications to victim's software. This approach is one instantiation of a more generic technique: using the *behaviour* of the target system itself as a trigger. For example, Buhren *et. al.* [24] uses the size of bus traffic as a trigger for voltage glitches. Van *et. al.* [34] uses pattern matching on normal device power signal to trigger fault injections.

Portability of the attack. Note that although we choose OP-TEE as our attack target, our attack is still generic. The attack succeeds not due to any shortcomings of OP-TEE, but due to ① lack of metallic shields on Raspberry Pi 3 processor allowing fault injections, and ② loopholes in GlobalPlatform (GP) API specification for Trusted Execution Environments leading to attacks discussed in Sec 5.2. Therefore, any other implementation of ARM TrustZone based off GP API specifications and running on an unshielded processor is vulnerable to similar attack vectors.

Paper Organization. The paper is organized as follows. In Sec. 2, we introduce OP-TEE: our target ARM TrustZone implementation. Then we introduce a three stepped end-to-end attack in Sec. 3. We cover the first phase of the attack in Sec. 4, the second phase in Sec. 5, and finally the last phase in Sec. 6. We provide a detailed implementation discussion in Sec. 7, wherein we also provide an end-to-end attack on a MLaaS TA. Finally, we conclude the paper.

2 OP-TEE implementation

We focus on OP-TEE's implementation of ARM TrustZone [35]. It is one of the most popular open-source implementations of ARM TrustZone with major industrial players involved in its design and development (otherwise known as the Trusted Firmware project) and moderate to large scale production systems like Apertis [36] and iWave systems [37] mentioning their use-cases with OP-TEE integrations. It is a TrustZone implementation for Cortex-A cores with design goals - isolation, small memory footprint, and portability. All architectural design decisions conform to GlobalPlatform's (GP) specifications for embedded applications on secure hardware. The REE is modelled by the GlobalPlatform Client API specifications [38], while the TEE is governed by the GP Internal Core API specification [39].

Intuition for the isolation. ARMv8 adopts a similar convention of privilege rings as Linux. As referred in Fig. 1, the REE and TEE are divided into two layers for granular control: Exception layers (EL) 0 and 1, which provide granular control over the actions of the REE and TEE. EL0 usually houses the

userspace applications while EL1 houses the kernel. On the REE side, Linux is used as the operating system; while on the TEE side, an OP-TEE OS is used as the operating system. Because of no relevance to the paper, we omit the shared cache region between TEE EL0 and REE EL0 (which all cache-based attacks on ARM TrustZone use). Exception Layer 2 houses the secure monitor call (SMC) handler allowing cross-world communication through software interrupts triggering context switches. OP-TEE also has a collection of interrupts named *supervisor calls* (SVC) that allow the transfer of control from EL0 to EL1 in the same world. Briefly, apart from the shared memory and these interrupts, *there is no contact between the REE and the TEE*. This design implies that even if the REE has been compromised, TEE is still secure.

Intuition for the world switching. OP-TEE ensures the SMC interrupt is non-maskable and controlled by EL2. On an SMC interrupt, EL2 changes the ownership of the processor unless the current world is done executing or EL2 receives a non-maskable interrupt from the idle world. Within EL1 in both worlds, normal scheduling policies apply. Additionally, the ARM Generic Interrupt Controller (GIC) can also signal either world. Without loss of generality, let us assume the GIC signals REE. If REE is in possession of the processor, it branches to the known interrupt service routine (ISR). If TEE is running, it first relinquishes control of the processor to the REE through an SMC interrupt, and then the REE jumps to the known ISR to handle GIC's signal.

3 Three pronged attack model

In Sec. 1.2, we introduced a goal where an adversary aims to use services of a private victim TA even in the presence of several defence mechanisms (c.f. Sec. 1.1). In this work, we demonstrate a three-pronged attack by undermining the isolation guarantees provided by both software (OP-TEE + third-party extensions) and hardware (ARM TrustZone). From the two dimensions of security features discussion in Sec 1.1, we need not consider dimension ② since it solely focuses on isolation through virtualization, preventing unauthorized manipulation of CPU state, secure memory, and secure peripherals. Our attack succeeds without taking these into consideration. We launch the attack in three stages:

1. **Installing a malicious TA**. For a specific device, all TAs are signed and loaded at the time of device manufacturing. The private key for signing of TAs is never loaded onto the device. Hence, bypassing the signature verification is the only possible way to load a self-signed TA onto the victim system. We utilize specifically targeted electromagnetic fault injections to load self-signed TAs onto the system.

2. **Replacing the *Gatekeeper* TA through UUID confusion**. We take advantage of the way OP-TEE decides which TA receives communication from the normal world. When a CA requests communication from the secure world, it initializes a session with the secure world and passes the *Universally Unique Identifier* or UUID of the TA with whom it wishes to communicate. The secure world loads the TA with this UUID. However,

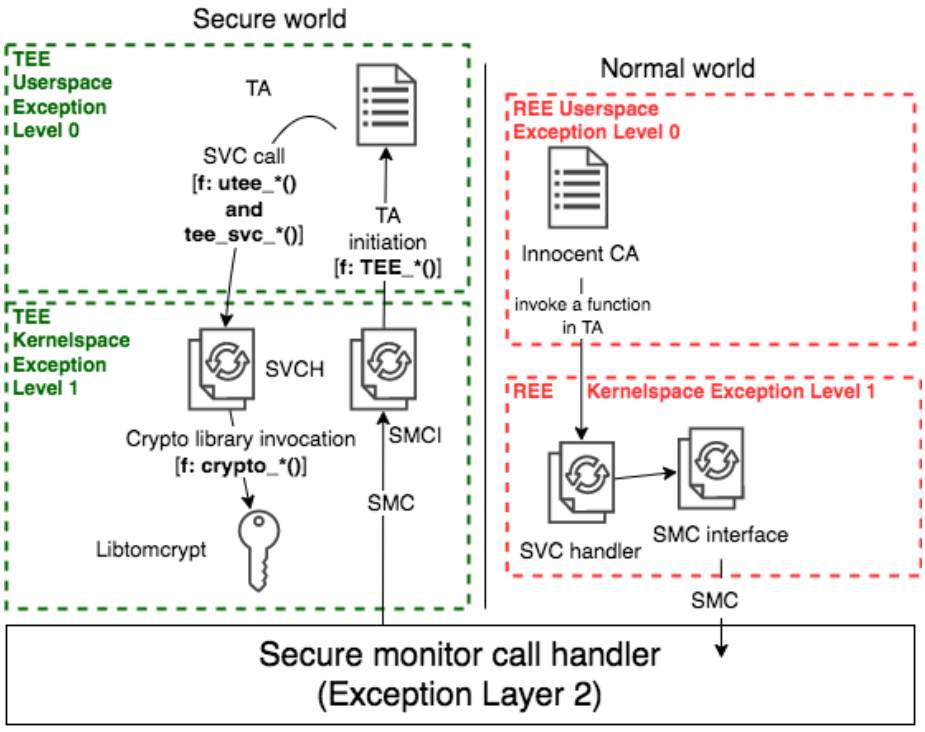

Figure 1: OP-TEE Architecture. `SVCH` and `SMCI` denote Supervisor Call (SVC) Handler and Secure Monitor Call (SMC) Interface respectively.

through our empirical observations, we realized that there is no restriction on two different TAs sharing the same UUID. Therefore, an adversary deploys a TA whose identifier (called UUID) matches that of the *Gatekeeper* TA already present in the system. Under certain circumstances (c.f. Sec. 5.2), control flow meant for the *Gatekeeper TA* reaches a malicious TA belonging to the adversary. Thereby, the adversary is basically able to hijack communication between the two worlds.

3. **Breaking encryption/authentication through SIGSEGV**: Even after achieving a Man-in-the-middle attack through UUID confusion, the adversary does not achieve its actual goal, since we assumed the communication is encrypted. Moreover, due to presence of *SeCReT* (c.f. Sec. 1.1), the encryption/signing keys are safely managed and are not trivially leaked through known side-channel attacks. This is because the *innocent CA* does not have access to these keys except for a short duration, after which all memory belonging to the keys is flushed by *SeCReT*. To counter this, we introduce faults that send carefully timed SIGSEGV signals. In our experiments, we have been able to leak the entire signing/encryption key through a single fault.

With these three attack vectors in place, an adversary is able to *sit* between

the victim TA and the *innocent CA* (c.f. Fig 3). With the help of leaked encryption/signing keys, the adversary decrypts messages meant for victim TA, edits them, re-encrypts them, resigns them, and then sends them to victim TA. The signature verification done by victim TA will succeed and the victim services the requests. Thereby, an adversary is able to gain access to an otherwise private victim TA.

4 Attack 1: Installing malicious TAs

OP-TEE follows an offline signing and online verification process. The underlying assumption is that the root-of-trust lies with the manufacturer itself. The manufacturer signs each Trusted Application with its private key [40], stitches the signature with the symbol-stripped TA binary, and loads the binary onto the system along with the public key for signature verification. Listing 1 depicts this verification process. `TEE_ERROR_SECURITY` is a special error code returned by OP-TEE in case signature verification fails.

```
#define TEE_SUCCESS 0x00000000
#define TEE_ERROR_SECURITY 0xFFFF000F

TEE_Result verify_signature(char* ta_binary, uint8_t* signature){
    if(/*signature is valid*/)
        return TEE_SUCCESS;
    return TEE_ERROR_SECURITY;
}

// load a TA referenced by a CA, through its UUID
void load_TA(...){
    // some code here
    TEE_Result res = verify_signature(...)
    if(res != TEE_SUCCESS)
        // abort execution
    // some more code here
}
```

Listing 1: OP-TEE TA signature verification process

4.1 Bypassing verification: register-cleaning attack

Since the signature keys used to sign the TAs are not present on-device, forging of signatures through signing key leakage is not possible. Therefore, the only possible way to load a self-signed TA is to attempt a bypass of this verification step.

Observation. From listing 1 (and from reading OP-TEE's source code), we observed that while several error states (like security errors, out-of-memory errors etc) were given a 32-bit value, `TEE_SUCCESS` was given a value `0x0`. This is not uncommon: almost all Linux based function calls conventionally use a 0 to denote success, and other integers denote several erroneous operations.

We exploit this convention. Listing 2 gives the aarch64 disassembly of Listing 1. The point of interest is one `mov` instruction which loads the return value of `verify_signature` into `res`, which in turn resides in a register. We attack this operation by injecting a powerful electromagnetic (EM) pulse train at the appropriate point in the execution, leading to pollution of values of `res`. If the

EM injection is precise enough temporally and spatially, the value of `res` can be completely cleaned (i.e. `res` could be forced as `0x0`).

```
.text.verify_signature:
    // some code here
    mov w0, <return_code>
    ret

.text.load_TA:
    // some code here
    // setting up parameters
    bl <verify signature>
    cbnz w0, <error_out>
    // some more code here
```

Listing 2: Sample disassembly of OP-TEE TA signature verification process. A `mov` instruction moves the actual return code into the register `w0`, which then goes through a `cbnz` or a "compare and branch on non-zero" instruction to decide whether to error out or not. Through a powerful enough EM injection on these instructions, `w0` can be cleared out to `0x0`, thereby bypassing the otherwise jump that `cbnz` would have taken.

4.2 Non-invasive triggering mechanism

In Sec. 4.1, we implicitly assumed that we have temporal precision over the *appropriate point in the execution* wherein the value of `res` is being updated. This is not entirely true. As we discussed in detail in Sec 1.3.2, several works assume a code-based change to *trigger* the fault injections. For usual code-based triggers, just before the security critical operation of interest begins, a signal is generated to an attack controlled device, which in turn starts injecting faults. Concretely, in the case of Raspberry Pi 3 running Raspbian OS, one probable way to create such a code-based trigger is given in Listing 3. Just before the operation of interest starts, the attacker induced code sends a square pulse over GPIO pin 7. The adversary would capture this pulse through an external signal generator, which will then start generating a train of pulses. This train of pulses shall be used by fault injection probes to inject faults temporally localised to the security critical operation.

```
#define TARGET_GPIO_PIN 7
void vicim_code(...){
    digitalWrite(TARGET_GPIO_PIN, HIGH);
    // some delay to let the output stabilize
    digitalWrite(TARGET_GPIO_PIN, LOW);
    // some security-critical operation
}
```

Listing 3: Code-based trigger mechanism for Raspbian based attack targets

However, there are two problems with this approach:

- **Generic problem.** In the context of our work, we consider code-based triggers as *invasive* triggering mechanisms. Reason being that from an adversarial point of view, an adversary has to *replace* the process binaries running on the target system with its own instrumented binaries. This is *invasion* in every sense of the word in most practical settings.
- **OP-TEE specific problem.** OP-TEE's normal world is a stripped down Linux kernel exposing a BusyBox interface to the adversary. From our

empirical observations, we found that OP-TEE's normal world Linux does not expose a GPIO interface which an adversary could use.

Putting both these problems together implies that a lesser non-invasive triggering mechanism is required to temporally localise faults. Therefore, we use analysis of power traces as a triggering mechanism (c.f. Fig. 8). Power trace acquisition allows an adversary to monitor the power consumption by the process throughout the course of execution. Historically, it has been established that some operations take more power than others, leading to a bias in acquired power traces. Through this bias, adversaries have been able to break cryptographic primitives [41, 42].

Observation. Multiplications are computation heavy operations, requiring higher power consumption than other operations. And multiplications are heavily used in the signature verification listed in Listing 3.

To motivate the use of multiplications as triggering mechanisms, we note ideas in literature on two fronts: ① the power consumption during multiplications *vs* other generic operations [43], and ② the heavy use of multiplications in OP-TEE's RSA signature verification process [44, 45]. From the combination of ① and ②, along with ③ the fact that a multiplication-heavy signature verification happens just before our point of interest (c.f. Listing 1) and ④ the fact that OP-TEE codebase is open-source, we have a non-invasive triggering mechanism to denote when our signal generators should begin sending pulses to inject because we know in advance the approximate time when signature verification starts. Experimental setup is discussed in Sec. 7.1.

5 Attack 2: UUID confusion

In Sec. 4, we dealt with the problem of loading a self-signed TA into the secure world. However, just loading a TA into the system does not do much damage. This is because of several defences in place preventing a TA launching an attack on another TA (c.f. Sec 1.1). Through the attack summarized in Sec 4, we just have a rogue self-signed TA into the system. In this section, we deal with the question on how to make use of the loaded self-signed TA to hijack communication meant for other TAs. We describe the second level of exploit that we introduced in Sec. 3 to replace the *Gatekeeper* TA.

5.1 Gatekeeper architecture

We introduced the *Gatekeeper* TA concept in Sec 1.1.1 and motivated its use because of the OP-TEE team themselves designing it [6]. We now introduce the *Gatekeeper*'s goals [6]:

- Password authentication

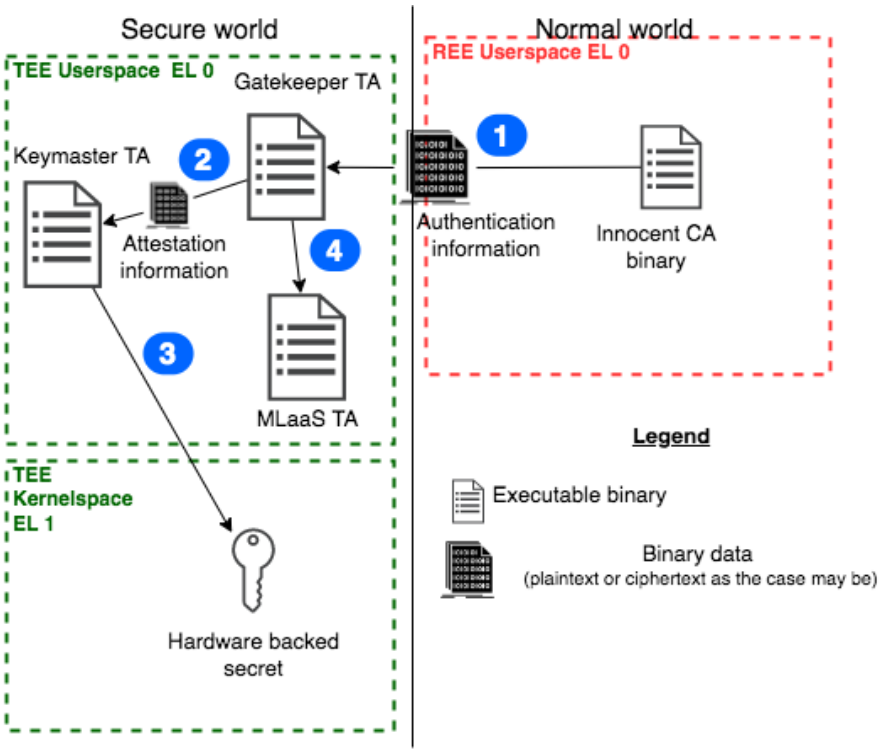

Figure 2: Schematic of the *Gatekeeper* TA architecture. Note that the communication flow has been depicted, for brevity, from *innocent CA* to the victim TA (a MLaaS TA in our example in Sec. 7.4). The reverse directional communication is straightforward to infer.

- Creation of attestations to be sent to the Keymaster (c.f. Fig 2). The Keymaster TA is responsible for binding all incoming connections to a hardware root-of-trust.

- Leveraging hardware-backed secret key

We give a high level overview of the *Gatekeeper TA* architecture in Fig 2. The *innocent CA* sends some authentication information, which is verified and attested by the *Gatekeeper TA*, which in turn is sent to the Keymaster TA which binds the attestation to a hardware-backed root-of-trust. Once this is done, the *Gatekeeper TA* allows the *innocent CA* to initiate communication with the victim TA. However, we have a crucial observation that will be central to replacing the *Gatekeeper TA*

Observation. The *Gatekeeper TA* resides in TEE Userspace Exception Level 0, which is the same privilege level as of the self-signed TA loaded by the adversary (c.f. Sec 4).

5.2 Replacing the Gatekeeper TA

In this section, we present a novel UUID confusion exploit and motivate reasons for its existence. The UUID confusion exploit allows a TA to effectively spy on the communication intended for another TA. When hardware-based isolation between normal world and secure world is developed, the designers need a way to allow a CA to communicate with TAs. OP-TEE does this through unique TA identifiers called *Universally Unique Identifier* (UUID). UUIDs are 128-bit numbers that uniquely identify TAs and are very hard to predict in advance. Moreover, OP-TEE plays no role in UUID assignment to TAs (UUID generation is responsibility of the TA developer) as well as TA UUIDs are considered **not sensitive** information; something that can be publicly available to CAs.

On an orthogonal note, OP-TEE also has the concept of **persistent** and **non-persistent** TAs. A persistent TA maintains its state even when no CA has an active session going on. Non-persistent TAs spawn only when a CA initiates a session, and close down when the CA closes the session. Most TAs acting as servers choose to run as persistent TAs: always waiting for new incoming connections to work upon. We note that the *Gatekeeper* TA, since it serves authentication services to end-users, will also be running as a persistent TA.

In our investigation, we made empirical observations about the interplay of persistence and UUID. OP-TEE makes two design decisions in its implementation: ① OP-TEE does not prevent two TAs from having the same UUID, and ② OP-TEE prefers opening sessions with non-persistent TAs than it does with persistent TAs. To elaborate on ②, if we have two TAs: persistent **TA-x** and a non-persistent **TA-y** with the same UUID **z** and a CA initiates communication with UUID **z**, OP-TEE will prefer opening a session with **TA-y**.

We show this exploit schematically in Fig. 3. The *innocent CA* wishes to get the services of a victim TA. It initiates a session with the secure world. The *Gatekeeper* TA receives the authentication data and the encrypted message from the CA, attests it, and then relays the same to the Keymaster TA. Once the Keymaster is satified, the *Gatekeeper TA* relays the original encrypted message to the victim TA. The victim TA performs operations and returns back the results.

However, should a malicious *non-persistent* TA deployed by an adversary activate UUID confusion, the communication meant for the *Gatekeeper* TA is now redirected to the malicious TA. This hijack is denoted by red communication paths in Fig. 3. Once an adversarial controlled TA has replaced the *Gatekeeper TA*, it does the following upon receipt of a connection request from *innocent CA*:

- Stops all communications to the Keymaster TA. Reason being that the malicious TA cannot attest to anything, lacking the resources *Gatekeeper TA* had.
- Facilitates normal communication between the *innocent CA* and the victim TA.

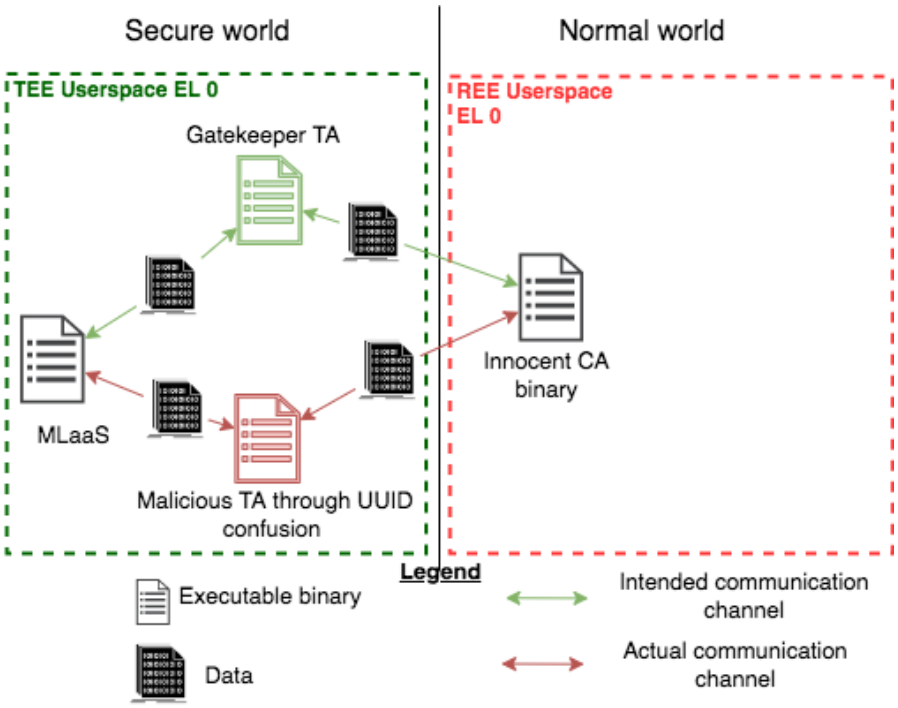

Figure 3: Schematic of combination of UUID confusion based man-in-the-middle attack. Bidirectional green lines denote the actual communication path that should have been followed. The bidirectional red lines denote the actual path followed after a **non-persistent** *malicious TA* (marked in red) is inserted whose UUID is same as that of the **persistent** TA (marked in green).

Stopping communication with the Keymaster TA is not an issue in the present context. The *Gatekeeper TA* was earlier relying on the Keymaster to establish a hardware root-of-trust. Now that an adversary has replaced the *Gatekeeper TA* with a malicious TA, the adversary is no longer concerned with a hardware root-of-trust of all inbound communication. It simply needed to mount a man-in-the-middle attack and get access to the communication meant for the victim TA, which it is successful in doing through UUID confusion.

5.3 Defences defeated by UUID confusion

Among the several defence mechanisms and security features discussed in Sec 1.1, we now note the ones successfully defeated by the UUID confusion exploit.

Breaking source authentication. As mentioned in Sec. 1.1 and Sec. 5.1, the *Gatekeeper* TA is responsible to authenticate all CAs. Once authenticated, a CA can access the TA of its choice. The victim TA was relying on this gatekeeper to ensure only authenticated CAs can access its service. However, through UUID confusion and the fact that the *Gatekeeper* TA resides in the TEE userspace (c.f. Sec. 5.1), we note that a malicious TA is able to replace

the *Gatekeeper* TA and hijack all checks related to source authentication. Since this TA is malicious, it simply allows all CAs to communicate with the victim as if they were correctly authenticated and does not communicate with the Keymaster TA to establish a hardware root-of-trust.

Breaking access control. In Sec. 1.1, we noted that defence mechanisms like *SeCReT* allow keeping an access control list (ACL) wherein every TA can note a list of CAs allowed to access that TAs. We also noted that attacks on *SeCReT* itself are outside scope, implying we do not assume any compromise of the ACL itself. In this backdrop, UUID confusion comes to our aid. Through UUID confusion, an adversary is able to hijack the *Gatekeeper* TA, performing a man-in-the-middle attack (c.f. Fig. 3). The adversary has now a direct communication channel with victim TA. Since this direct communication is solely on the secure world side (i.e. between two TAs), *SeCReT*'s ACL is helpless in preventing this, because it patrols the normal world only.

6 Attack 3: Breaking encryption/authentication through SIGSEGV

From Sec. 4, we have been able to install a malicious self-signed TA into the secure world. From Sec. 5, we have been able to use that loaded TA to mount a man-in-the-middle attack onto the communication meant for the victim TA. However, as we noted earlier, all such communication is encrypted by symmetric keys of which the *innocent CA* (c.f. Fig. 3) and the victim TA are the two owners. Moreover, in Sec. 1.1, we also described how *SeCReT* ensures that the secret keys are not leaked from the CA side to a compromised normal world. *SeCReT* allows a CA to access the secret keys only for the required duration of time, during which it blocks all reads/writes to the memory page holding the key. After encryption/signing is done, *SeCReT* flushes the cache to prevent leakage from any residue memory. Between an *innocent CA* and a victim TA, the sequence of operations follows these steps:

- The victim TA receives encrypted communication from the *innocent CA*.
- The victim TA verifies signature to ensure *innocent CA* is the actual source of the message.
- The victim TA decrypts the message, operates on it, encrypts and signs the output, and sends it back.

In this section, we break this encryption and source-authentication. We do it from the CA side since it is easier to target. We are assuming a symmetric encryption setting implying the TA side keys are automatically leaked if the CA is compromised. We begin by giving an introduction to Linux coredumps, what they contain, how they are generated, and what it means to have coredumps in Trusted Execution Environments. We then elaborate on ways to extract encryption/signing keys of a CA, through coredumps, both for a remote adversary and for a physical adversary.

6.1 Linux coredumps

Linux coredumps are essential utilities that allow access to the image of a process at the time of termination. Among many other triggers, the SIGSEGV signal generates a coredump. Once a coredump is generated, it (along with the binary that received the SIGSEGV signal) can be run through a debugger like GDB or LLD to trace back execution to the point where SIGSEGV was generated. Here, an adversary has access to the entire memory image of the process. Generally, system administrators prevent the system core from dumping by setting a zero limit to `ulimit`.

6.2 Coredumps in TEEs

As mentioned, a sysadmin can prevent a process from dumping the core by putting a zero limit to the `ulimit` on the system. Moreover, changing the `ulimit` is beyond the privileges offered to a non-root user. However, the threat model of Trusted Execution Environments (TEEs) assumes security even in a complete breakdown of trust on the normal world side- the *untrusted* normal world OS threat model normally considered in case of TEEs [46, 47]. We note the interplay of this assumption and the presence of coredumps as a dangerous combination since normal world applications have, to some extent, data which is directly related to the secure world (since a secure world cannot operate in isolation without a normal world). In the next two subsections, we elaborate how this flawed assumption leads to leakage of encryption/signing key that breaks encryption and source authentication.

6.3 Remote adversary: SIGSEGV through `kill`

We start by assuming a remote attack from an adversary and how such an adversary succeeds in leaking encryption/signing key even in the presence of *SeCReT* and other defences. *SeCReT* and other defences offer sound key management services. However, for a short duration of time, the defences *lend* the encryption/signing key to the normal world *innocent CA* (c.f. Fig. 3) which performs encryption/signing. During this process, the defences guard against unauthorized read accesses to the memory page holding the key (schematically depicted in Fig. 4). At some point in its execution, *innocent CA* (c.f. Fig. 3) requests for the keys held by *SeCReT*, which in turns provides the key. After the innocent CA is done, it surrenders the key, wherein caches are flushed and the key is removed from innocent CA's local storage.

However, the CA is vulnerable in the duration where it holds the key. Should an adversary succeed in forcing a coredump *after* the key transfer has finished and *before* the key surrender starts, then the generated coredump would reflect a process image where the key would be with the CA. Therefore, to the adversary, the encryption/signing key would be accessible through analysis of the coredump file.

Assuming a remote attack from an adversary, at the time when the innocent CA is performing the encryption/signing operation, the adversary sends a SIGSEGV through `kill -11 <innocent CA PID>`. As already noted in Sec. 6.2,

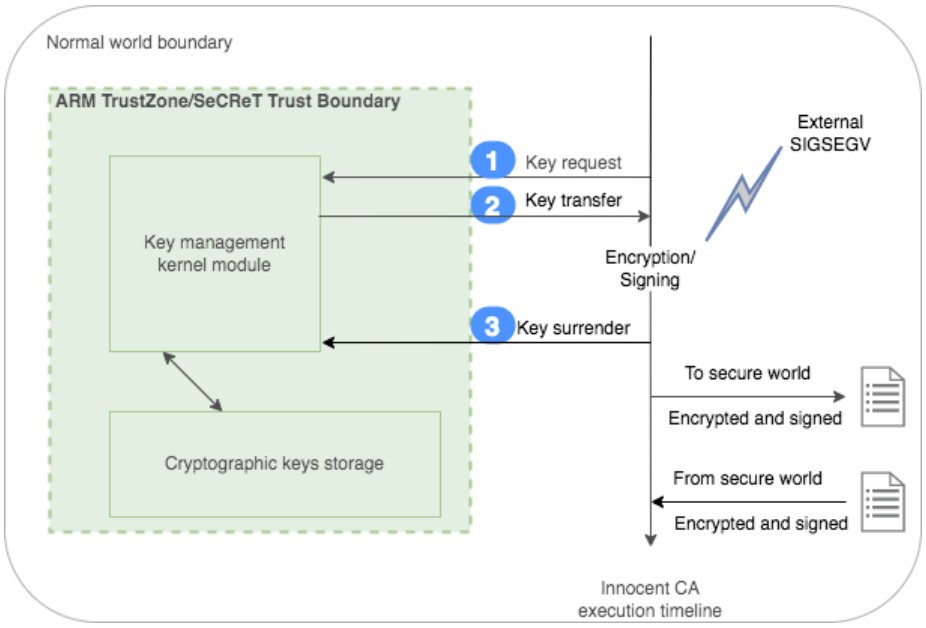

Figure 4: Innocent CA's interaction with *SeCReT* to perform encryption/signing. A carefully timed SIGSEGV leaks the keys while they are in *innocent CA*'s memory.

a kernel configured with coredumps cannot prevent dumping cores in the threat model considered for Trusted Execution Environments. Therefore, *innocent CA*'s core is dumped and accessed by the adversary. Since the core was dumped at a time when the encryption/signing keys were with the *innocent CA*, the keys are present in the coredump and extracted by the adversary.

6.4 Physical adversary: SIGSEGV through EM injections

We now detail the same process of leaking encryption/signing keys with electromagnetic injections. `SIGSEGV` signals arise when memory access violations happen. In programming languages like C, pointer variables are the prime targets of forcing memory access violations.

Observation. Breaking something is far easier than breaking something in a *particular* way. An adversary does not require the precise fault injections of Sec. 4.1 for this phase. They just have to break *something*.

An adversary simply has to inject long and powerful enough electromagnetic pulses onto the memory chip embedded into the Raspberry Pi 3 board. When the *innocent CA* does a memory operation, the EM injection is successful in polluting values of the pointers while execution of `STR/LDR` instructions. More often than not, we empirically observed `SIGSEGV` signals being raised and cores being dumped. We emphasize this point further in Listing 4. After the point

where keys are gathered from *SeCReT*, a powerful enough fault injection for a long duration of time can corrupt a memory access operation. We assume that such encryption.signing operations do not complete their entire execution without needing any memory accesses. An adversary is required to corrupt any one of these memory accesses after keys from *SeCReT* are gathered to dump the core and retrieve both the encryption and the signing keys.

```
void encrypt_message(char* encrypted_buffer, char* buffer, char* key){
    /*encryption of messages meant for victim TA*/
}

void sign_message(char* sign_buffer, char* encrypted_buffer, char* key){
    /*signing of the encrypted messages*/
}

void encrypt_and_sign(){
    /*gather keys from SeCReT*/
    char* encryption_key = getEncryptionKey();
    char* signing_key = getSigningKey();
    /*--------- FAULT WINDOW ----------*/
    encrypt_message(/*encrypted_buffer*/, /*buffer*/, encryption_key);
    signing_key(/*sign_buffer*/, /*encrypted_buffer*/, signing_key);
    /*--------- FAULT WINDOW ----------*/
    /*surrender keys*/
}
```

Listing 4: Encryption and signing of messages on the innocent CA side.

6.5 Defences defeated by SIGSEGV faults

In Sec. 1.1, we introduced several security factors and elaborated the third-party extensions responsible for securing those factors. In Sec. 5.3, we successfully demonstrated breaking source authentication and bypassing *SeCReT*'s access control lists. However, by then, although we had a successful man-in-the-middle attack and a direct communication channel with the victim TA, it was not of much use since the communication channel was still encrypted.

Breaking communication privacy. In the current section, however, we demonstrated how careful timing of SIGSEGV signals and flawed assumptions on coredumps in Trusted Execution Environments leak encryption/signing keys even in the presence of defences like *SeCReT*.

7 Experimental details

In this section, we give experimental details of all three attack vectors. Our test setup is depicted in Fig. 5. The victim device is a Raspberry Pi 3 Model B. The adversarial device is a Raspberry Pi 4, which is connected to a Keysight 33500B signal generator responsible for triggering the entire fault injection process. When the Raspberry Pi 4 produces a digital `HIGH` on one of the adversarial controlled GPIO pins, the Keysight 33500B signal generator forwards the signal to a Keysight 81160A pulse train generator. The Keysight 81160A pulse train generator generates 15 pulses of frequency 200 MHz, pulse width 2 nanoseconds, and amplitude -8.13 dbm. These 15 pulses are received by the signal amplifier and amplified to pulses with frequency 400 MHz (as a comparison, the lowest operational frequency of Raspberry Pi 3 Model B is 600 MHz). These amplified pulses are received by Rigol NFP-3 P3 EM (electromagnetic) probe and

Table 1: OP-TEE monorepo state at the time of the attack. `HEAD` represents the commit ID of the topmost commit in a particular sub-repository.

Sub-repository	HEAD of the commit tree	HEAD's timestamp
buildroot [49]	e6e12337f1874a5a53b42badf3d7fdd258d86a38	Dec 5 20:59:16 2021
edk2 [50]	b24306f15daa2ff8510b06702114724b33895d3c	Jan 26 13:12:21 2022
linux [51]	b9a16995c467cc18cc26716d566c512fbac11069	Jun 30 21:48:49 2022
mbedtls [52]	e483a77c85e1f9c1dd2eb1c5a8f552d2617fe400	Mar 12 16:55:26 2021
optee_benchmark [53]	875be7f1959f19ed601ef37501f1cf0bfbee9da4	May 30 20:34:57 2020
optee_client [54]	f7ed8e3d3955e0b7a7d3ff77ab2abcfd8fb1cdb9	Apr 18 09:53:32 2022
optee_os [55]	837adc0a4c5dc462bfcc690618b812d838534fa5	Jun 28 16:13:07 2022
optee_test [56]	da5282a011b40621a2cf7a296c11a35c833ed91b	Apr 7 11:24:08 2022
trusted-firmware-a [57]	a1f02f4f3daae7e21ee58b4c93ec3e46b8f28d15	Nov 23 14:14:26 2021
u-boot [58]	b46dd116ce03e235f2a7d4843c6278e1da44b5e1	Apr 5 11:03:29 2021

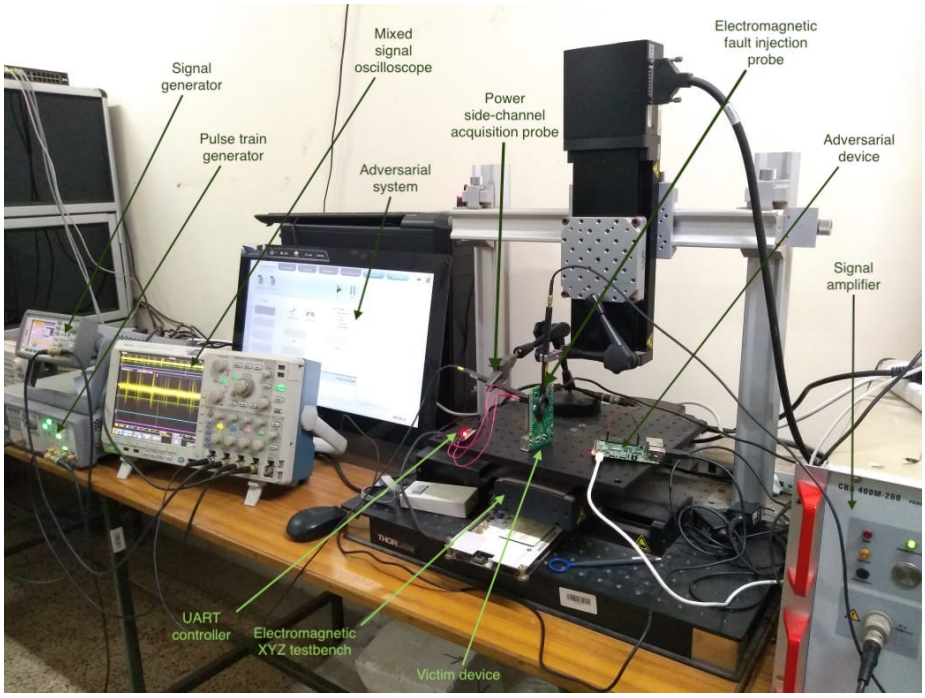

Figure 5: The fault testbed developed to mount the attacks mentioned in this paper. The victim device is mounted upon a XYZ table allowing careful positioning of the electromagnetic (EM) probe. Note that the EM probe is positioned to inject EM pulses from the backside of the device: thereby faulting both the Broadcom processor situated on the frontside as well as the memory chip located on the backside. A power side-channel analysis probe is mounted on the Broadcom processor on the frondside to gather power traces. All signals are recorded on a mixed signal oscilloscope.

injected onto the system. The OP-TEE codebase was compiled with the default optimization flag [48]. Finally, table 1 gives the state of the OP-TEE monorepo at the time of the attack.

7.1 Attack 1: Installing TAs

To generate a TA-like binary, the adversary creates a normal C based-program based off the GlobalPlatform's Internal Core API specification [39]. This binary is named <uuid>.ta where <uuid> denotes the publicly known UUID of the *Gatekeeper* TA (c.f. Sec. 7.2 for more details of actual choice of the UUID). The adversary- having root access in Ring 3 on the normal world Linux as per considered threat model- loads this binary <uuid>.ta in lib/optee_armtz. Upon being invoked by a CA, <uuid>.ta is loaded and checked for verification, wherein we bypass the check completely by cleaning the register holding the return value.

7.1.1 Searching the fault parameter space

From the testbed shown in Fig. 5, there are several parameters to be tweaked. Searching through the entire parameter space is an exponential problem, and we require heuristics to converge upon a range of parameters most likely to give faults of our interest. There is also another problem: our empirical observations suggest that introducing *incorrect* faults in OP-TEE TA loading driver causes immediate reboots. This happens because poorly localised faults cause memory corruption.

To circumvent this and to find fault parameters, we decided to search for optimal parameters on a dummy program in the normal world Linux (c.f. Listing 5). From analyzing the disassembly of the signature verification process presented in listing 2, we know that a certain `mov` instruction updates value of a variable, which is later used in a `cbnz` instruction to abort execution if needed. Since this is a profiling phase undertaken by an adversary to find optimal fault parameters, we can rely on code-based triggers. We use three triggers and two delays to precisely isolate the actual instruction to be faulted. Refer to the oscilloscope output in Fig. 6. The reason for adding three triggers in Listing 5 was to distinctly observe when three separate events are taking place: ① actual fault injection trigger through a GPIO pin, ② `mov` instruction execution, and ③ beginning of verification based off the value updated in ②. Similarly, the reason for using two delays (especially a longer second delay) was to understand if the injected fault is actually being isolated in the region of interest (i.e. `mov` instruction execution) or is it percolating into the verification step too.

```
#define GPIO_PIN 7
#define TEE_SUCCESS 0x00000000

void search_fault_parameter_space(){
    uint32_t res;          // the sample variable to fault
    trigger(GPIO_PIN);  // trigger Keysight 33500B signal generator

    for(int i = 0; i < 20; i++)
        for(int j = 0; j < 20; j++);    // programmable delay

    trigger(GPIO_PIN);  // signal beginning of "mov" operation

    __asm__ __volatile__("mov %0, #4294901775"
        : "=r"(res));   // load 0xFFFF000F into "res"

    for(int i = 0; i < 20; i++)
        for(int j = 0; j < 100; j++);    // a little longer programmable delay

    trigger(GPIO_PIN);      // signal beginning of verification
    if(res == TEE_SUCCESS)
```

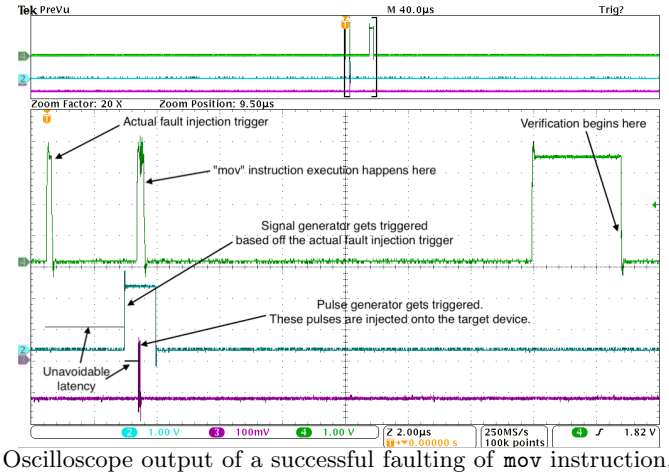

Figure 6: Oscilloscope output of a successful faulting of `mov` instruction, as depicted in Fig 7. The topmost signal represents the input coming from Raspberry Pi 3. The middle signal represents the output of the Keysight 33500B signal generator. The lowermost signal represents the output of the Keysight 81160A pulse train generator. Note that there are unavoidable latencies between these signals. It is therefore crucial that in the victim device, to fault a `mov` instruction, the trigger should be raised sometime before execution of `mov` instruction starts. The actual value of this delay varies from device to device. Moreover, for a successful faulting, all three signals should be aligned in time (as they are in the figure) exactly at the time of `mov` instruction exection.

Table 2: Set of parameter values chosen for our setup.

Parameter	Signal generator values	Pulse train generator values
Frequency	10 KHz	200 MHz
Amplitude	2V	-8.13 dBm
Offset	1V	0V
Pulse width	1 micro-second	2 nano-seconds
Cycle count	1	15
Trigger threshold	1V	1V
Trigger signal	Rising edge	Rising edge
Trigger delay	2.312 micro-seconds	0 micro-seconds
Device operational frequency	600 MHz	600 MHz

```
21          printf("[SUCCESS] Register value corrupted to 0x%lx\n", res);
22      else if(res != 0xFFFF000F)
23          printf("[PARTIAL SUCCESS] Register value corrupted to 0x%lx\n", res);
24      else
25          printf(".");     // No corruption
26  }
```

Listing 5: A dummy program isolating the `mov` instruction used by OP-TEE signature verification process to signal attempts of loading self-signed TAs. From the parameter set tuned to this `mov` instruction from this program, we fault the signature verification step on OP-TEE.

From repeating the experiments over 1,000,000 times, we observed higher probabilities of faulting the `mov` instruction with the parameter set detailed in table 2. While frequency, amplitude, and pulse width are self-explanatory parameter sets, we shed some light on the others. Offset refers to the y-intercept value of the signal (i.e. the amount by which a signal is shifted in the y-axis).

Cycle count refers the number of square pulses to generate once the trigger is received. Trigger threshold refers to the threshold above which the amplitude of an input signal causes the signal generators to start operating. Trigger signal means that the trigger must be checked on the rising edge of the input signal. Trigger delay refers to the additional wait-time a signal generator waits from the moment it is triggered to the moment when it actually starts producing output. Trigger delay is essential to account for the unavoidable latencies mentioned in Fig. 6. Through these parameters, the faults injected on the program in Listing 5 were as Fig. 7 depicts.

7.1.2 Analysing power traces for triggering mechanism

In Sec 7.1.1, we assumed presence of code-based triggers. To circumvent this requirement and develop completely non-invasive triggers, we look for specific desirable patterns in the power trace outputs. Consider Fig. 8. OP-TEE's verification process involves a RSA signature verification. By observing power trace output, we narrowed down the part of oscilloscope output where the power traces match those of known RSA power traces [59]. Since the `mov` instruction of our interest comes after RSA signature verification, this pattern is hereafter used to trigger fault injections.

7.2 Attack 2: UUID confusion

The crux of our second attack vector was that since TA UUIDs are considered *non-sensitive* information that is *publicly* available, a malicious TA can easily reuse the UUID of another TA and OP-TEE does not complain about this reuse. In listing 6, we present the boilerplate code that an *innocent CA* would use to initiate communication with the secure world TA. In listing 7, we present the `TA_FLAGS` the malicious TA uses to force UUID confusion and transfer traffic to itself. The UUID confusion attack is an *instantaneous* attack that is mounted the moment two TAs with same UUIDs are loaded, until the time one of the TAs is removed from the system. Of all the experiments we performed, a non-persistent malicious TA, through UUID confusion, was always able to redirect traffic meant for another persistent TA, as depicted in in Fig 9.

```
1 void authenticate_with_TEE(char* ca_binary_digest, uint8_t* signature){
2     TEEC_UUID ta_uuid = RANDOM_CONST_UUID; /* UUID of the TA to be invoked */
3     TEEC_InitializeContext(NULL, ta_ctx); /* Initialize context with TEE */
4     TEEC_OpenSession(ta_ctx, ta_session, ta_uuid, TEEC_LOGIN_PUBLIC, ....); /* Open
       session with the TA */
5     TEEC_InvokeCommand(ta_session, /* command */, /* parameters */, /*exit code
       reference */); /* invoke command to decrypt ciphertext passed in 'parameters'
       */
6     TEEC_CloseSession(ta_session);
7     TEEC_FinalizeContext(ta_ctx);
8 }
```

Listing 6: Innocent CA accessing a TA with UUID `RANDOM_CONST_UUID`

```
1 ////////////////// Persistent innocent TA ////////////////////
2 #define TA_FLAGS (TA_FLAG_SINGLE_INSTANCE | TA_FLAG_INSTANCE_KEEP_ALIVE |
      TA_FLAG_MULTI_SESSION)
3 #define ta_uuid RANDOM_CONST_UUID
4
5 void authenticate(const char* digest, const uint8_t* signature){
6     /* initialize context and accept session from CA */
7     TEE_AsymmetricVerifyDigest(/* operation handle */, NULL, 0, digest,
       digest_length, signature, signature_length);
```

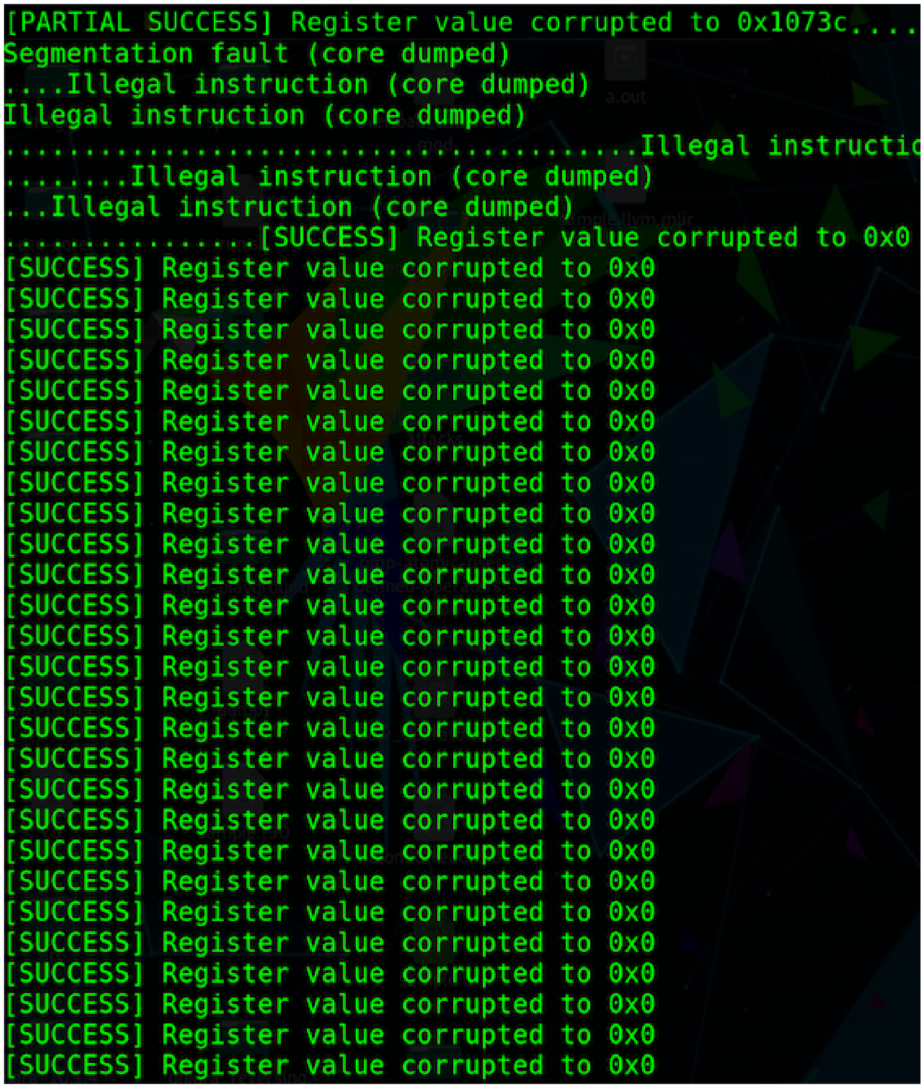

Figure 7: The faults observed while attacking `mov` instruction in Listing 5 with the parameter set detailed in Tab 2. Every dot (.) represents a correct expected execution. The `SIGSEGV` and `SIGILL` faults observed actually percolate to the instructions following the `mov` instruction. However, as the screenshot depicts, we were able to observe one fault wherein `mov` loaded an incorrect value into `res`. And then we observed many faults wherein `mov` cleared out `res` completely. This empirical observation is the foundation of the countermeasure proposed in Sec 8.1.

```
    /* teardown context and close session */
}
///////////////// Non-persistent malicious TA //////////////////
#define TA_FLAGS (TA_FLAG_EXEC_DDR)
#define ta_uuid RANDOM_CONST_UUID /* same UUID as innocent TA */

void authenticate(const char* digest, const uint8_t* signature){
    /* initialize context and accept session from CA */
```

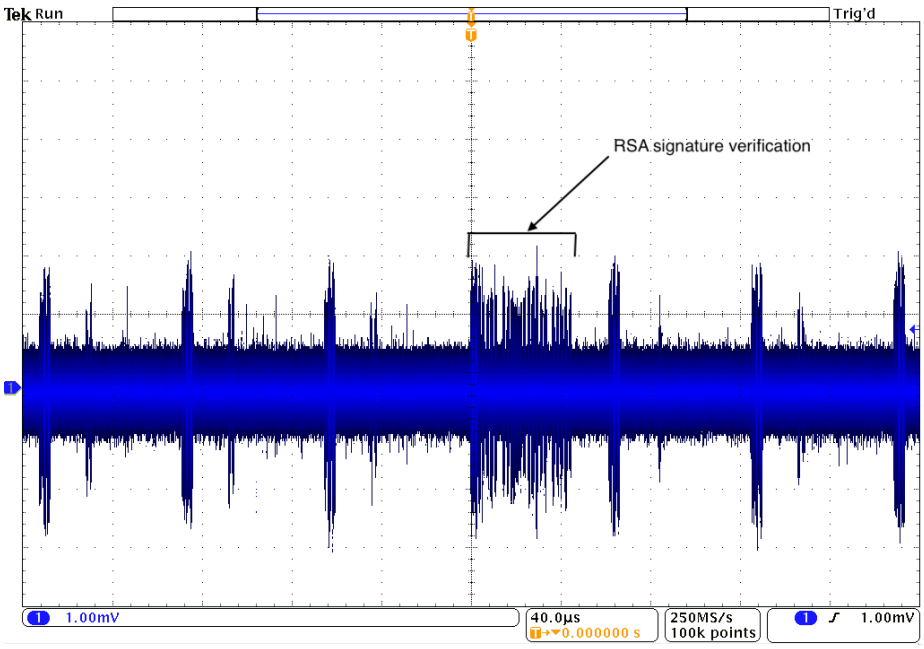

Figure 8: The zoomed-in oscilloscope output during OP-TEE's verification of TAs.

```
16      TEE_AsymmetricVerifyDigest(/* operation handle */, NULL, 0, digest,
         digest_length, signature, signature_length);
17      /* teardown context and close session */
18  }
```

Listing 7: Two TAs with same UUID `RANDOM_CONST_UUID`

7.3 Attack 3: Breaking encryption/authentication

The core idea of our third attack vector was to force a premature memory access violation in the *innocent CA*'s execution *after* it has received encryption and signed keys from *SeCReT*. From the dumped core, an adversary is able to recover both the encryption and the signing keys of an *innocent CA*. Here, we give intuition on the recovery of keys from the dumped cores. The main problem in such analysis is that OP-TEE's normal world Linux kernel has no support for debuggers. This means all analysis and key recovery is an offline process. However, a core problem with offline analysis of coredumps is the absence of symbols, making it harder. As such, dedicated reverse engineering is needed to recover the keys from the coredumps. We followed the following process post a coredump:

- **Online Step.** Since an adversary has root access in Ring 3 of normal world Linux kernel, we were able to extract the coredump and the *innocent CA* binary from OP-TEE to the adversarial controlled system.

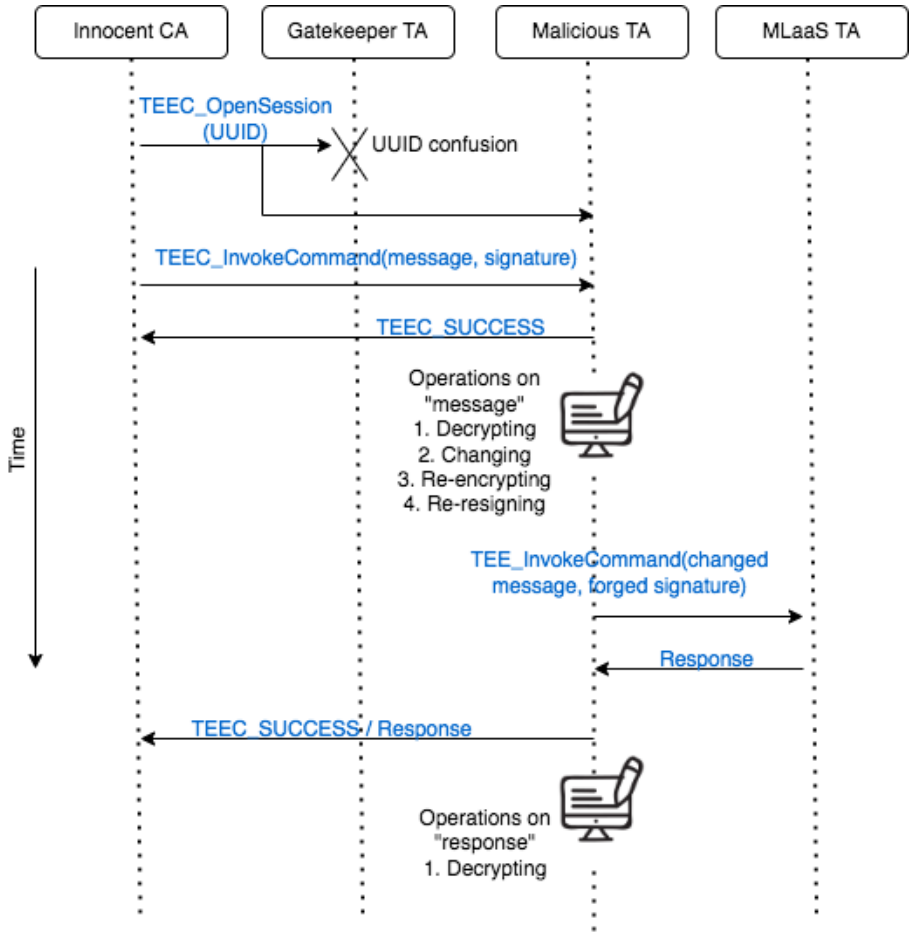

Figure 9: The flow of API calls denoting UUID confusion. `TEEC_OpenSession` is the TEE Client side API call responsible for opening a session with a TA. Based off the UUID passed as parameter, a TA is loaded into secure memory, and all subsequent operations occur with the loaded TA.

- **Offline Step.** We disassembled the *innocent CA* binary through `aarch64-linux-gnu-objdump` obtained from publicly available ARM's aarch64 toolchain.

- **Offline Step.** From analysing the coredump in `aarch64-linux-gnu-gdb` again obtained from ARM's aarch64 toolchain as well as from the disassembly of *innocent CA* binary, a function call stack was constructed.

- **Offline Step.** Once a function call stack was successfully constructed, we used `aarch64-linux-gnu-gdb` to dump the stack frames for each function. If listing 4 is considered, we would be interested in the stack frame for `encrypt_and_sign()` function, which would contain the encryption and signing keys on the stack.

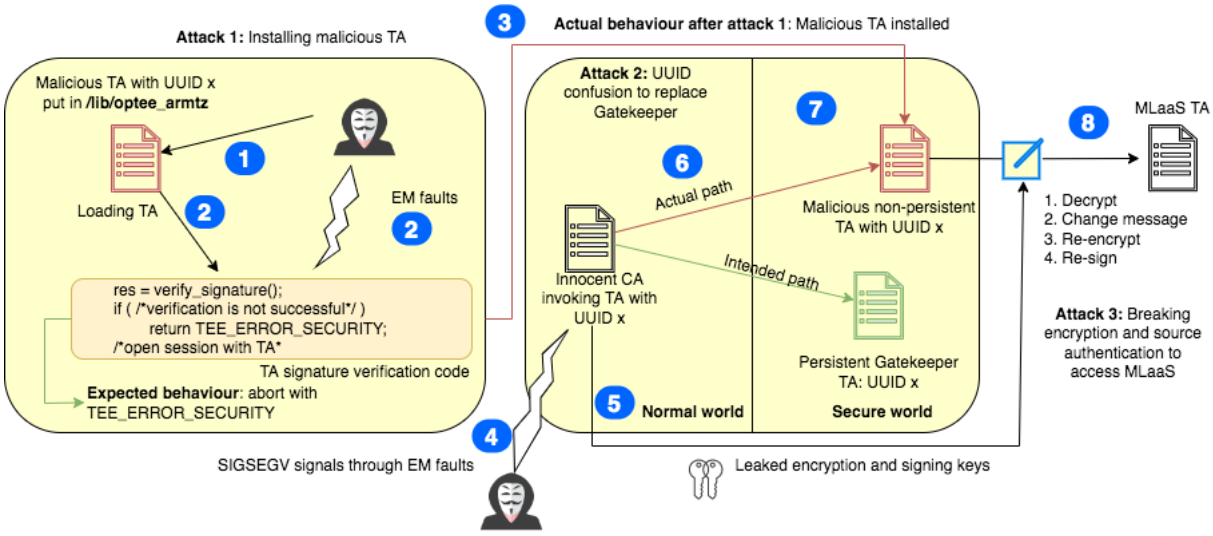

Figure 10: End-to-end attack on a MLaaS TA. Steps of the attack: ① attacker loads self-signed TA, ② while signature verification is happening for self-signed, attacker introduces an EM fault to bypass verification, ③ self-signed TA is loaded, ④-⑤ through `SIGSEGV` signals introduced by EM faults, encryption/signing keys are leaked, ⑥-⑦ UUID confusion causes control transfer to reach malicious TA instead of *Gatekeeper* TA, ⑧ using leaked encryption/signing keys to send arbitrary, source-authenticated messages to MLaaS TA.

7.4 Accessing private MLaaS

In this section, we bring together all attack vectors to explain how an end-to-end attack works on a possible use-case: MLaaS server running in the secure world. MLaaS is a perfect demonstration example because of its critical security-centric nature, the need to prevent unauthorized access to it, and the economic loss accumulated should MLaaS start servicing unauthorized requests. We implemented a convolutional neural network to classify MNIST dataset images. For an input of size 28×28, we applied two back-to-back convolutions that respectively had 32 and 64 5×5 convolutional filters. Finally, there were two fully connected layers with output size 1024 and 10 respectively. A zero-padding was used during convolution operations, and max pooling was the chosen strategy for pooling operations.

We had two security dimensions discussed in Sec. 1.1. The security dimension 2 (c.f. Sec. 1.1.2) is concerned with TEE virtualization in presence of many TEEs. We do not discuss it since our attack vectors succeed without taking them into consideration. Security dimension 1 (c.f. Sec. 1.1.1), however, aiming to secure the entire communication flow had three facets: source authentication, granular access control, and communication privacy. We break the first two with our UUID confusion exploit. The last is broken by SIGSEGV based coredumps. Both of these attacks are aided by installation of a self-signed malicious TA, which then intercepts and decrypts all communication passing between the *in-*

nocent CA and the MLaaS (c.f. Fig. 3). Once decrypted, the malicious TA can modify the communication, re-encrypt it, and send the modified ciphertext to MLaaS. Once a response is received from MLaaS, the malicious TA decrypts it. Thereby, an adversary is able to get complete access to a private, pay-per-use MLaaS service. Moreover, on a more general note, the adversary has been successful in breaking the promised isolation between the normal world and the secure world through our attack vectors. This entire end-to-end attack is depicted in Fig. 10.

8 Countermeasures

Here, we propose some software-based countermeasures for the different attack vectors proposed in the paper.

8.1 Attack 1: Installing TAs

This portion of the attack has been successful because of the interplay of two independent factors: ① Raspberry Pi Model 3 B's Broadcom processor being exposed without a metallic shield, ② OP-TEE using `0x0` to denote success. Therefore, we were able to inject powerful enough electromagnetic pulses to clear out the register holding error code value to `0x0`, thereby bypassing signature verification. For this, we suggest two software countermeasures: ① port OP-TEE to Raspberry Pi Model 4 which has a metallic shielding over its processor making it harder for electromagnetic (EM) attacks to succeed, and/or ② change return values denoting success from `0x0` to a non-zero value. From our empirical observations (c.f. Fig. 7), EM injections find it easier to clear a register than to set it to a particular value. In countermeasure ①, however, there is a caveat. Raspberry Pi 3 still remains a popular SoC for IoT networks [4, 5]. Therefore, porting in-production software to Raspberry Pi 4 not only requires a non-trivial engineering effort, but the ported software also needs to be validated in case the porting creates newer attack surfaces.

8.2 Attack 2: UUID confusion

For UUID confusion, as we demonstrated, it is dangerous to let two TAs share UUID. Therefore, TEE OS based software checks to ensure uniqueness will prevent UUID confusion based attacks.

8.3 Attack 3: Breaking encryption/authentication

It is not sufficient to switch off coredumps in the case of Trusted Execution Environments, since a root level adversary on Ring 3 of the normal world can switch it back on. This lies within the attack model of Trusted Execution Environments (TEEs) since TEEs claim security even with complete breakdown of trust in the normal world. Therefore, we propose to not configure the normal world kernel with coredumps at all. Therefore, the normal world kernel should be configured with `CONFIG_COREDUMP` option disabled.

9 Conclusion

In this paper, we evaluated the claimed security of an implementation of ARM TrustZone (OP-TEE) on a System-on-Chip (Raspberry Pi 3). We evaluated ARM TrustZone based and third-party defence mechanisms protecting Trusted Applications (TAs) running in the system. To this end, we gave a three stepped end-to-end attack vector wherein we ① bypassed signature verification to install a self-signed TA through electromagnetic fault injections triggered by non-invasive side-channel power traces, ② invoked UUID confusion to use the TA installed in 1) to hijack communication meant for another TA, and ③ injected `SIGSEGV` signals through electromagnetic faults to leak encryption and signing keys. With these attack vectors working together, we were able to break normal world and secure world isolation by performing man-in-the-middle attack on encrypted/authenticated communication channels as well as decrypting the communication. We also showed the end-to-end attack on a Machine-learning-as-a-service (MLaaS) TA running on the secure world. Finally, we proposed purely software based countermeasures to the attacks.

References

[1] ARM., "ARM TrustZone." https://www.arm.com/technologies/trustzone-for-cortex-m.

[2] L. Chen, J. Li, R. Ma, H. Guan, and H.-A. Jacobsen, "Enclavecache: A secure and scalable key-value cache in multi-tenant clouds using intel sgx," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 14–27.

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

[4] S. Kurkovsky and C. Williams, "Raspberry pi as a platform for the internet of things projects: Experiences and lessons," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, 2017, pp. 64–69.

[5] C. P. Kruger and G. P. Hancke, "Benchmarking internet of things devices," in *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, 2014, pp. 611–616.

[6] J. Bech and V. Chong., "Keymaster and Gatekeeper," https://static.linaro.org/connect/yvr18/presentations/yvr18-414.pdf, 2018.

[7] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "Secret: Secure channel between rich execution environment and trusted execution environment." in *NDSS*, 2015, pp. 1–15.

[8] J. Jang and B. B. Kang, "Securing a communication channel for the trusted execution environment," *computers & security*, vol. 83, pp. 79–92, 2019.

[9] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing arm trustzone," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 541–556.

[10] R. Liu and M. Srivastava, "Virtsense: Virtualize sensing through arm trustzone on internet-of-things," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018, pp. 2–7.

[11] H. C. Tanuwidjaja, R. Choi, S. Baek, and K. Kim, "Privacy-preserving deep learning on machine learning as a service—a comprehensive survey," *IEEE Access*, vol. 8, pp. 167 425–167 447, 2020.

[12] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman, "Sok: Security and privacy in machine learning," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 399–414.

[13] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving machine learning as a service," *arXiv preprint arXiv:1803.05961*, 2018.

[14] D. Bacciu, S. Chessa, C. Gallicchio, and A. Micheli, "On the need of machine learning as a service for the internet of things," in *Proceedings of the 1st International Conference on Internet of Things and Machine Learning*, 2017, pp. 1–8.

[15] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against intel sgx," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1466–1482.

[16] A. Tang, S. Sethumadhavan, and S. Stolfo, "{CLKSCREW}: Exposing the perils of {Security-Oblivious} energy management," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1057–1074.

[17] P. Qiu, D. Wang, Y. Lyu, R. Tian, C. Wang, and G. Qu, "Voltjockey: A new dynamic voltage scaling-based fault injection attack on intel sgx," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 6, pp. 1130–1143, 2020.

[18] J. Van Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017, pp. 1–6.

[19] S. D. Kumar, S. Patranabis, J. Breier, D. Mukhopadhyay, S. Bhasin, A. Chattopadhyay, and A. Baksi, "A practical fault attack on arx-like ciphers with a case study on chacha20," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2017, pp. 33–40.

[20] G. Canivet, P. Maistri, R. Leveugle, J. Clédière, F. Valette, and M. Renaudin, "Glitch and laser fault attacks onto a secure aes implementation on a sram-based fpga," *Journal of cryptology*, vol. 24, no. 2, pp. 247–268, 2011.

[21] C. O'Flynn, "{MIN () imum} failure:{EMFI} attacks against {USB} stacks," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[22] S. Nashimoto, D. Suzuki, R. Ueno, and N. Homma, "Bypassing isolated execution on risc-v with fault injection," *Cryptology ePrint Archive*, 2020.

[23] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald, and F. D. Garcia, "{VoltPillager}: Hardware-based fault injection attacks against intel {SGX} enclaves using the {SVID} voltage scaling interface," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 699–716.

[24] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert, "One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2875–2889.

[25] T. L. K. documentation., "CPU Performance Scaling," https://www.kernel.org/doc/html/v4.14/admin-guide/pm/cpufreq.html.

[26] R. P. Documentation., "Configuration," https://www.raspberrypi.com/documentation/computers/configuration.html.

[27] P. Ravi, D. B. Roy, S. Bhasin, A. Chattopadhyay, and D. Mukhopadhyay, "Number "not used" once-practical fault attack on pqm4 implementations of nist candidates," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2019, pp. 232–250.

[28] S. K. Bukasa, R. Lashermes, J.-L. Lanet, and A. Leqay, "Let's shock our iot's heart: Armv7-m under (fault) attacks," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, 2018, pp. 1–6.

[29] K. Xagawa, A. Ito, R. Ueno, J. Takahashi, and N. Homma, "Fault-injection attacks against nist's post-quantum cryptography round 3 kem candidates," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2021, pp. 33–61.

[30] T. Trouchkine, S. K. Bukasa, M. Escouteloup, R. Lashermes, and G. Bouffard, "Electromagnetic fault injection against a system-on-chip, toward new micro-architectural fault models," *arXiv preprint arXiv:1910.11566*, 2019.

[31] C. Bozzato, R. Focardi, and F. Palmarini, "Shaping the glitch: optimizing voltage fault injection attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 199–224, 2019.

[32] T. Korak and M. Hoefler, "On the effects of clock and power supply tampering on two microcontroller platforms," in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2014, pp. 8–17.

[33] L. Wouters, B. Gierlichs, and B. Preneel, "On the susceptibility of texas instruments simplelink platform microcontrollers to non-invasive physical attacks," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2022, pp. 143–163.

[34] J. G. Van Woudenberg, M. F. Witteman, and F. Menarini, "Practical optical fault injection on secure microcontrollers," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2011, pp. 91–99.

[35] optee blog, "OP-TEE Blog ," https://www.trustedfirmware.org/blog/, 2021.

[36] Apertis., "Integration of OP-TEE in Apertis." https://www.apertis.org/concepts/op-tee/.

[37] iWave., "Securing Edge IoT devices with OP-TE." https://www.iwavesystems.com/news/securing-edge-iot-devices-with-op-tee/.

[38] GlobatPlatform., "TEE Client API specification," https://globalplatform.org/specs-library/tee-client-api-specification/.

[39] ——, "TEE Internal Core API specification," https://globalplatform.org/wp-content/uploads/2016/11/GPD_TEE_Internal_Core_API_Specification_v1.2_PublicRelease.pdf.

[40] OP-TEE., "Trusted Applications signing process." https://optee.readthedocs.io/en/latest/building/trusted_applications.html/.

[41] O. Lo, W. J. Buchanan, and D. Carson, "Power analysis attacks on the aes-128 s-box using differential power analysis (dpa) and correlation power analysis (cpa)," *Journal of Cyber Security Technology*, vol. 1, no. 2, pp. 88–107, 2017.

[42] Y. Kim, T. Sugawara, N. Homma, T. Aoki, and A. Satoh, "Biasing power traces to improve correlation power analysis attacks," in *First international workshop on constructive side-channel analysis and secure design (cosade 2010)*. Citeseer, 2010, pp. 77–80.

[43] D. Bayhan, S. B. Ors, and G. Saldamli, "Analyzing and comparing the montgomery multiplication algorithms for their power consumption," in *The 2010 International Conference on Computer Engineering & Systems*. IEEE, 2010, pp. 257–261.

[44] A. P. Fournaris and O. Koufopavlou, "A new rsa encryption architecture and hardware implementation based on optimized montgomery multiplication," in *2005 IEEE International Symposium on Circuits and Systems*. IEEE, 2005, pp. 4645–4648.

[45] H. Nozaki, M. Motoyama, A. Shimbo, and S. Kawamura, "Implementation of rsa algorithm based on rns montgomery multiplication," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2001, pp. 364–376.

[46] D. C. G. Valadares, N. C. Will, J. Caminha, M. B. Perkusich, A. Perkusich, and K. C. Gorgônio, "Systematic literature review on the use of trusted execution environments to protect cloud/fog-based internet of things applications," *IEEE Access*, vol. 9, pp. 80 953–80 969, 2021.

[47] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: what it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 57–64.

[48] G. Documentation., "Options That Control Optimization." https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[49] Buildroot., "Buildroot." https://git.busybox.net/buildroot/.

[50] Tianocore., "edk2." https://github.com/tianocore/edk2/.

[51] L. S. W. Group., "Linux." https://github.com/linaro-swg/linux.

[52] Mbed-TLS., "mbedtls." https://github.com/Mbed-TLS/mbedtls/.

[53] L. S. W. Group., "OP-TEE benchmarks." https://github.com/linaro-swg/optee_benchmark.

[54] OP-TEE., "OP-TEE client." https://github.com/OP-TEE/optee_client.

[55] ——, "OP-TEE OS." https://github.com/OP-TEE/optee_os.

[56] ——, "OP-TEE tests." https://github.com/OP-TEE/optee_test.

[57] L. S. W. Group., "Trusted Firmware." https://github.com/linaro-swg/arm-trusted-firmware.

[58] U-Boot., "U-Boot." https://github.com/u-boot/u-boot.

[59] Myrelabs, "BREAKING RSA WITH CHIPWHISPERER," https://myrelabs.com/breaking-rsa-with-chipwhisperer/, 2019.