Faults in Our Bus: Novel Bus Fault Attack to Break ARM TrustZone

Nimish Mishra, Anirban Chakraborty, Debdeep Mukhopadhyay
Indian Institute of Technology Kharagpur
nimish.mishra@kgpian.iitkgp.ac.in, anirban.chakraborty@iitkgp.ac.in, debdeep@cse.iitkgp.ac.in

***Abstract*—The ever-increasing growth of Internet-of-Things (IoT) has led to wide-scale deployment of high-frequency, highly complex *Systems-on-a-Chip* (SoCs), which are capable of running a full-fledged operating system (OS). The presence of OS and other software countermeasures make SoCs resilient against the traditional fault attacks that are relevant on FPGAs and microprocessors. In this work, we present the first practical implications of targeting an orthogonal aspect of SoC's architecture: *the system bus*. We inject electromagnetic pulses onto the system bus during the execution of instructions involving processor-memory interaction. We show how address bus faults compromise software implementations of masked implementations of ciphers, illustrated using implementations of state-of-the-art post-quantum cryptography (PQC) schemes, leaking entire secret keys *with a single fault*. We also demonstrate that data bus faults can be controlled and exploited to launch Differential Fault Analysis (DFA) attacks on table-based implementation of the Advanced Encryption Standard (AES). Furthermore, we demonstrate that the impact of such bus faults can be far-reaching and mislead the security guarantees of the popular and widely used ARM TrustZone. We use data-bus faults (along with loopholes in the GlobalPlatform API specification) to mislead the signature verification step to load a malicious Trusted Application (TA) inside the TrustZone. We follow this up with address bus faults to steal symmetric encryption keys of other benign TAs in the system, leading to complete breakdown of security on TrustZone. We note that since the attack relies upon loopholes in the GlobalPlatform API specification, it is portable to any TEE following this specification. To emphasize upon this portability of the attack, we demonstrate successful installation of malicious TAs on two TrustZone implementations (OP-TEE and MyTEE) on two different platforms (Raspberry Pi 3 and Raspberry Pi 4). Finally, we propose countermeasures that can be integrated into the SoC environment to defend against these attack vectors.**

I. Introduction

The Internet of Things (IoT) has become an essential part of modern-day life, catering to billions of users across various applications, starting from home appliances to critical services in different industries. The growth and inclusion of IoT in daily life have been so pervasive that it is estimated that the number of IoT-connected devices worldwide is projected to be 30.9 billion units by 2025 [1]. Quite naturally, the security aspect of these devices in the IoT hemisphere has intrigued researchers for the past few years. The security model of IoT ecosystem presents newer threats and opportunities as the majority of these embedded devices are deployed *in-the-wild*, mostly without any human supervision, therefore exposing themselves to physical adversaries. Moreover, the range of devices used in IoT ecosystem varies widely, depending on their respective usage and application. This poses a challenging scenario to evaluate the security guarantees of such devices and the application running thereof.

Due to their nature of deployment and physical exposure, the possibility of active physical attacks becomes very pertinent. In such scenarios, an adversary can mount a plethora of physical attacks, including (but not limited to) fault injection (FI) attacks [2]. Fault Injection attacks are a class of active attacks where the adversary, having physical (or remote) access to the system, can perturb intermediate computations of the target algorithm in a seemingly controlled manner. The intention of the adversary is to either make the victim device enter into an erroneous state [3] or to introduce statistical bias in the target algorithm, which can be later exploited to leak secret information [4], [5]. Over the years, the literature on FI attacks has been enriched with myriads of techniques which can be broadly classified into three types - ① manipulating system parameters such as clock frequency or voltage, ② changing external operating environment such as temperature, and ③ injecting external electromagnetic or optical (laser) pulses.

Fault Injection Techniques. The idea of fault attacks was first applied on RSA-CRT [6], [7] and eventually on DES by the introduction of Differential Fault Analysis (DFA). The gamut of FI attacks has two separate aspects - *Fault Injection*, where faults are practically injected into the target system (or circuit) and *Fault Analysis*, which deals with the aftermath of the fault injection to fulfil adversary objective. Traditionally, the target of fault injection attacks has been cryptographic implementations on hardware. For instance, in [8], dynamic laser faults are used to target AES implementations in static RAM-based Field Programmable Gate Arrays (FPGAs). Likewise, in [9], the security of AES S-Box has been evaluated against fluctuations in the power circuitry of an FPGA. In [10], the authors evaluated the leakages from an AES implementation in the presence of dual-rail with pre-charge logic based countermeasures, caused by fault injections. In [11], an orthogonal approach of manipulating operational temperatures is used to mount practical attacks on RSA implementations for AVR micro-controllers like ATmega162. Along similar lines, in [12], the practicality of fault injections on AES through overclocking has been explored in the context of ASICs. Other works like [12]–[18] also explored the effects of similar fault injection techniques on FPGAs and similar targets, in the context of attacks on

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24499
www.ndss-symposium.org

cryptographic implementations. Likewise, [19] performs fault injections by placing the probe over the *die* (instead of system-bus) of a 32-bit micro-controller causing *instruction level faults*. However, considering the complexity of a SoC against a micro-controller (like ① high, scalable frequency ranges, ② pipeline optimizations, ③ advanced memory controller etc), direct reusability of such micro-controller based fault characteristics to SoCs becomes difficult. For want of conciseness, we defer the reader to a survey [20] on similar lines. Apart from breaking ciphers, FI attacks have been extensively used in other attacks, such as bypassing security checks in smartcards [21]. Likewise, in [22], laser faults are used to bypass verification of memory integrity. In [23], faults are used to bypass secure boot related verification checks.

Current FI Protections. In response to such attacks, several proposed countermeasures aim to harden either the algorithm or its target platform. However, the former is more lucrative for two reasons: ① it allows for more flexible use of the algorithm, since it does not depend only on a particular form of hardware, and ② it helps provide provable security guarantees. In the context of FI attacks, the majority of the countermeasures focus on a detection-based approach where the principle is to detect the presence of fault(s) via redundancy. Once a fault is detected, the output or the state of the process is mutated in such a way that the effective information is no longer useful for the adversary. Incidentally, the concept of *masking*, which is one the most prominent countermeasure for *Side Channel attacks*[1], has been used in the context of FI attacks as well [24], [25]. In masking, a secret (usually the cryptographic key) is divided into a number of *shares* at the level of cryptographic circuits [26]–[29]. Not all shares are used at the same point in time, and the shares have statistical mutual independence. The core idea is that an adversary now has to attack each share independently, in order to attack the underlying schemes. However, there have been fault attacks [30]–[32] even on masked cryptographic implementations.

A. Beyond faults on hardware: Stepping into the SoC world

In modern embedded systems market (more specifically in IoT ecosystem) other than FPGAs and ASICs, *Systems-on-Chip* (or SoCs) are being increasingly used. An SoC is a general-purpose computer capable of running a full-fledged operating system (OS) and several applications atop it. More importantly, unlike FPGAs, SoCs are not re-configurable, implying all cryptography is implemented in software (and not through hardware circuits). Due to the lack of reconfigurability and the generic micro-processing nature, the hardware-based countermeasures (on FPGAs) are not applicable to SoCs. However, several higher-end SoCs (like the Raspberry Pi[2] family) are inherently resistant to various FI attacks. The hardness of these devices against practical FI attacks can be attributed to ① the lack of an externally manipulable voltage/-clock interface preventing any voltage/clock glitch attacks, ② metal shield over the processor which, without depackaging, prevents successful injection of electromagnetic/laser faults into the processor, ③ access control policies of the operating system interfering with reliable exploitation vectors, ④ other processes running on the SoC making accuracy and success rate of the fault attack low, among others. This makes the study of fault attacks on SoCs interesting because, while on one hand, several known fault attack vectors do not work on SoCs, on the other hand, a richer attack surface (more than just cryptographic implementations) sprouts up.

State-of-the-art FI Attacks on SoCs. While SoCs have sufficient protection against classical FI attacks, they are not completely invincible. In the recent past, several new fault attack vectors have come up in the context of SoCs, focusing on both cryptographic and non-cryptographic software targets. All such attacks on SoCs can be characterized by the target of the fault (i.e. the architectural or algorithmic aspect of the victim software). One such target is the ① dynamic voltage-frequency scaling interface (DVFS). In [33], strategic use of undervolting caused timing violations in the Arithmetic Logic Unit (ALU), leading to faults breaking secure AES software implementation. Likewise, in [34], the authors used similar DVFS interfaces to inject faults through overclocking. Similarly, in [35], the same DVFS interface has been used for voltage manipulation (in contrast to frequency manipulation) to achieve successful faults. There are also attacks that, contrary to DVFS wherein software interfaces are used, target ② external, hardware-backed, manipulable injections which force erroneous computations. For instance, in [36], the authors resort to physically hooking upon the Serial Voltage Identification bus to manipulate voltages for similar objectives. In [37], a clock glitch is used to bypass RISC-V's physical memory protection. While in [38], voltage glitches are used to load malicious firmware that decrypts virtualized memory and fake attestations. Likewise, [39], [40] also use externally manipulable voltage glitches to attack SoCs. In [41], electromagnetic injections are used to fault an integer pointer and read secret data. There have also been successful laser fault injections [8], [22], [23], [42] targeting a variety of SoCs.

FI-resistant SoC Architecture. All these known FI attacks on SoCs already have strong countermeasures. To prevent exploitation of ① DVFS based fault attacks, the concerned software interfaces are put behind proper access control policies, denying the adversary any control over them. Similarly, use of low-cost Systems-On-Chip (SoCs) like Raspberry Pi (which lack any ② externally manipulable clock/voltage interfaces) instead of micro-controllers eliminates the possibility of glitch attacks. Finally, most modern SoCs have a metal casing over their processors, which makes attacks through electromagnetic/laser injections improbable without depackaging. Contrary to these attacks, there are memory fault attack like Rowhammer [43] that rely on dynamic Random Access Memory's (DRAM's) operational physics to inject permanent memory faults. However, Rowhammer styled fault attacks are also prevented on SoCs [44]–[46].

B. Contributions

Given the presence of such defences against fault attacks on SoCs, we ask a very fundamental question: *are there other architectural features which can be used for faults, for which no known defences are deployed yet*? In this work, to the best of our knowledge, we provide the first practical demonstration of one such feature: *system bus faults*. In particular, we provide the following contributions in this paper:

1) **Practical Demonstration of System Bus Fault Attack:** While prior works have shown simulations (e.g [47]), this

[1] A class of attacks that rely on passively gained information (like time taken for execution, cache access patterns specially in the case of SoCs) to leak secret data of the victim process

[2] Small single-board computers based off Broadcom processors.

work, to the best of our knowledge, provides the first practical demonstration of bus fault on an SoC. We provide a detailed characterization of faults in both the data bus and the address bus. For the data bus, at the time of execution of an instruction fetching data from memory, a precise electromagnetic injection is able to *change* the value of loaded data, without affecting any other register in the processor. Likewise, at the time of execution of an instruction involving a memory dereference, precise electromagnetic injections are able to change this memory address, causing memory access violations that the adversary exploits.

2) **Data Bus Faults to break symmetric-key ciphers:** Using data bus faults, we introduce reliable faults in lookup table (software-based) implementation of AES S-boxes on an SoC, making such an implementation also vulnerable to Differential Fault Analysis (DFA) [48]. This is in contrast to literature which relies on hardware circuit implementation of AES S-Box transformation to mount fault attacks.
3) **Address Bus Faults to break Post-Quantum Cryptographic schemes:** Using address bus faults, we provide an insight into how the system environment can make the masked implementations of ciphers (for instance, the secure NIST recommended post-quantum cryptographic algorithms) vulnerable to key recovery attacks. Such address bus faults, when combined with other factors in the system outside the control of the cryptographic algorithm's implementation, led to secret key leakage from all implementations *with a single fault injection*.
4) **Breaking ARM TrustZone Isolation for Trusted Execution Environment:** Finally, we use the novel bus faulting strategies to mount an attack on a popular Trusted Execution Environment (TEE) implementation for ARM TrustZone: Open Portable Trusted Execution Environment [49] as well as on its hardened implementation: MyTEE [50]. Using bus faults as well as loopholes in the GlobalPlatform API specification, we skip signature verification and install self-signed, malicious Trusted Applications in the secure world side of OP-TEE and MyTEE. This allows an adversary to execute self-signed code in the secure world of such Trusted Execution Environments (TEEs). Moreover, we validate the success of the attack across two different platforms, thereby establishing the portability of the attack vector. Through this finding, we contextualize the implications of bus faults into domains beyond cryptography.
5) **Universally Unique IDentifier confusion**: We uncover a design specification issue in the GlobalPlatform API specification that allows an adversary to install a malicious TA which can *masquerade* as any other innocent TA in the system, allowing *man-in-the-middle* attack.

C. Responsible disclosure

We notified the OP-TEE security team and Linaro about our findings, post which we entered into an embargo of 90 days wherein the fix was developed and tested by us for fault tolerance. The incident report for the same and the fix have been made public. The attack has been assigned CVE-2022-47549 under the category "Improper Verification of Cryptographic Signature" (CWE-347).

D. Organization

The paper is organized as follows. In Sec. II, we introduce the concept of Trusted Execution Environments (TEE). In Sec. III, we characterize faults on both aspects of the system bus: *data* bus and the *address* bus. Then, in Sec. IV, we use bus faults to mount an end-to-end attack on a commercial implementation of ARM TrustZone based on GlobalPlatform (GP) API specification: Open-Portable Trusted Execution Environment (OP-TEE). To consolidate portability of our attack, in Sec. V, we show the same attack works (with no modifications) to ① a new TEE named MyTEE, as well as ② on a different hardware platform. Finally, in Sec. VI, we describe the experimental details and then detail countermeasures in Sec. VIII which are currently deployed in production.

II. Background

A. Trusted Execution Environments

Trusted Execution Environments (TEEs) are hardware-backed solutions designed for security even in the presence of a compromised kernel. The core idea is to partition resources on an SoC into a trusted and an untrusted execution environment and shift critical executions (like cryptography) 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 trusted part is compromised. Since we consider TEE security in the context of embedded systems, we focus on ARM TrustZone as ARM chips dominate the majority of the embedded systems market. ARM TrustZone has been an integral part of ARM chip-sets since ARMv6 [3], including Cortex-A (for application grade systems like mobile devices, workstations etc) 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-a-Chip hardware/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). REE supports a complex software stack and can be prone to privilege escalation due to software bugs [51].

B. Introduction to OP-TEE

Our choice of TEE for ARM TrustZone is Open Portable Trusted Execution Environment (OP-TEE) [49]. While there are other TEEs too for embedded systems (like Samsung's mTower[4]), we chose OP-TEE for a number of reasons. First, to the best of our knowledge, OP-TEE is the only open-source TEE that conforms to a wide variety of SoCs [52], with detailed porting guidelines to add new devices [53]. Secondly, OP-TEE integration has been announced by commercial entities like Apertis [54] and iWave systems [55], indicating trust in its capabilities. Moreover, OP-TEE conforms to GlobalPlatform API specifications for TEEs [56], allowing easier articulation of the general impact of our attack vectors. Lastly, OP-TEE has been developed for a period of 9 years by major embedded systems market players, including STMicroelectronics/Linaro and is currently being maintained by the TrustedFirmware project (whose board members include ARM, Google, NXP, Infineon and so on).

[3]The current ARM version used in a majority of processors is ARMv8.
[4]https://github.com/Samsung/mTower

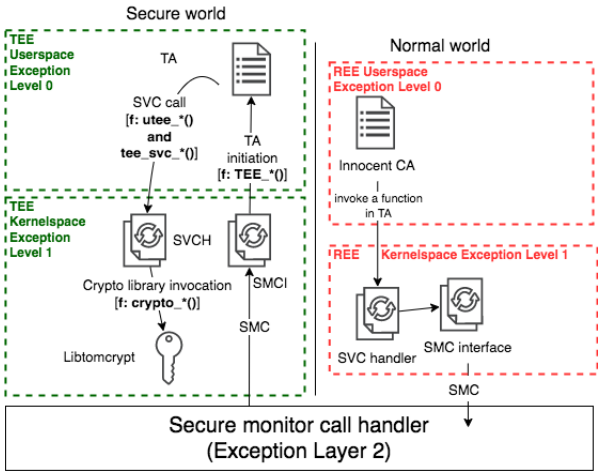


Fig. 1: OP-TEE Architecture. `SVCH` and `SMCI` denote Supervisor Call Handler and Secure Monitor Call Interfaces.

Intuition for the isolation. As referred in Fig. 1, the REE and TEE are divided into two layers: Exception levels (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, custom OP-TEE OS is used as the operating system. Exception level 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*.

III. Bus Faults: A Novel Fault Attack on SoCs

In this section, we introduce *bus faults* - faults injected *while* the system bus is operational, rather than *during* the computation in the processor *after* the operation of the system bus is already over. For completeness, let us first understand how the system bus interacts with the processor and memory. In all modern systems, memory is aligned in a hierarchy to balance the trade-off between cost and speed. The fastest memory (i.e. registers inside the processor) is costly and thus in short supply. Likewise, the cheapest memory is in abundance but extremely slow in operation. As a result, a lot of data exchange happens between the different levels of memory. Although the details vary with different Instruction Set Architectures (ISAs), in general, every ISA supports `load` and `store` instructions responsible for data transfer from the memory to registers and vice-versa. Whenever these instructions are executed, there is bidirectional transfer of data between the processor and memory via the system bus. The system bus has two parts: the *data* bus and the *address* bus. For every memory operation, the memory address to be accessed is placed onto the **address bus**. Likewise, the corresponding data is placed onto the **data bus**. While most of the countermeasures have been focused on either protecting the processor or the memory module, the system bus has not been taken into consideration. As both data (also instruction) and address travel through the system bus, it provides an additional attack surface for FI attacks. We show that a precise fault injection during the execution of `store` and `load` instructions causes corruption of contents of both the address and the data buses. This corruption percolates into execution since the contents of address/data buses drive several operations in the processor and memory. By targeting system bus instead of directly faulting the processor/memory, our bus faults bypass countermeasures preventing processor faults, as well as avoid persistent memory faults (like Rowhammer [44]) that can cause permanent data corruption.

We note that this work significantly differs from [57]–[59] in the sense that practical fault attacks on buses (like CAN bus) are not directly portable to *system* buses in high-frequency SoCs. First, such works use *invasive* fault attacks, as opposed to our bus faults (which do not require any irreversible modifications to the victim device). Secondly, the type of faults injected in [57], [58] (like shorting micro-controller pins, disconnecting Vcc/GND pins etc) are completely different to the kinds of faults we explore in this work. Finally, our bus faults differ from the invasive clock glitches in [59], in which an external FPGA is used to insert clock glitches while the hypervisor is issuing read/write cycles to memory. The inserted glitches *skip* memory cycles, allowing an adversary to access page tables that should have been de-allocated. In contrast, the goal of injecting bus faults in our work is not to *skip* read/write cycles from hitting memory, but rather to *change* memory contents involved in the cycle. Such manipulated memory contents are henceforth used in critical decisions (eg. TEE signature verification as discussed in Sec. IV), thereby having more drastic effects. As such, simply skipping memory cycles as in [59] are unsuitable in our context, since skipping memory cycles will have lesser control on the memory content, making the attack infeasible in a practical setting. Through our bus faults, on the other hand, we are able to *modify* the content of a memory cycle, thereby making our attack practically viable.

A. Characterizing Data Bus Faults

Before characterizing data bus faults, we first describe the operation of the data bus during execution of a `load` instruction. Generically, a `load` instruction is characterized as `load <destination register>, <memory address>` such that it loads the data at address `<memory address>` to the register at `<destination register>`. To do this, the memory unit places the requested data onto the data bus, which then stores it into `<destination register>`, for further processing by other instructions.

Data Bus Faults to enable DFA on AES: We show that during the transition of data from memory to the processor, practical fault injections on the system bus are possible which can lead to corruption of the data. To demonstrate the power of data bus faults, we show how such faults allow the extension of DFA attacks into table-based software implementation of AES S-Boxes[5] (as opposed to prior attacks on circuit implementations, as detailed in Sec. I). We now note prior works on attacking S-Boxes. Works like [31] assume a circuit-based

[5]We note that in desktop and server systems that use x86 ISA, AES is implemented using special instructions, collectively called `AESNI`. However, such specialized instructions are not present in SoCs and embedded devices. Moreover, most of the block ciphers use table-based or array-based implementation of S-Box for software implementation as it is relatively faster than actually performing S-Box operation (as matrix multiplication and vector addition) for each plaintext byte.

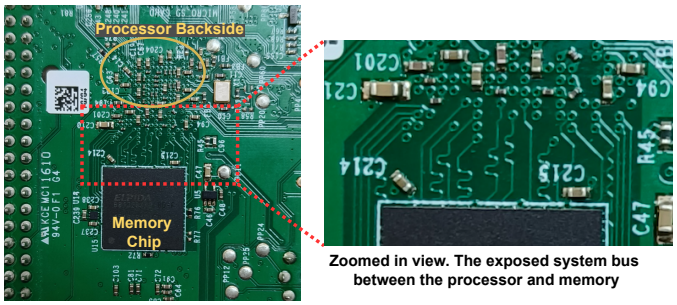


Fig. 2: The backside of a generic Raspberry Pi 3 board showcasing the memory chip, the backside view of the processor's position, and the system bus interconnecting the two

implementation of such S-Boxes wherein faulting inputs to individual gates is possible. Similarly, works like [60] do not consider circuit implementations, but rather use faults to manipulate the memory itself. Likewise, works like [61] use undervolting interfaces to inject fault while AES computation is in progress. However, our approach is orthogonal. We claim that, in a table-based implementation of an AES S-Box, when the S-Box table in memory is being accessed, a powerful electromagnetic pulse over the position where the system bus is printed on the Printed Circuit Board (PCB) causes faults in S-Box substitutions. Such positioning of the electromagnetic pulse injection also ensures that the injected faults do not cause disturbances in the operation of the processor or the memory[6].

Locating Optimal Fault Injection Point: For fault injection, we use Electromagnetic (EM) pulses as it provides a means for *non-invasive* attacks (i.e. without making any physical alteration to the device). To find a correct position for electromagnetic injections, we take a look at the RPi3 Printed Circuit Board (PCB) as shown in Fig. 2 (RPi4 PCB is approximately similar to RPi3's PCB, hence positioning on RPi4 is similar as that on RPi3). The memory chip resides in the middle of the PCB, while the processor resides a few centimetres towards the top (as per Fig. 2). A dense network of bus interconnects runs between the two. It is worth mentioning that in most SoCs, similar PCB prints can be found where the processor and the memory module are situated on the PCB, with the *exposed* system bus providing the interconnection. The test-bed schematic is given in Fig. 3. We note that since we do not target the processor, there is no threat of the injected fault *spilling* into neighboring regions (like neighboring registers during fault injections in the processor). This makes it easier to find a suitable fault injection probe position. Further details on the fault injection hardware, probe position, and detailed description of the attack setup will be discussed in Sec. VI.

Differential Fault Analysis on AES: In the classic (Differential Fault Analysis) DFA setting [48], the circuit implementation of AES is faulted at the 8^{th} round to obtain faulty ciphertext. More specifically, a single bit-fault in the register during the operation of 8^{th} round can leak one byte of the key by applying the well-known differential fault analysis technique using both *faulty* and *correct* ciphertexts[7]. For our experiment, we use a generic C-based

[6]In the later part of the paper, we experimentally verify that our fault attack does not corrupt the original memory contents.

[7]Implementation available at https://github.com/Daeinar/dfa-aes

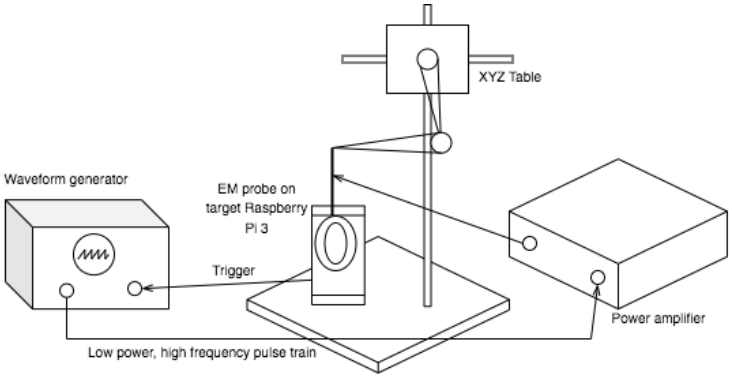


Fig. 3: A schematic of the fault test-bed. Once the waveform generator is triggered, it emits a pulse train comprising low power, high frequency square pulses. This pulse train is amplified by a power amplifier and then injected into the victim device by an electromagnetic fault injection probe.

TABLE I: Parameter values allowing characterization of data bus faults on AES.

Parameter	Value
Signal generator delay	2 microseconds
Pulse generator amplitude	-7.79 dbm
Pulse generator burst length	20 pulses
Amplifier output frequency	400 MHz
Probe position	Fig. 9
Fault position	Execution of `load` instruction for the first S-Box in the 8^{th} round

TABLE II: Results of the fault injection. Note the faulted byte (in **red**) in 47-th injection. To show the preciseness, this table captures the intermediate state of the cipher just after the 8-th round, before the injected fault pollutes the entire state in subsequent rounds.

Iteration	Plaintext	Ciphertext
1	0x00112233445566778899aabbccddeeff	0x8ea2b7ca516745bfeafc49904b496089
2	0x00112233445566778899aabbccddeeff	0x8ea2b7ca516745bfeafc49904b496089
3	0x00112233445566778899aabbccddeeff	0x8ea2b7ca516745bfeafc49904b496089
...	...	...
47	0x00112233445566778899aabbccddeeff	0x**2**ea2b7ca516745bfeafc49904b496089
...	...	...
100	0x00112233445566778899aabbccddeeff	0x8ea2b7ca516745bfeafc49904b496089

implementation[8] on 16 byte input plaintexts wherein the S-Box substitutions are carried out by table look-ups into a table comprising of 256 8-bit integers[9]. In every round, 16 substitutions are performed, and every substitution requires one memory access to the AES S-Box table stored in memory. By extension, this implies that every substitution requires the execution of a `load <destination register>, <memory address>` instruction. During the execution of this instruction, we were able to pollute S-Box substitutions. The parameter values to obtain faults on this operation are detailed in Tab. I. A detailed explanation of the parameters can be found in Sec. VI.

By repeating fault injection operation enough times, we were able to induce bit-flip faults in round 8. The result of our attack is shown in Tab. II. As shown, we observed that in the 47^{th} injection in our batch of experiments, the first nibble suffered bit-flips. Equipped with such faults in round 8 and correct ciphertext pairs on the same plaintext, we successfully perform a full-key retrieval attack on AES-128 using the DFA

[8]Implementation available at https://github.com/dhuertas/AES

[9]Table or array based AES implementations are common for SoCs.

technique [48].

B. Characterizing Address Bus Faults

In the previous subsection, we demonstrated practical FI attacks on the data bus by utilizing the `load` instructions that carry data from memory to the processor. We now turn our attention on the execution of `store` instructions. Generically, a `store` instruction is characterized as `store <destination memory address>, <source register>` such that the data in `<source register>` is stored in the memory addressed by `<destination memory address>`. For a `store` instruction, the processor places `<destination memory address>` onto the *address bus* and the contents of `<source register>` onto the *data bus*, and issues control signals indicating transfer of data from the data bus to the memory referenced by the address bus.

While it is possible to generate data bus faults (c.f. Sec. III-A) in this context as well, we focus on another possibility. A high-end SoC is able to run a full-fledged OS, atop which several user applications reside. As such, all benefits of a well-designed kernel[10] are inherently available to the user applications. One such benefit is memory isolation and protection. The OS guarantees isolation between processes by virtually partitioning the entire system's memory such that no two processes can interact illicitly without the intervention of the OS. Additionally, *memory access violation* signals are raised by the OS when one process tries to access memory owned by another. This presents us with a unique attack opportunity: forcing memory access violations through faulting the address placed onto the address bus, during the execution of `store` instructions.

Memory access violations in isolation are inherently unexploitable even when they occur. However, since our experimental platform runs an OS, there are several other (potentially harmless) utilities provided by the OS which when combined with our address bus faults, lead to disastrous consequences like leaking cryptographic keys. Characterization of such faults reinforces the belief that the complexity of the execution environment of a program in the context of SoCs introduces novel attack vectors, which are beyond the control of the program itself. As a proof-of-concept, we use address bus faults to force raise `SIGSEGV`[11] signals during executions of post-quantum cryptographic (PQC) algorithms, and utilize other aspects of the system (outside the control of the PQC algorithms) to recover secret keys.

Dilithium: Dilithium [62] is a post-quantum signature scheme and one of the selected algorithms in NIST's process of standardizing post-quantum cryptographic implementations. The reference implementation[12] of Dilithium from [63] provides the core functionality needed from a digital signature scheme, and needs a wrapper (c.f. Listing 1) to invoke the core functions within the library.

```
1 void dilithium_wrapper(const uint_8* message){
2     uint8_t *pk = (uint8_t*)malloc(1312); // public key
3     uint8_t *sk = (uint8_t*)malloc(2528); // secret key
4     uint64_t signature_len = 2420;
5     uint8_t *signature = (uint_8*)malloc(signature_len); // signature
6     PQCLEAN_DILITHIUM2_CLEAN_crypto_sign_keypair(pk, sk);
7     PQCLEAN_DILITHIUM2_CLEAN_crypto_sign(signature, &signature_len, message, 30, sk)
          ;
8 }
```

Listing 1: Wrapper code invoking Dilithium's signing functionality.

TABLE III: Parameter values allowing characterization of address bus faults on PQC algorithms. Note the extended burst length of 100 pulses, as opposed to 20 in Tab. II.

Parameter	Value
Signal generator delay	2 microseconds
Pulse generator amplitude	-8.19 dbm
Pulse generator burst length	100 pulses
Amplifier output frequency	400 MHz
Probe position	Fig. 9
Fault position	Execution of `PQCLEAN_DILITHIUM2_CLEAN_crypto_sign`

As part of the function `PQCLEAN_DILITHIUM2_CLEAN_crypto_sign`, the secret key is loaded onto the stack[13], and is used as part of the signing process. This is precisely when we inject an EM pulse (for a longer period of time compared to the one in Sec. III-A) that causes the target `store` instruction to raise a `SIGSEGV` signal. This happens because the injected fault changes the contents of the address bus to point to a memory location inaccessible by the process, hence raising a `SIGSEGV` (memory access violation) signal. We note that a `SIGSEGV` is inherently useless for an adversary. But under default system configuration (outside the control of the process running Dilithium), a `SIGSEGV` is accompanied by a coredump whose analysis reveals the contents of the in-memory stack (where the secret key resides *in plaintext form*), thereby converting a seemingly harmless address bus fault into an attack vector that *leaks the entire secret key with a single fault*. Based on the parameters reported in Tab. III, we raise `SIGSEGV` signals in the execution of Dilithium, causing a `SIGSEGV` in `PQCLEAN_DILITHIUM2_CLEAN_polyt0_unpack` (we refer to Appendix, Fig. 7 for a sample backtrace). From the backtrace in the coredump, the secret key is easily recoverable (argument `sk` in `PQCLEAN_DILITHIUM2_CLEAN_crypto_sign`). To show a proof-of-concept attack, here we assume availability of coredumps to an unprivileged attacker. In Sec. IV, we relax this assumption in context of a real-world TEE.

The consequence of the aforementioned attack is that an adversary without the knowledge of the underlying cryptographic algorithm can essentially leak the entire secret key by a single fault by carefully faulting the operations at the correct time. In short, a `SIGSEGV` signal issued, after the secret key is loaded onto the stack, is sufficient for the attack to succeed. This brings into question the viability of deploying PQC schemes on IoT edge devices where the presence of a physical adversary constitutes a valid threat model. In order to validate our claims, we perform similar experiments on other PQC candidates, which are touted to be protected against classical FI attacks by the use of masking.

First-order masked implementation of SABER: We now experimentally validate whether the address bus based fault attack is also applicable in the context of masked imple-

[10] Software that sits at the core of OS and has the highest privilege over the hardware.

[11] An operating system signal noting the occurrence of a memory access violation, causing the offending process to be terminated.

[12] Implementation available at https://github.com/PQClean/PQClean

[13] Part of a process' memory map which stores all statically allocated data, function frames, and other book-keeping data necessary for program execution.

mentations. As a proof-of-concept, we take the example of a first-order masked implementation of SABER from [64][14]. Listing 2 shows how all shares of SABER being encapsulated in a single structure. By injecting an address bus fault in the decapsulation function `crypto_kem_dec_masked`, the adversary has access to the coredump wherein the structure `sk_masked_s` is loaded onto the stack. Since all shares are encapsulated within the same C structure `sk_masked_s`, a single fault once the structure is loaded onto the stack is sufficient to leak all secret shares.

```
typedef struct
{
    uint16_t s[SABER_SHARES][SABER_L][SABER_N]; // shared
    uint8_t pk[SABER_INDCPA_PUBLICKEYBYTES];
    uint8_t hpk[SABER_SHARES][32]; // shared
    uint8_t z[SABER_KEYBYTES];
} sk_masked_s;
```

Listing 2: An encapsulation of SABER shares.

One must note that this attack succeeds because the address bus faults do not target the algorithm. The attack rather relies upon the fact that the implementation keeps all shares encapsulated in a single structure, which is loaded onto the stack at once. An address bus fault in our context causes the memory image of the stack to be dumped as a coredump file, from where the shares are retrievable in simple plain text-form.

First-order masked implementation of KYBER

Similar to SABER, KYBER also has a first-order masked implementation in [65] [15]. In this implementation as well, all secret shares are encapsulated within a single structure (c.f. Listing 3). At the time of execution of the decapsulation function `crypto_kem_dec_masked`, the structure `masked_sk` is loaded onto the stack and the decapsulation proceeds. Like the attack on SABER, a single address bus fault during the execution of this function causes all the shares in `masked_sk` to be leaked.

```
typedef struct {
    masked_polyvec indcpa_sk;
    uint8_t pk[KYBER_INDCPA_PUBLICKEYBYTES];
    uint8_t hpk[KYBER_PUBLICKEYBYTES];
    masked_u8_symbytes z;
} masked_sk;
```

Listing 3: An encapsulation of KYBER shares.

C. One Fault to Break Them All: Implications for post-quantum cryptographic algorithms

Fault Attacks have been used extensively in the literature to compromise mathematically robust cryptographic schemes. Many block ciphers as well as public key algorithms, including AES and RSA, have been subjected to wide analysis in the context of fault attacks. Towing along similar lines, PQC candidates have been analysed with respect to fault attacks to ensure secure and wide deployment after NIST standardisation. One such interesting work has been done in [66], where the authors require $6,500$ faults for KYBER's smallest parameter set. It is interesting to note that the majority of such attempts target the post-quantum algorithms and their designs, and do not necessarily consider the target platform on which these schemes are supposed to work. In this work, we put forth a different perspective with the demonstration of practical bus faults. We show that it is not sufficient to design leakage-resistant designs. The first-order masked versions of SABER and KYBER are theoretically resistant to fault injections. However, due to some design considerations (using the same structure to store all the components of the key), the entire secret key can be made to leak on appropriate devices. Therefore, this work, in essence, begets a new aspect that should be considered during designing as well as implementation - *what effect the environment where the post-quantum algorithm's reference implementation is running has on the security of these schemes*. Due to a combination of factors, which are outside the reference implementation's purview, a *single fault* allows leakage of the entire secret key, without the knowledge of the underlying algorithm. In other words, by not analysing the implementation platform on which the post-quantum algorithm is executing, we have inadvertently *expanded* the trust boundary of the algorithm to include other aspects of the system the algorithm has no direct control over. Furthermore, as is the case with address bus faults and dumped cores, the post-quantum algorithm itself is oblivious to this extension of its trust boundary, since it has no way to *verify* the integrity of other system aspects whence an adversary can influence generation of coredumps. We believe that such unintended *expansions in trust boundaries* of post-quantum algorithms should be a prime concern while implementing an otherwise theoretically secure algorithm.

D. Comparing Bus faults with generic processor or memory faults

In this section, we provide justification and empirical arguments on how our bus faults are significantly different from generic faults in processor/memory.

Why is it not a generic memory fault attack?: We note that in [60], laser faults were injected in memory leading to persistent corruption of memory. Our attack is fundamentally different from [60] for two reasons: ① our probe positioning does not influence the memory chip, and ② we observed faulty ciphertext due to fault injection, but subsequently observed correct ciphertexts as soon as the EM probe was removed. Point ② emphasises the transient nature of our fault attack, thereby implying that the S-Box itself in memory was not manipulated as we got correct ciphertext computations *after* the faulty ciphertext was generated. Moreover, in our experiments, memory corruption with electromagnetic fault injection had a distinct behaviour. We observed that when the probe is placed covering the entirety of the memory chip, the system *freezes* and requires a reboot. This is expected, as injecting powerful electromagnetic pulses into the memory may corrupt more than just AES S-Box tables, causing the system to freeze.

Why is it not a generic processor fault attack?: This attack is also significantly different from prior attacks on the processor wherein the fault is injected while the computation is underway. This is because ① we did not depackage the processor casing, ② the processor is beyond the field of influence of our probe, and ③ in our experiments (with the probe directly placed over the packaged processor), we were unable to reproduce classic processor faults like instruction skips, thereby reinforcing the belief that the packaged processor is not susceptible to electromagnetic manipulations.

E. Bus Fault Characteristics and Success Rates

We now focus on characterizing the amount of control an adversary has on both data and address bus faults. To characterize data bus faults, we use the parameter set detailed

[14]Implementation available at https://github.com/KULeuven-COSIC/SABER-masking

[15]Implementation available at https://github.com/masked-kyber-m4/mkm4

in Tab. I. Similarly, for characterization of address bus faults, we use the parameter set in Tab. III. For both parameter sets, we repeat 10000 EM fault injections on `load`/`store` instructions and report fault characteristics.

For the experiment concerning data bus faults, we define three sub-classes: ① the data was not corrupted, ② the data was corrupted but not changed to `0x0`, and ③ the data was corrupted to `0x0`. Among 10000 injections, 62% of the injections were a combination of cases ② and ③, implying a data bus fault failure rate (i.e. case ①) of 38%. Out of the successful fault injections, we observed case ③ 27% of the times. On the other hand, upon analysing the corrupted non-zero data values obtained in case ②, we observed that all data corruptions happened in the upper 16-bits of the 32-bit wide data values on RPi3/RPi4. Moreover, in case ②, we did not observe a significant difference between the probabilities of $1 \rightarrow 0$ and $0 \rightarrow 1$ bit flips.

For the experiment concerning address bus faults, we again define three sub-classes: ① no corruption in the address (i.e. no `SEGFAULT`), ② corrupted address is invalid (i.e. `SEGFAULT`), and ③ corrupted address is valid. The last case arises if the corrupted address corresponds to a memory location within the confines of memory range of the stack or the heap in the process memory map. Among 10000 injections, we observed 31% successful address bus faults. Further analysis shows that, among successful injections, *no* case ③ is observed. This observation is in line with similar observations for data bus faults- *lower* 16*-bits of the* 32*-bit wide addresses are extremely difficult to fault*. Note that faulting *upper* 16-bits of the address almost always causes the resulting address to point *outside* the address ranges for stack/heap, resulting in invalid addresses.

We finally state that an in-the-wild adversary can choose the pulse generator burst length to assert control over whether the injected fault is a data bus or an address bus fault. From Tab. I and Tab. III, one can observe that a *shorter* length of around 20 pulses correspond to data bus faults, while a *longer* length of around 100 pulses correspond to address bus faults.

Takeaway. Characterization of bus faults into sub-classes establishes one useful insight: data bus faults are capable of *register sweeping*, where data bus faults change an entire 32-bit register from a non-zero value to `0x0`.

In this section, we introduced novel system bus fault and developed proof-of-concept attacks using data bus and address bus faults in separate scenarios. Now since we have separate characterizations of data bus and address bus faults as well as our main takeaway of *register sweeping*, we combine the two to mount an end-to-end attack on a practical system with a sizable market share in the IoT market. We note that a *single* successful register sweeping fault is sufficient to compromise ARM TrustZone. This is possible in around 40 independent injections as per the fault characterization done in this section.

IV. End-to-end attack on ARM TrustZone

We now show a use-case wherein bus faults can be used to install malicious Trusted Applications (TA) in the secure world side of a Trusted Execution Environment (TEE) for ARM TrustZone.

A. Comparison with previous works

In this section, we articulate concrete objectives for the adversary as well as how prior literature falls short of achieving the mentioned objectives. This helps emphasize the merits of bus faults in the context of SoCs. The main objective of the adversary is to install a *malicious TA* that is self-signed by the adversary into the secure-world. Ideally, all TAs must be signed by the Original Equipment Manufacturer (OEM), implying installation of any malicious TA essentially implies a complete breakdown of trust guarantees provided by ARM TrustZone. In addition to this, we also articulate the following objectives an adversary needs to consider.

- **G1**. The entire attack must be *online*. At no point in time must the adversary take the device offline. This requirement ensures that the attack evades remote monitoring systems like SCADA (Supervisory Control And Data Acquisition systems), if any [67], [68].
- **G2**. Attack without physical modifications (like chip decapsulation) to the device (i.e. an *non-invasive* attack), preventing future detection of the attack. This goal ensures no physical trace of the attack, evading post-attack hardware forensics [41].
- **G3**. Attack in a *reasonable* time-frame of 0.5 – 2 hour window [41]. This encapsulates two ideas: ① how long does an adversary have physical access to the victim IoT device (depending on its physical environment), and ② how long does it take to mount the attack.

We now discuss how other fault injection mechanisms in literature fail in our context, which also helps to emphasize the difference of bus faults compared to these techniques. A major attack point on Trusted Execution Environments is the **secure-boot**, which is a set of mechanisms that establish the foundational trust boundaries of the system when it boots up. Prior works like [22], [23], [69] use both non-invasive and invasive techniques to bypass secure-boot and install adversarial controlled applications inside the TEE. However, attacking *secure-boot* violates **G1**. Even if we assume an adversary with 100% fault reproducibility, it requires at least 1 reboot of the system, thereby voiding **G1**. This is because *secure-boot* can only be attacked when the device is booting up. Moreover, the userspace TAs are loaded in memory when they are required, long after the system boot is completed.

Likewise another class of works like [33]–[35] rely on an exposed dynamic voltage-frequency scaling (DVFS) interface, usually in perspectives of client workstations or server machines. However, in context of SoCs, there are two ways to dynamically control voltage/frequency: ① through Linux's scaling governors [70], and ② through `config.txt` that SoCs like RPi3/RPi4 initialize from during boot time [71]. Since ① no longer allows software-based overclocking post similar attacks [34] and ② is not editable from OP-TEE's stripped down Linux kernel in the normal world, remote fault injections like [33], [34] are not possible in our context. Note that one could argue that ② can be edited by removing RPi3's (or RPi4's) SD card (memory card housing the OS) and editing `config.txt` offline. However, that would void goal **G1** as the device needs to be taken offline to remove the SD card.

On similar lines, works like [13], [37], [39], [40], [72], [73] allow an adversary to attach a *second* adversarial controlled device to the victim device, where the second device controls

the inputs to the clock/voltage inputs of the victim device. Such well timed glitches can introduce precise faults. However, such attack vectors do not apply to our setting because of RPi3/RPi4's lack of an external clock/voltage interface. Although there have been works to this end [74], [75], it is a clear violation of **G2** due to the need of an external interface.

Finally, in addition to running on low-end devices like FGPAs and using external voltage/clock glitches, works like [37] assume the ability of an attacker to introduce code-based *triggers*, signalling the start of an execution of interest where the fault must be injected. Although an acceptable fault model in academic settings, it does not agree with our goal **G1** (which we believe is more relevant to practical situations). 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, and a more potent attack would ideally require no modifications to the victim code. Hence, our attack cannot rely upon such code-based triggers, and needs alternative triggering mechanisms, which we discuss next.

B. Choice of Raspberry Pi as target platform

For our experiments, we chose target devices as Raspberry Pi 3 and Raspberry Pi 4[16] boards for a number of reasons. First, except for a few works like [69], [76], [77], not much focus has been given to electromagnetic fault injections on *high-frequency* SoCs. Secondly, the RPi3/RPi4 is an SoC with wide adaptability [78], [79], making it suitable to be used widely in IoT ecosystem for applications that demand relatively higher computation capabilities. Moreover, easy accessibility of such boards makes the reproducibility of fault attacks simpler. Thirdly, RPi3/RPi4 boards pose challenges in areas where its simpler counterparts (like micro-controllers and FPGAs) offer numerous surfaces with respect to physical attacks. One such challenge area is the absence of externally manipulable clock/voltage interfaces, thereby eliminating a wide body of research attacking such interfaces. These factors allow for a more thorough analysis of the benefits of bus faults against other faulting strategies.

We note that the official OP-TEE port on RPi3 Model B misses some key aspects [52] of a generic TEE. Although the RPi3 processor provides ARM TrustZone exception states, the mechanisms and hardware required to implement secure boot, memory, peripherals or other secure functions are not available. We however note that our attack does not target secure boot, memory, peripherals etc. We target error code definitions to install rogue TAs, which are defined by GlobalPlatform specification and are platform-agnostic. We quote OP-TEE developers during our disclosure: "*Even though the Raspberry Pi 3 device only should be used for educational purposes, we believe that this attack is something that could be done on other devices as well and therefore we think it's now justifiable to introduce the software mitigation patterns.*"

[16]Hereafter abbreviated as RPi3 and RPi4 respectively.

C. Bus Faults to Break ARM TrustZone

We now establish the adversarial assumptions and describe how to install a self-signed, malicious TA onto the secure world of ARM TrustZone.

Attacker Threat Model: Before describing the attacks, we establish goals and assumptions for the adversary. The adversarial objective is two fold: ❶ installing a malicious TA by attacking the TEE, and ❷ break symmetrically encrypted communication channels between CA/TA by attacking the REE (only if third-party extensions offering advanced protections are present along with default OP-TEE build). To achieve these objectives, we assume an adversary that ① has physical access to the device, ② can execute code on REE side, and ③ whose privileges are confined to Exception Level 0 (EL0) in the normal world side as depicted in Fig. 1.

We note that achieving objective ❶ is sufficient to compromise a default OP-TEE configuration. However, for a holistic evaluation of production-level systems, we also assume state-of-the-art third party defences like [80], [81] in addition to default OP-TEE configuration, encapsulated by objective ❷. We note that our adversarial assumptions are in line with the threat model in [80], [81] which guarantee secure key management services in presence of a compromised REE. Our adversary, with privileges confined to Exception Level 0 (EL0) on REE, injects address bus faults on CAs running in REE side and leaks *symmetric* encryption keys through coredumps. Adversarial access to such coredumps is enabled by default in OP-TEE. Moreover, even if coredumps are disabled, it is within our adversary assumptions that the adversary can re-enable coredumps (since both TEE and defences like [80], [81] guarantee security even from a compromised REE EL0).

Installing a malicious TA through data bus faults: We must first decide *where* in the TEE's execution the attack must be mounted. Since we cannot target *secure-boot*, we target the loading of userspace TAs post boot-up. All TAs in the TEE are signed by the OEM's private key (which is not present on the device post-deployment) and signature verification happens every time a TA is loaded. Installing a malicious TA without fault injection then becomes the equivalent of forging digital signatures. With fault injection, one logical attack strategy is to use an *instruction skip* to bypass this signature verification. However, instruction skips are not viable because of the presence of control-flow integrity checks [82], [83]. Consequently, we use data bus faults and exploit the programming convention of using non-zero integers to represent errors and a `0x0` to represent success. We use data bus faults to *convert* a non-zero return value of a function to `0x0`, which then tricks TA signature verification to load a malicious TA. Once we know *where* to attack, we decide *how* to carry out the attack. We cannot use DVFS style fault techniques nor can we use clock/voltage glitches. Hence, we choose to use electromagnetic fault injections in this work. We did not choose laser based fault injections as that requires chip depackaging voiding goal **G2**.

We now explain in detail how attack works. OP-TEE follows an offline signing and online verification process. The underlying assumption is that the root-of-trust lies with the OEM, which itself signs each TA with its private key [84], 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 4 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
void load_TA(...){
    // some code here
    TEE_Result res = verify_signature(...)
    if(res != TEE_SUCCESS)
        // abort execution
    // some more code here
}
```

Listing 4: OP-TEE TA signature verification process

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

- **Observation 1**. Return code convention in Linux aids data bus faults.

From Listing 4 (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 non-zero values, `TEE_SUCCESS` was given a value `0x0`. This is common as almost all Linux-based function calls conventionally use a 0 to denote success, and other integers denote several erroneous operations. We exploit this convention for the attack. Listing 5 shows the aarch64/ARM64 disassembly of Listing 4. The point of interest is one `mov` instruction which loads the return value of `verify_signature` from the process stack memory into the register `w0`. We attack this operation by injecting an EM pulse train while `mov` is executed, leading to a *data bus fault* and pollution of `w0` value. In our experiments, we were able to cause *data bus faults* that cleared the value of `w0` to `0x0`. This change then forces the branch instruction `cbnz` to take a non-intended branch, thereby loading the malicious TA.

```
.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 5: 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 data bus faults on these instructions, `w0` can be cleared out to `0x0`, thereby bypassing the otherwise error state jump that `cbnz` would have taken.

- **Observation 2**. Using side-channel power consumption as non-invasive triggers.

One question still remains: *how to reliably know when `mov` is executed (i.e. when to trigger the fault injection)?* Attempting to attack in a non-invasive setting means we cannot use code-based triggers. We rather rely on the open-source nature of OP-TEE and side-channel power trace acquisitions to generate triggers, not requiring any modifications to victim code. This approach is one variant of a more generic technique: using the *behavior* of the target system itself as a trigger. For example, [38] uses the size of bus traffic as a trigger for voltage glitches. Similarly, [85] uses pattern matching on normal device power signals to trigger fault injections. It is worth mentioning that we also considered using OP-TEE's Universal Asynchronous Receiver/Transmitter communication protocol (UART) as means of triggering, but the communication's internal buffering latencies made synchronization hard.

Now we explain how prior works (c.f. Sec IV-A) achieve reliable code-based triggers for similar attacks, and point out problems with those approaches in our case. For usual code-based triggers, just before the security critical operation of interest begins, a signal is generated to an attacker-controlled device, which in turn starts injecting faults. Concretely, in the case of RPi3/RPi4 running Raspbian OS, one possible way to create such a code-based trigger is given in Listing 6. Just before the operation of interest starts, the attacker-induced code could send 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 6: Code-based trigger mechanism for Raspberry Pi

However, there are two problems with this approach. The generic problem is that in our work, we consider code-based triggers as *invasive* triggering mechanisms. 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 most practical settings. Moreover, this approach needs modifications to the kernel as well as kernel recompilation, which is against our assumptions on adversarial capabilities. Additionally, this fails our initial goal **G2** of launching the attack without taking the device offline [17]. And a more OP-TEE specific problem is that OP-TEE's normal world is a stripped-down Linux kernel exposing a BusyBox [18] 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.

Both these problems together imply a *non-invasive* triggering mechanism is required to temporally localise faults. Therefore, we use analysis of power traces as a triggering mechanism (we defer to Appendix A, Fig. 8 for the actual power traces we used). 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 (like multiplications) take more power than others, leading to a bias in acquired power traces, through which adversaries break cryptographic primitives [86], [87]. For our purposes, we use power based side-channel traces of multiplications for triggering fault injection. Multiplications are computation heavy operations, requiring higher power consumption than other operations. And multiplications are

[17] Kernel recompilation would require taking the device offline.

[18] BusyBox is a software suite providing Unix utilities as an executable file.

heavily used in OP-TEE's RSA based signature verification. 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 [88], and ② the heavy use of multiplications in OP-TEE's RSA signature verification process [89], [90]. From the combination of ① and ②, along with the fact that ③ a multiplication-heavy signature verification happens just before our point of interest (c.f. Listing 4), we have a non-invasive triggering mechanism to denote when our signal generators should begin sending EM pulses. The result of this attack is a self-signed adversarial controlled TA getting installed in the secure world of OP-TEE, thereby violating the security guarantees of ARM TrustZone, thereby completing objective ❶ for the adversary.

Stealing encryption keys through address bus faults: With regards to objective ❷, the default OP-TEE builds do not provide any security to the communication channel between a CA and TA. This raises the possibility of man-in-the-middle attacks on this unencrypted channel, as demonstrated in [91]. As such, third-party extensions like *SeCReT* [80], [81] have been proposed, which provide key management services to CAs and TAs, allowing to encrypt the communication between them. Briefly, upon assignment of a symmetric session key between a CA and a TA, *SeCReT* focuses on preventing a compromised REE from extracting this key from the otherwise vulnerable CA[19]. To do so, *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 and prevents an adversary from creating a *memory snapshot* (by attaching a debugger for example) [80]. As such, even after successfully installing a malicious TA through data bus faults, we still need to compromise *SeCReT*'s defences in order to decrypt other benign TAs' communications.

However, our address bus faults are able to bypass *SeCReT*'s defences because our faults *force the CA's access to its own memory pages to be invalid/faulty*, and thus cause no violation of *SeCReT*'s threat model. In other words, *SeCReT* trusts the CA owning the symmetric session key to behave innocently when handling its own key, while *SeCReT* guards malicious accesses from other agents (including debuggers to trace the victim) in the system. However, because of the address bus faults, we are able to force the CA itself to incur faults. This causes successful creation of coredumps with the symmetric session key (shared between CA/TA) accessible to an adversary. From the coredump, the session key is extracted using the same backtrace analysis as detailed in Sec. III-B.

We note here that generation of coredumps on the compromised REE side is an acceptable threat model, and is enabled on default OP-TEE build configuration. *SeCReT*'s defence mechanism is not to disable coredumps but to *prevent* any other process (including debuggers which can trace the victim and extract the keys) from attaching to a CA's memory page holding symmetric keys and creating such coredumps. As such, we bypass this defence by using the CA itself to access the concerned memory page and then inject address bus faults to force erroneous behaviour in an otherwise benign CA. Subsequently generated coredumps are stored on the normal world (not on the trusted world) and are accessible to an adversary. Finally, since the design decision of *SeCReT* is to support *symmetrically* encrypted communication channels between normal world and trusted world, compromising the session key on the normal world side is sufficient to breakdown security in both worlds. We refer to Fig. 5 in the Appendix for a pictorial depiction. We further emphasize that a successful accomplishment of objective ❷ does not necessarily imply fulfilment of objective ❶. The two objectives are independent and can be performed in any order. In our case, we perform ❶ on the TEE side first and then ❷ on the REE side.

[19]CAs, being on the REE side, are vulnerable to malicious REE OS.

V. Portability of the attack

The attack described in Sec. IV combined bus faults with the specific descriptions of error codes in the Global Platform (GP) API specification to bypass signature checks and install adversarial controlled, self-signed, malicious Trusted Applications in the secure world side of OP-TEE. However, the foundation of the attack (i.e. the specific error code definitions in GP API specifications) is more fundamental, thereby ensuring portability of the attack. To empirically consolidate this point, we successfully extended the attack in the following independent directions:

- **Attack on a new TEE based on GP API specifications**: MyTEE [50] is an extension of OP-TEE with several security features allowing better trust guarantees.
- **Attack on a new platform**: We ported both OP-TEE and MyTEE to a new platform (RPi4) and successfully recreated the attack.

In the following subsections, we elaborate on details of these two directions.

A. Attacking a new TEE: MyTEE

To empirically justify portability of our attack on a new TEE, we extend our attack on MyTEE [50]: a hardened implementation of OP-TEE that fixes several problems OP-TEE's RPi3 port has, with following enhancements:

- **Implementing secure memory**: OP-TEE port on RPi3 clearly lacks support for secure memory isolation, which MyTEE plugs in. To do so, MyTEE implements deliberate management routines for page tables. Some routines of MyTEE reside in the hypervisor, which ensure the TEE's memory map is immutable.
- **Implementing secure DMA extensions**: In OP-TEE on RPi3, a malicious Direct Memory Access controller (DMA) can issue memory-mapped I/O requests (MMIO) and build a harness for an attack. However, MyTEE plugs in this problem with a specialized DMA filter that interrupts any MMIO request to memory maps belonging to the secure world.
- **Implementing secure I/O**: The OP-TEE port on RPi3 also does not guarantee secure peripheral isolation on the secure world side. To implement secure peripheral extensions, MyTEE moves memory regions (like I/O buffers) to TEE's secure memory. Moreover, the concerned device drivers are broken into two parts: one portion residing in the normal world and the other part residing in the secure world, with secure communication APIs between the two.

The secure world part is given hypervisor privilege to access the secure objects (like I/O buffers).

We note that even with these extensions to a generic OP-TEE port, we are still able to launch our attack successfully. The reason for this is that the attack exploits design decisions in the GlobalPlatform API specifications, and not the implementation. Even though OP-TEE and MyTEE heavily differ in their security guarantees, they still share the same design principles, which is what we target. The attack described thereby still works on MyTEE without any modifications.

B. Attacking a new platform: targeting RPi4 TEE ports

To further consolidate the portability of this attack on a different platform, we ported both OP-TEE and MyTEE to RPi4 [92]. To find optimal fault locations, we repeated all the steps (c.f. Appendix VI) earlier performed for RPi3. We note that apart from modifications to attack parameters (Tab. V) because of change in target platform, no other change to the attack was necessary. We reiterate that this outcome is because our attack is dependent on specific design decisions in the GlobalPlatform API specification (off which both OP-TEE and its hardened variant MyTEE are based). Hence, the attack rationale is based off a more fundamental observation, and thereby is permeable across different target platforms, establishing the portability of the attack.

VI. Experimental Setup and Results

We now provide experimental details of the attack setup. The victim device is a RPi3 Model B (or RPi4 after applying the RPi4 port [92] as discussed in Sec. V-B). The adversarial device is a RPi4, which is connected to a Keysight 33500B signal generator responsible for triggering the entire fault injection process. When the RPi4 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 RPi3 Model B is 600 MHz). These amplified pulses are received by Rigol NFP-3 P3 EM (electromagnetic) probe and injected onto the system. The OP-TEE codebase was compiled with the default optimization flag [93].[20] We defer to Appendix A for more details on reproducing the experimental setup.

To generate a TA-like binary, the adversary creates a normal C based-program based off the GlobalPlatform's Internal Core API specification [56]. This binary is named `<uuid>.ta` where `<uuid>` denotes the publicly known UUID chosen by the adversary. The adversary then 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 mislead the check completely by using a **data bus fault** to change the value of the return code to `0x0`.

A. Searching the fault parameter space

Searching through the entire parameter space (wrt. all parameters related to the different devices used for fault injection) 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 and the TrustZone bails out in such an occurrence.

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 7). From analyzing the disassembly of the signature verification process presented in Listing 5, 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. The reason for adding three triggers in Listing 7 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. Refer to the oscilloscope output in Fig. 4 for pictorial annotations of Listing 7.

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

Listing 7: 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 Tab. V. While frequency, amplitude, and pulse width are self-explanatory, 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

[20] Tab. IV in Appendix gives the state of the OP-TEE monorepo.

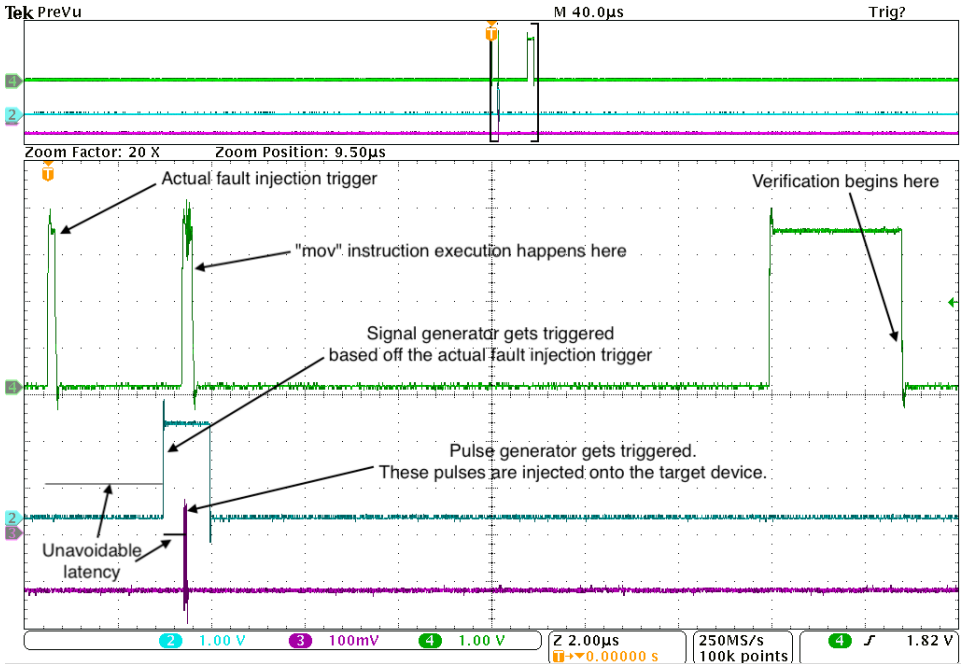


Fig. 4: Oscilloscope output of a successful faulting of `mov` instruction, as depicted in Fig 10. The X-axis denotes time/sampling duration (in order of μs). The Y-axis denotes amplitudes (in volts, milli-volts, and volts respectively for the three signals). The topmost signal represents the input coming from RPi3. 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 execution.

delay is essential to account for the unavoidable latencies of signal transmission. We refer to the appendix A (Fig. 10) for more details on the exact result of faults injected in Listing 7.

B. *Analysing power traces for triggering mechanism.*

In Sec VI-A, 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 [94]. Since the `mov` instruction of our interest comes after RSA signature verification, this pattern is hereafter used to trigger fault injections. To do this, the adversarial device (RPi4) was in constant communication with the oscilloscope through the Virtual Instrument Software Architecture (VISA) protocol, receiving the oscilloscope output as Comma Separated Values (CSV) file. By analysing CSV values for known executions, the adversarial device was able to identify when the RSA signature verification kicked in. At that moment, the adversarial device switched a chosen `GPIO` pin as `HIGH`, thereby launching the entire attack.

VII. UUID CONFUSION: A GLOBALPLATFORM API SPECIFICATION VULNERABILITY

In Sec. VI, we discussed how the adversary can choose its own unique identifier (or UUID). In this section, we bring forward a novel UUID confusion exploit stemming from the GlobalPlatform API specification that allows *an adversary to install a self-signed, malicious TA that can* ***masquerade*** *as another innocent TA in the TEE*. Concretely, the adversarial TA can re-route communication meant for other innocent TAs in the system to itself. This exploit can have far-reaching implications as even if the OEM whitelists trusted TAs, an adversary can masquerade as one of the whitelisted TAs and install a malicious one instead.

We note that the GlobalPlatform API specification mentions that ① each TA has its own UUID (Universally Unique Identifier) and ② this UUID is public knowledge. UUIDs allow a CA in the normal world to initiate communication with a TA in the secure world by uniquely labelling all TAs. Now since the TAs are supposed to be pre-deployed (signed by the OEM), the chances of two TAs sharing the same UUIDs do not arise because of OEM's involvement. Interestingly, the specification does not mention how such a situation should be handled in case two TAs are assigned the same UUID. However, with our bus faults, the adversary can not just deploy a malicious TA, but also choose an arbitrary UUID for it.

UUID confusion exploit works by relying upon 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. In our investigation, we made empirical observations about the interplay of persistence and UUID. OP-TEE makes two design decisions based off GlobalPlatform API specification: 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**. Thereby, an adversary needs to install its malicious TA with the GP API specification flag `TA_FLAG_EXEC_DDR`, which can then masquerade as any innocent pre-installed TA compiled with the GP API specification flags for ensuring persistence: `TA_FLAG_SINGLE_INSTANCE`, `TA_FLAG_INSTANCE_KEEP_ALIVE`, and `TA_FLAG_MULTI_SESSION`.

There are far reaching consequences of the UUID confusion exploit for an in-the-wild deployment of OP-TEE. On similar lines as *SeCReT* (cf. Sec. IV) providing communication privacy, Linaro and OP-TEE developers have proposed extensions like *Gatekeeper* [95] that provides *source authentication*, which is missing from default OP-TEE configuration. This *Gatekeeper* resides as a persistent TA, receives incoming communication requests from CAs, authenticates them, and then allows access to relevant TAs. Essentially, communications to any TA succeed only upon successful authentication by the *Gatekeeper*. However, through the UUID confusion exploit, a non-persistent malicious TA installed through bus faults (cf. Sec. IV) can *replace* this *Gatekeeper*, and whitelist all incoming connection requests from any CA, thereby effectively removing any source authentication from the system. A possible countermeasure could be to modify the GlobalPlatform API specification to force explicit checks while launching TAs, such that no two TAs can have the same UUID.

VIII. Countermeasures

In this section, we propose software-based countermeasures for preventing our attack. We note that the installation of a malicious TA was successful because of the convention of using `0x0` to denote success. For this, we suggest two software countermeasures: ① introduce redundancy checks (i.e. checks on the return values of sensitive functions), and/or ② change return values denoting success from `0x0` to a non-zero value. At the time of submission, the OP-TEE developers have implemented countermeasure ①, which we have evaluated and have found to be successful in preventing the mentioned attack vectors. Technically, two new functions are introduced: `callee_done_not_zero` and `callee_done_memcmp`. Function `callee_done_not_zero` is added in checks for every function call which are susceptible to fault injection attacks. This function makes sure that the return code is actually in conformance with the state `state` of the run. Likewise, function `callee_done_memcmp` is responsible for making sure all copying to TEE memory has indeed not been affected by injected faults, by redoing the memory copy and doing redundant comparisons with the return code. We refer to Listing 8 for the exact implementation.

```

#define TEE_SUCCESS 0x00000000
#define TEE_ERROR_SECURITY 0xFFFF000F

void callee_done_not_zero(struct state *st){
    /*Used when we are sure the return code should NOT be zero*/
    if(st->return_code == 0)
        PANIC();    // Panic and exit execution
}

void callee_done_memcmp(struct state *st, const void *p1, const void *p2, size_t nb)
    {
    /*Used to ensure non-faulty memcmp upon TEE invocation*/
    int redundant_result_code = memcmp(p1, p2, nb);
    if(!nb && redundant_result_code != st->return_code)
        PANIC();    // Panic and exit execution
}

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
void load_TA(...){
    // some code here
    struct state *st = /*initialize hash state*/
    TEE_Result res = verify_signature(..., st) /* Updates st->return_code */
    if(res != TEE_SUCCESS && callee_done_not_zero(st))
        // abort execution
    // some more code here
}
```

Listing 8: Redundant checks to ensure the return code of verify_signature() is not faulted.

IX. Conclusion

Presence of countermeasures like access control checks, control flow integrity checks etc. complicate the portability of Fault Injection attacks relevant to FPGAs and ASICs to modern System-on-a-Chip (SoC). In this work, we mount a novel attack on a previously unexplored architectural aspect of an SoC- the system bus. We show how targeted electromagnetic injections during operation of memory instructions cause faults in the contents of data and address buses. Through such faults, we enable classic Differential Fault Analysis attack on table-based implementation of AES as well as leak secret keys of NIST post-quantum finalists like Dilithium, Kyber and Saber with *a single fault*. To further contextualize the potency of bus faults, we mount an end-to-end attack on a practical implementation of ARM TrustZone, demonstrating a complete breakdown of TrustZone's security guarantees. This work, therefore, reinforces the importance of considering not only the cryptographic implementations but also the execution environment where the algorithms are expected to operate.

Acknowledgement

The authors would like to thank the anonymous reviewers and shepherd for their insightful comments for improving the paper. They would also like to thank the Department of Science and Technology (DST), Govt of India, IHUB NTIHAC Foundation, C3i Building, Indian Institute of Technology Kanpur, and Centre on Hardware-Security Entrepreneurship Research and Development, Meity, India, for partially funding this work.

References

[1] L. S. Vailshery, "Statista," Sep 6, 2022, accessed on Feb 15, 2023. [Online]. Available: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/

[2] D. Mukhopadhyay and R. S. Chakraborty, *Hardware security: design, threats, and safeguards*. CRC Press, 2014.

[3] M. Gross, J. Krautter, D. Gnad, M. Gruber, G. Sigl, and M. Tahoori, "Fpganeedle: Precise remote fault attacks from fpga to cpu," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, 2023, pp. 358–364.

[4] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitras, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks." in *USENIX Security Symposium*, 2019, pp. 497–514.

[5] G. A. Sullivan, J. Sippe, N. Heninger, and E. Wustrow, "Open to a fault: On the passive compromise of {TLS} keys via transient errors," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 233–250.

[6] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault attacks on rsa with crt: Concrete results and practical countermeasures," in *CHES*, vol. 2523. Springer, 2002, pp. 260–275.

[7] C. H. Kim and J.-J. Quisquater, "Fault attacks for crt based rsa: New attacks, new results, and new countermeasures," in *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems: First IFIP TC6/WG 8.8/WG 11.2 International Workshop, WISTP 2007, Heraklion, Crete, Greece, May 9-11, 2007. Proceedings 1*. Springer, 2007, pp. 215–228.

[8] 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.

[9] S. Bhasin, N. Selmane, S. Guilley, and J.-L. Danger, "Security evaluation of different aes implementations against practical setup time violation attacks in fpgas," in *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*. IEEE, 2009, pp. 15–21.

[10] N. Selmane, S. Bhasin, S. Guilley, T. Graba, and J.-L. Danger, "Wddl is protected against setup time violation attacks," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2009, pp. 73–83.

[11] M. Hutter and J.-M. Schmidt, "The temperature side channel and heating fault attacks," in *Smart Card Research and Advanced Applications: 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers 12*. Springer, 2014, pp. 219–235.

[12] L. Zussa, J.-M. Dutertre, J. Clédiere, B. Robisson, A. Tria *et al.*, "Investigation of timing constraints violation as a fault injection means," in *27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France*. Citeseer, 2012, pp. 1–6.

[13] 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.

[14] C. O'flynn and Z. Chen, "Chipwhisperer: An open-source platform for hardware embedded security research," in *Constructive Side-Channel Analysis and Secure Design: 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers 5*. Springer, 2014, pp. 243–260.

[15] A. Barenghi, G. Bertoni, E. Parrinello, and G. Pelosi, "Low voltage fault attacks on the rsa cryptosystem," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2009, pp. 23–31.

[16] N. Timmers, A. Spruyt, and M. Witteman, "Controlling pc on arm using fault injection," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2016, pp. 25–35.

[17] T. Korak, M. Hutter, B. Ege, and L. Batina, "Clock glitch attacks in the presence of heating," in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2014, pp. 104–114.

[18] P. Maistri, R. Leveugle, L. Bossuet, A. Aubert, V. Fischer, B. Robisson, N. Moro, P. Maurine, J.-M. Dutertre, and M. Lisart, "Electromagnetic analysis and fault injection onto secure circuits," in *2014 22nd International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2014, pp. 1–6.

[19] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. Ieee, 2013, pp. 77–88.

[20] B. Yuce, P. Schaumont, and M. Witteman, "Fault attacks on secure embedded software: Threats, design, and evaluation," *Journal of Hardware and Systems Security*, vol. 2, pp. 111–130, 2018.

[21] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[22] A. Vasselle, H. Thiebeauld, Q. Maouhoub, A. Morisset, and S. Ermeneux, "Laser-induced fault injection on smartphone bypassing the secure boot," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2017, pp. 41–48.

[23] C. Fanjas, C. Gaine, D. Aboulkassimi, S. Pontié, and O. Potin, "Combined fault injection and real-time side-channel analysis for android secure-boot bypassing," in *Smart Card Research and Advanced Applications: 21st International Conference, CARDIS 2022, Birmingham, UK, November 7–9, 2022, Revised Selected Papers*. Springer, 2023, pp. 25–44.

[24] T. Schneider, A. Moradi, and T. Güneysu, "Parti–towards combined hardware countermeasures against side-channel and fault-injection attacks," in *Advances in Cryptology–CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II 36*. Springer, 2016, pp. 302–332.

[25] S. Saha, D. Jap, D. B. Roy, A. Chakraborty, S. Bhasin, and D. Mukhopadhyay, "A framework to counter statistical ineffective fault analysis of block ciphers using domain transformation and error correction," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1905–1919, 2019.

[26] Y. Ishai, A. Sahai, D. A. Wagner *et al.*, "Private circuits: Securing hardware against probing attacks." in *CRYPTO*, vol. 2729. Springer, 2003, pp. 463–481.

[27] O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede, "Consolidating masking schemes," in *Advances in Cryptology–CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I 35*. Springer, 2015, pp. 764–783.

[28] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold implementations against side-channel attacks and glitches," in *ICICS*, vol. 4307. Springer, 2006, pp. 529–545.

[29] H. Groß, S. Mangard, and T. Korak, "An efficient side-channel protected aes implementation with arbitrary protection order," in *Topics in Cryptology–CT-RSA 2017: The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14–17, 2017, Proceedings*. Springer, 2017, pp. 95–112.

[30] C. Dobraunig, M. Eichlseder, T. Korak, S. Mangard, F. Mendel, and R. Primas, "Sifa: exploiting ineffective fault inductions on symmetric cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 547–572, 2018.

[31] S. Saha, A. Bag, D. Basu Roy, S. Patranabis, and D. Mukhopadhyay, "Fault template attacks on block ciphers exploiting fault propagation," in *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*. Springer, 2020, pp. 612–643.

[32] S. Saha, A. Bag, D. Jap, D. Mukhopadhyay, and S. Bhasin, "Divided we stand, united we fall: Security analysis of some sca+ sifa countermeasures against sca-enhanced fault template attacks," in *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part II*. Springer, 2021, pp. 62–94.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] N. Timmers and C. Mune, "Escalating privileges in linux using voltage fault injection," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2017, pp. 1–8.

[40] X. M. Saß, R. Mitev, A.-R. Sadeghi, and V. F. I. VFI, "Oops..! i glitched it again! how to multi-glitch the glitching-protections on arm trustzone-m," *arXiv preprint arXiv:2302.06932*, 2023.

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

[42] 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.

[43] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019.

[44] R. Elnaggar, S. Chen, P. Song, and K. Chakrabarty, "Detection of rowhammer attacks in socs with fpgas," in *2020 IEEE European Test Symposium (ETS)*. IEEE, 2020, pp. 1–2.

[45] ——, "Securing socs with fpgas against rowhammer attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 7, pp. 2052–2065, 2021.

[46] M. Taouil, C. Reinbrecht, S. Hamdioui, and J. Sepúlveda, "Lightroad: Lightweight rowhammer attack detector," in *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2021, pp. 362–367.

[47] M. Lajolo, "Bus guardians: an effective solution for online detection and correction of faults affecting system-on-chip buses," *IEEE Transactions on very large scale integration (VLSI) systems*, vol. 9, no. 6, pp. 974–982, 2001.

[48] M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential fault analysis of the advanced encryption standard using a single fault," in *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication: 5th IFIP WG 11.2 International Workshop, WISTP 2011, Heraklion, Crete, Greece, June 1-3, 2011. Proceedings 5*. Springer, 2011, pp. 224–233.

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

[50] S. Han and J. Jang, "Mytee: Own the trusted execution environment on embedded devices," *Network and Distributed System Security (NDSS)*, 2023.

[51] D. Suciu, S. McLaughlin, L. Simon, and R. Sion, "Horizontal privilege escalation in trusted applications," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[52] OP-TEE., "Supported Platforms." https://optee.readthedocs.io/en/latest/general/platforms.html#platforms-supported.

[53] ——, "Porting guidelines." https://optee.readthedocs.io/en/latest/architecture/porting_guidelines.html.

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

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

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

[57] M. Systems., "CAN Bus Fault Injection." https://www.machsystems.cz/en/products/embedded-networking/accessories/can-bus-fault-injection.

[58] Riscure., "Huracan automotive security tools." https://www.riscure.com/security-tools/huracan-automotive-security-tools/.

[59] N. L. . G. Hotz., "How the PS3 hypervisor was attacked," https://rdist.root.org/2010/01/27/how-the-ps3-hypervisor-was-hacked/, 2010.

[60] J.-M. Schmidt, M. Hutter, and T. Plos, "Optical fault attacks on aes: A threat in violet," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2009, pp. 13–22.

[61] A. Barenghi, G. M. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi, "Low voltage fault attacks to aes," in *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2010, pp. 7–12.

[62] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-dilithium: A lattice-based digital signature scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 238–268, 2018.

[63] M. J. Kannwischer, P. Schwabe, D. Stebila, and T. Wiggers, "Improving software quality in cryptography standardization projects," in *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2022, pp. 19–30.

[64] M. V. Beirendonck, J.-P. D'anvers, A. Karmakar, J. Balasch, and I. Verbauwhede, "A side-channel-resistant implementation of saber," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 17, no. 2, pp. 1–26, 2021.

[65] D. Heinz, M. J. Kannwischer, G. Land, T. Pöppelmann, P. Schwabe, and D. Sprenkels, "First-order masked kyber on arm cortex-m4," *Cryptology ePrint Archive*, 2022.

[66] P. Pessl and L. Prokop, "Fault attacks on cca-secure lattice kems," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 37–60, 2021.

[67] A. Sajid, H. Abbas, and K. Saleem, "Cloud-assisted iot-based scada systems security: A review of the state of the art and future challenges," *IEEE Access*, vol. 4, pp. 1375–1384, 2016.

[68] A. Boudguiga, N. Bouzerna, L. Granboulan, A. Olivereau, F. Quesnel, A. Roger, and R. Sirdey, "Towards better availability and accountability for iot updates by means of a blockchain," in *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2017, pp. 50–58.

[69] A. Cui and R. Housley, "{BADFET}: Defeating modern secure boot using {Second-Order} pulsed electromagnetic fault injection," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

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

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

[72] 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.

[73] 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.

[74] C. Cellar., "Voltage Fault Injection on a Modern RPi SBC." https://circuitcellar.com/research-design-hub/design-solutions/voltage-fault-injection-on-a-modern-rpi-sbc/.

[75] Riscure., "Controlling PC on ARM using Fault Injection." https://riscureprodstorage.blob.core.windows.net/production/2017/09/Riscure_Controlling_PC_FDTC_slides.pdf.

[76] C. Gaine, D. Aboulkassimi, S. Pontié, J.-P. Nikolovski, and J.-M. Dutertre, "Electromagnetic fault injection as a new forensic approach for socs," in *2020 IEEE International Workshop on Information Forensics and Security (WIFS)*. IEEE, 2020, pp. 1–6.

[77] F. Majéric, "Etude d'attaques matérielles et combinées sur les" system-on-chip"," Ph.D. dissertation, Lyon, 2018.

[78] 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.

[79] 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.

[80] 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.

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

[82] R. Schilling, P. Nasahl, and S. Mangard, "Fipac: Thwarting fault-and software-induced control-flow attacks with arm pointer authentication," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2022, pp. 100–124.

[83] J.-L. Danger, A. Facon, S. Guilley, K. Heydemann, U. Kühne, A. S. Merabet, and M. Timbert, "Ccfi-cache: A transparent and flexible

hardware protection for code and control-flow integrity," in *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, 2018, pp. 529–536.

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

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] D. Rosenberg., "Unlocking the motorola bootloader," http://blog.azimuthsecurity.com/2013/04/, 2013.

[92] P. Nebe., "RPi4 Port." https://github.com/peter-nebe/optee_os.

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

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

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

APPENDIX

TABLE IV: 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
buildroot	e6e12337f1874a5a53b42badf3d7fdd258d86a38
edk2	b24306f15daa2ff8510b06702114724b33895d3c
linux	b9a16995c467cc18cc26716d566c512fbac11069
mbedtls	e483a77c85e1f9c1dd2eb1c5a8f552d2617fe400
optee_benchmark	875be7f1959f19ed601ef37501f1cf0bfbee9da4
optee_client	f7ed8e3d3955e0b7a7d3ff77ab2abcfd8fb1cdb9
optee_os	837adc0a4c5dc462bfcc690618b812d838534fa5
optee_test	da5282a011b40621a2cf7a296c11a35c833ed91b
trusted-firmware-a	a1f02f4f3daae7e21ee58b4c93ec3e46b8f28d15
u-boot	b46dd116ce03e235f2a7d4843c6278e1da44b5e1

A. More Details on attack reproducibility

In this section, we elucidate the details on the experimental setup to aid attack reproducibility. To perform all fault injections in this work, we used a combination of the following hardware: Keysight 33500B signal generator, Keysight 81160A pulse train generator, Teseq CBA 400M-260 power amplifier, and Rigol NFP-3 P3 EM (electromagnetic) probe. The outputs of all these devices are connected to a Tektronix 4034B Mixed Signal Oscilloscope (MSO). The MSO is connected over LAN to the adversarial device, which allows the adversary to export all captured signals to a comma separated value (CSV) file for precise triggering timing analysis. Finally, the Rigol NFP-3 P3 EM (electromagnetic) probe is mounted upon an automated XYZ table that allows an automated mechanism to search for optimal fault injection position.

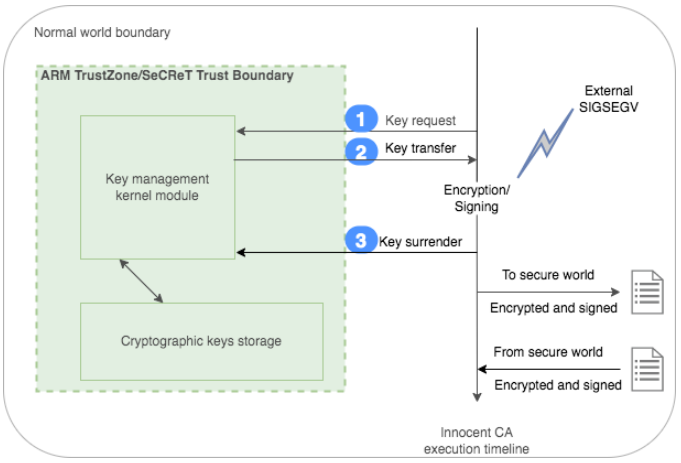


Fig. 5: CA's interaction with *SeCReT* to perform encryption/signing. A carefully timed SIGSEGV leaks the keys while they are in CA's memory.

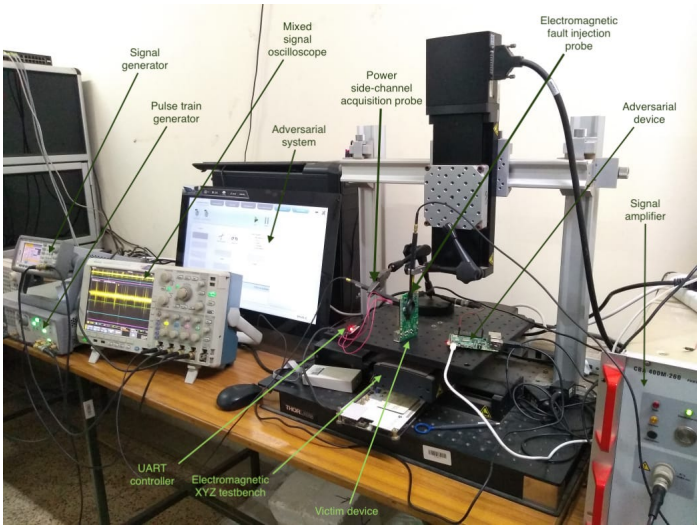


Fig. 6: The fault testbed developed to mount the attacks mentioned in this paper. The victim device is mounted upon a XYZ table allowing careful positioning of the electromagnetic (EM) probe. Note that the EM probe is positioned to inject EM pulses from the backside of the device: thereby faulting both 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.

In Tab. V, we note the various parameters for the Keysight

TABLE V: 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

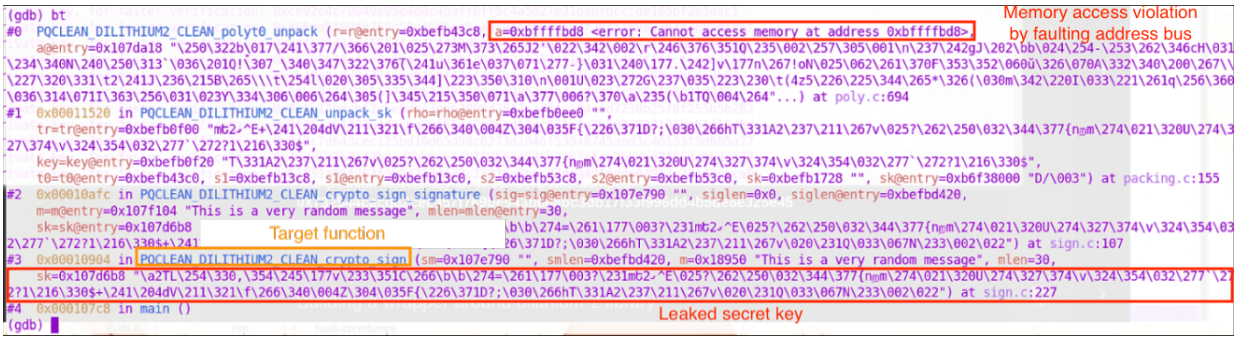


Fig. 7: A sample backtrace of a faulted Dilithium execution. Due to the injected electromagnetic pulses, a `const uint8_t * ` pointer in `PQCLEAN_DILITHIUM2_CLEAN_polyt0_unpack` got changed to an invalid address, causing the fault.

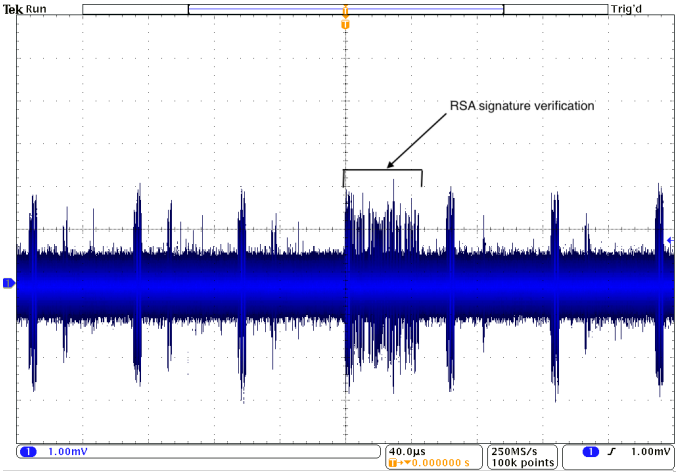
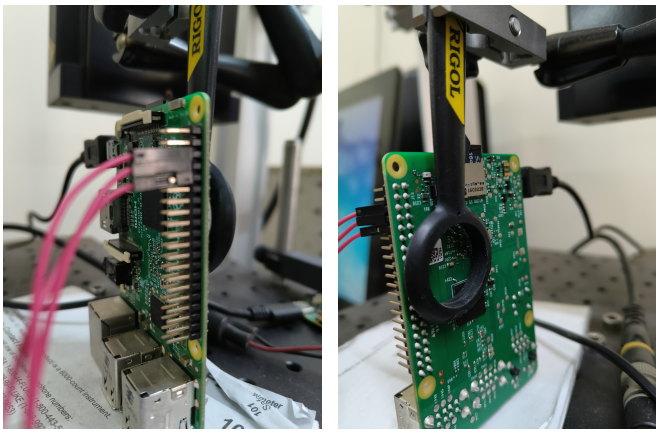


Fig. 8: The zoomed-in oscilloscope output during OP-TEE's verification of TAs. The x-axis denotes time/sampling duration (in order of μs). The y-axis denotes amplitudes (in milli-volts).



(a) Frontside. (b) Backside.

Fig. 9: Probe positioned over a RPi3 board: frontside/backside clicks.

33500B signal generator and the Keysight 81160A pulse train generator. Here, *frequency* denotes the frequency of the outputs of the two devices. Likewise, the *amplitude* of the signal generator output is uncontrolled, but the amplitude of the pulse train is carefully parameterized to generate pulses which are likely to create faults. Moreover, *offset* denotes the Y-axis shift of the generated signals. Likewise, *pulse width* denotes the

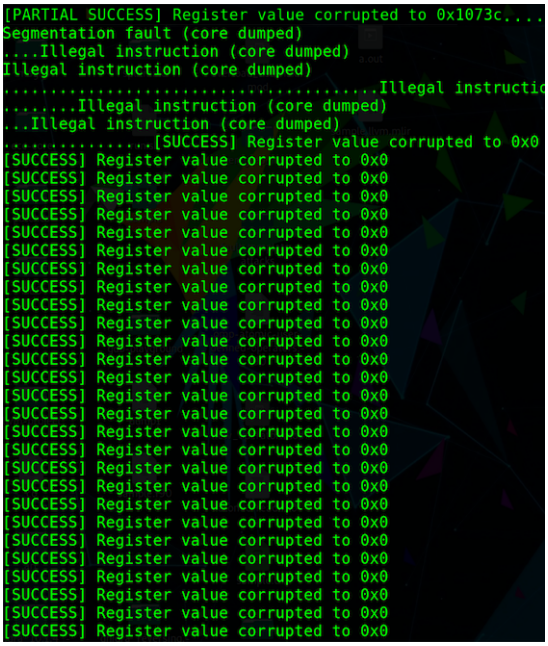


Fig. 10: The faults observed while attacking `mov` instruction in Listing 7 with the parameter set detailed in Tab V. Every dot (.) represents a correct expected execution. The `SIGSEGV` and `SIGILL` faults observed actually percolate to the instructions following the `mov` instruction. However, as the 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 proposed countermeasures.

time period of the output signals from both hardware. The pulse width of the signals from the pulse train generator is about 2 nanoseconds to allow high precision fault injections. The last three parameters are related to trigger signals. *Trigger threshold* refers to the maximum cutoff input voltage beyond which both generators start producing their respective outputs. Likewise, *trigger signal* denotes which portion of the square pulse (*rising edge*, *falling edge*, or *level*) to use for trigger. Without loss of generality, we choose the *rising* edge. Finally, in Tab. V, *trigger delay* represents the time duration between receipt of the input and generation of the output. This parameter can be tweaked to precisely time the fault injection process to coincide with the victim operation.