IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2022, No. 1, pp. 28–68. DOI:10.46586/tches.v2022.i1.28-68

Bypassing Isolated Execution on RISC-V using Side-Channel-Assisted Fault-Injection and Its Countermeasure

Shoei Nashimoto[1,2], Daisuke Suzuki[1], Rei Ueno[2] and Naofumi Homma[2]

[1] Mitsubishi Electric Corporation, Japan, Nashimoto.Shoei@bx.MitsubishiElectric.co.jp
[2] Tohoku University, Japan, homma@riec.tohoku.ac.jp

Abstract. RISC-V is equipped with physical memory protection (PMP) to prevent malicious software from accessing protected memory regions. PMP provides a trusted execution environment (TEE) that isolates secure and insecure applications. In this study, we propose a side-channel-assisted fault-injection attack to bypass isolation based on PMP. The proposed attack scheme involves extracting successful glitch parameters for fault injection from side-channel information under cross-device conditions. A proof-of-concept TEE compatible with PMP in RISC-V was implemented, and the feasibility and effectiveness of the proposed attack scheme was validated through experiments in TEEs. The results indicate that an attacker can bypass the isolation of the TEE and read data from the protected memory region. In addition, we experimentally demonstrate that the proposed attack applies to a real-world TEE, Keystone. Furthermore, we propose a software-based countermeasure that prevents the proposed attack.

Keywords: Fault Injection · RISC-V · Memory Protection · Trusted Execution Environment

1 Introduction

RISC-V is an open instruction set architecture (ISA), published in 2011 [PW17]. It has attracted considerable attention from both academia and industry due to features such as the absence of license fees, eliminating unnecessary functions in existing ISAs, and flexibility with respect to modular extensions [Int20]. Therefore, it can be used in various applications, from low-end embedded devices running bare-metal programs to high-end servers running the Linux operating system (OS).

It is important to design RISC-V by considering its security. Privileged instructions and a memory protection unit called physical memory protection (PMP) play an important role in its security, preventing malicious applications and/or libraries from accessing protected memory regions. Application execution based on memory isolation and the secure area isolated from the insecure area are referred to as isolated execution and trusted execution environment (TEE), respectively. Intel Software Guard Extensions (SGX) and ARM TrustZone are popular TEE-enabler technologies used in web servers and embedded devices.

Physical attacks, such as side-channel attacks and fault-injection attacks, should be considered from the viewpoint of embedded devices such as smartphones, gaming consoles, and electrical appliances [RRR+04, Gil15, PT17]. In particular, fault-injection attacks induce improper operations and/or data corruption during the momentary distortion of the power supply or by providing an abnormal clock signal to a target device. It has been reported that security mechanisms, such as secure boot and read protections, can be bypassed by

Licensed under Creative Commons License CC-BY 4.0.
Received: 2021-07-15 Accepted: 2021-09-15 Published: 2021-11-19

fault injection [WP17, VTM+18]. Although PMP did not originally address resistance to physical attacks as with other TEE-enabler technologies, the security evaluation of RISC-V against fault-injection attacks is a significant issue in practice [WSUM19].

In this study, we present a fault-injection attack against the security mechanism of RISC-V, that is, memory isolation by PMP. The basic idea is to bypass the isolated execution by skipping the PMP configuration with fault injection. The proposed attack targets the instructions for realizing memory isolation by PMP, whereas existing attacks, as in [WP17, VTM+18, BFP19], target an implementation-dependent fragment of code such as a secure boot and security configuration check. In particular, we focused on three types of instructions that change the PMP configuration. The features of the instructions allow the application of the proposed attack to any RISC-V-based TEE, starting with the extraction of successful fault injection parameters. To verify the feasibility of the proposed approach, we performed experiments with a proof-of-concept (PoC) TEE implementation compatible with PMP in RISC-V, owing to its flexibility and analyzability. In addition, we experimented using a real-world TEE to demonstrate the practicality of the proposed attack. We demonstrate that the attack can read the memory of a victim application protected by the PMP and propose a software-based countermeasure to prevent the proposed attack absolutely[1].

Related works. Fault-injection attacks were first proposed to compromise cryptographic processors [BDL97, BECN+06]. Since then, various injection techniques have been reported, in addition to theoretical studies. Clock glitch is a technique of inserting a distorted clock signal with a sudden voltage drop over a very short time [BRSK17, TSS17]. When applied to power supply, the same concept is referred to as a power glitch [BFP19]. Another fault-injection technique directly irradiates laser or electromagnetic (EM) waves [WP17, VTM+18]. The effect of fault injection on a target processor is represented by a *fault model* [YSW18], such as *instruction skip* and *data corruption* models.

Fault-injection attacks have recently been adopted to overcome security mechanisms. In [GA03, BTG10], a type-check operation on a Java virtual machine was subverted with fault injection, which resulted in the execution of an arbitrary code. In [NHH+17], the size limitation of the user input was broken by skipping the increment of a loop counter, causing a buffer overflow. In [VTM+18], the secure boot was bypassed by inducing bit errors in a security register with laser fault injection. In [TM17], as an attack after booting, privilege escalation was demonstrated with fault injection at the system call. Examples of practical attack scenarios include bypassing attacks against secure boot and TrustZone-based TEE by corrupting the program counter register [TSW16]. In [TSS17, QWLQ19a, QWLQ19b, MOG+20, KFG+20], dynamic voltage and frequency scaling (DVFS) was used to inject faults and successfully subvert ARM TrustZone and Intel SGX. In [WP17], joint test action group (JTAG) protection was proven to be subverted even in automotive safety integrity level D (ASIL-D)-certified microcontrollers. In [MTW+18, BFP19], memory dumps were performed by bypassing authentication or parameter checks with faults.

The methods for extracting fault-injection parameters and attacker models in some studies have not yet been clarified [GA03, BTG10, NHH+17], or attack scenarios are not realistic [TM17, TSW16, WP17]. For example, in [TSW16], there is no valid scenario in which an attacker's code can be executed on the target. In [WP17], fault timing is determined from the difference in power waveforms but implicitly assumes that the two power waveforms (with and without a countermeasure) can be obtained from the same target device.

The attacks on TEEs presented in [TSS17, QWLQ19a, QWLQ19b, MOG+20, KFG+20] are related to our proposed attack to break the TEE isolation. The main differences between

[1]The word *absolute* indicates that it is not a stochastic countermeasure that reduces the success rate of an attack, but a countermeasure that prevents an attack in principle.

the proposed attack and previous attacks lie in the architecture and protection mechanisms for isolation. In addition, the proposed attack defeats the isolation itself by inducing a fault in the PMP configuration, whereas previous attacks, such as in [TSS17, QWLQ19a, QWLQ19b, MOG+20], exploit data corruption and apply cryptanalysis techniques to extract a secret key or to subvert the signature verification. In [KFG+20], data corruption was adopted to break the message authentication code. Thus, the previous fault model and target function are different from ours.

In [WSUM19], countermeasures against fault-injection attacks were implemented on the RISCY core, and the overhead was evaluated. In [LBDPP19], the attack targeting hidden registers was proposed, and a simulation evaluation was performed. In [ELG20], a profiling evaluation of EM fault-injection attacks on a device implementing the E31 core was performed. However, no existing studies have evaluated the fault-injection attack resistance of security mechanisms on RISC-V.

Contributions. The contributions of this study can be summarized as follows.

1. We propose a fault-injection attack that defeats isolated execution on RISC-V. Furthermore, considering a more realistic setting, we propose a method to search for fault-injection parameters in a cross-device environment, which is more effective than brute force. This is the first fault-injection attack that targets the security mechanism of RISC-V.

2. We validate the feasibility and effectiveness of the proposed attack through experiments with the PoC TEE. We also demonstrate the practicality of the proposed attack by attacking Keystone as a real-world TEE. We demonstrate that an attacker can access the memory region of a victim application bypassing the isolated execution provided by the PMP.

3. We propose a software-based countermeasure against the proposed attack. The proposed countermeasure integrates the control flow and value verification to guarantee that the correct PMP value is set when running each application. We formulate all instruction skips that realize the proposed attack and prove that the attack cannot succeed in principle.

Paper organization. The remainder of this paper is organized as follows. Section 2 describes the security mechanism of RISC-V and existing TEEs implemented with RISC-V. In Section 3, the attacker model is introduced, and the proposed attack is explained. Section 4 describes the PoC TEE implemented based on the TEEs described in Section 2. Sections 5 and 6 describe the experiments using PoC TEE and Keystone, respectively. Section 7 presents an absolute countermeasure against the proposed attack. Section 8 discusses the proposed attack and countermeasure including the applicability of the attack to another architecture and the resistance of the countermeasure to other attacks. Finally, Section 9 concludes the paper.

2 Security on RISC-V

This section briefly explains the privileged architecture and PMP mechanism based on [PW17] and [WA19]. Existing TEE examples and their features are then introduced. Hereinafter, we focus on the 32-bit RISC-V architecture (RV32).

2.1 Privileged Architecture

RISC-V defines four modes in descending order of privilege: machine (M-mode), hypervisor (H-mode), supervisor (S-mode), and user (U-mode). A higher privilege mode can access

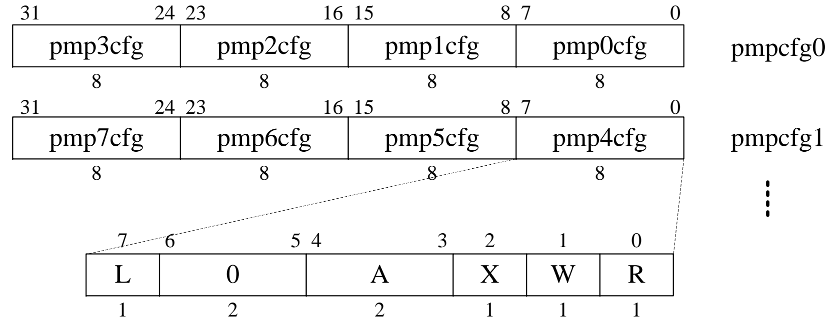


Figure 1: Format of `pmpcfg` [WA19]. Typically, `pmp`i`cfg` and `pmpaddr`i consist of a PMP entry.

all functions used in the lower-privilege modes. Therefore, the M-mode plays an important role in providing security. The M, H, S, and U modes are mainly used for the bootloader (or firmware), hypervisor, OS, and applications, respectively.

An important function of the M-mode is to handle exceptions. Hence, the privileged architecture provides special registers called control and status registers (CSRs). For example, in the CSRs, the `mcause` (Machine CAUSE) register memorizes why an exception occurs, and the `mie` (Machine Interrupt Enable) register defines the exceptions that should be handled. To realize isolated execution, the M-mode handles access fault exceptions caused by invalid memory access and environmental call exceptions caused by execution of `ecall` (environmental call) instruction.

2.2 Physical Memory Protection

The PMP consists of configuration (`pmpcfg`s) and address (`pmpaddr`s) registers included in the CSRs, defining the permission and its applied range, respectively. The PMP refers to these registers at every memory access and checks for permission. If not allowed, an access fault exception occurs, which is handled in the M-mode.

Figure 1 shows the structure of `pmpcfg`. An 8-bit `pmp`i`cfg` defines a PMP configuration ($0 \leq i \leq 15$). Four `pmp`i`cfg`s form a 32-bit `pmpcfg`j ($0 \leq j \leq 3$). Each `pmp`i`cfg` has attributes L, A, X, W, and R. X, W, and R indicate the executable, writable, and readable permission bits, respectively. L represents the lock bit; if $L = 1$, `pmp`i`cfg` does not change until the central processing unit (CPU) is reset. A represents the address-matching mode bit; it usually represents the naturally aligned power-of-two (NAPOT) and top of range (TOR) methods. According to `pmp`i`cfg`, NAPOT encodes `pmpaddr`i into the size and base address. The TOR covers the range between `pmpaddr`i-1 and `pmpaddr`i with `pmp`i`cfg`. Thus, NAPOT provides memory isolation with a pair of `pmp`i`cfg` and `pmpaddr`i, whereas TOR provides memory isolation with a set of `pmp`i`cfg`, `pmpaddr`i-1, and `pmpaddr`i. Hereinafter, the pair or set is referred to as *PMP entry*.

2.3 TEEs on RISC-V

Figure 2 shows a typical flow of the context switch under isolated execution by a TEE on RISC-V. The TEE is constructed by multiple applications running in U-mode, an OS in S-mode if it exists, and a monitor in M-mode. First, (1) an application calls the monitor using an exception or interrupt. Then, (2) the monitor handles the exception or interrupt and changes the PMP configuration by either switching a partial or rewriting all the PMP entries. Finally, (3) the monitor calls another application using privilege instructions.

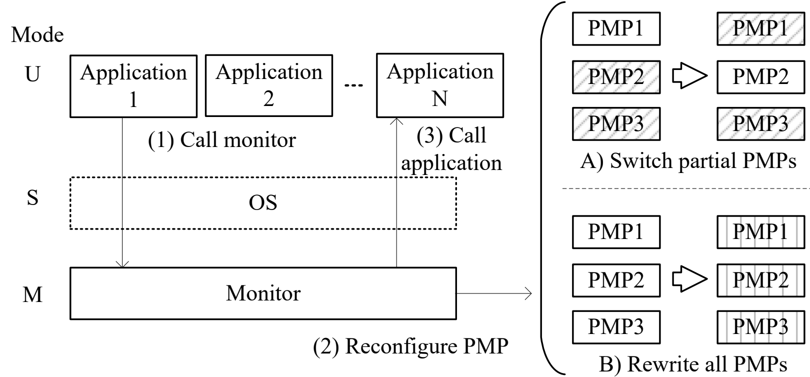


Figure 2: Context switch on TEE. A square box denoted as "PMPi" denotes i^{th} PMP entry.

The remainder of this section introduces two typical constructions of TEEs on RISC-V. The isolated execution can be represented as shown in Figure 2.

2.3.1 Keystone (UCB) [LK18, LKS+20]

Keystone adopts a two-world view model [VBOM+19] and separates the CPU memory into *untrusted* and *trusted* regions. An application running in U-mode in a trusted region is called an *enclave* application and is supported by the *enclave runtime* in the S-mode. The host OS and applications are considered untrusted. The enclave application is called from a host application. First, the host application calls the Keystone *security monitor* (SM) in M-mode via the OS by the supervisor binary interface (SBI) call implemented using `ecall`. Keystone SM deprives permissions of the caller application and permits the callee application (i.e., the application being called). Finally, Keystone SM calls the enclave application by the SBI call using `mret`. Keystone constructs a shared memory region using OS memory to exchange data between the host and the enclave applications or among the enclave applications.

2.3.2 MultiZone (Hex Five Security) [Sec21, Sec19]

The concept of MultiZone is to isolate all applications and libraries from each other. Each isolated unit running in U-mode is called a *zone* and is controlled by the *nanoKernel*. The context switch is realized as follows: First, a zone calls the nanoKernel by a timer interrupt or environmental call exception, according to the MultiZone application programming interface (API) function using `ecall`. Next, the nanoKernel changes the PMP entries for another zone[2]. Finally, the nanoKernel calls another zone using `mret`. MultiZone recommends using *InterZone Messenger* and not shared memory to exchange data between zones.

3 Proposed Attack

This section describes the attacker model to organize the information required for the proposed attack, and then presents the attack scheme to obtain the information and bypass isolated execution provided by the PMP.

[2] Although we do not perform operation analysis for the purpose of license agreement, there is no doubt that MultiZone adopts one of the PMP usages as shown in Figure 2.

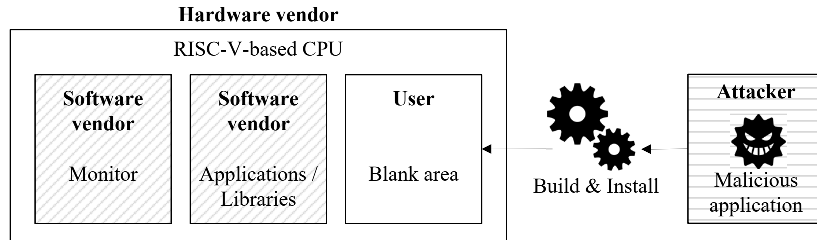


Figure 3: Attack scenario based on a use case for ARM TrustZone [Yiu15].

3.1 Attacker Model

We assume that the purpose of the attacker is to write and/or read memory regions protected by the PMP when the target device is running because TEEs guarantee only the isolated execution of applications[3]. Therefore, although reverse engineering or tampering with applications are attack vectors, they were not considered in this study.

The assumptions required for the attacker to inject faults into the device and collect side-channel information, such as the EM wave of the device, are summarized as follows: 1) the target device is present with the attacker; 2) the same device or chip as the target device is available for profiling (or reference); 3) the attacker can run any application in U-mode on the target device; 4) the attacker application can call other applications; 5) the TEE implementation is open, and 6) the attacker knows PMP values (i.e., allocated address) that are set when each application runs. As in assumption 5, the (open source) TEE code is known, but the applications running on the target device are unknown.

Figure 3 shows an attack scenario based on a typical use case for ARM TrustZone in which the above assumptions are valid. Given a CPU and software provided by hardware and software vendors, a user installs the application(s) in a blank region of the CPU [Yiu15], satisfying assumptions 1–4. Assumption 5 is satisfied if the attack target is an open-source TEE, such as the Keystone. Assumption 6 is satisfied by knowing in advance the addresses assigned to each application, as in the MultiZone example [Sec21], or by accessing the memory and identifying the range handled by the monitor as an exception.

The following steps are required for fault-injection attacks:

1. **Target instruction**: The attacker must determine which instruction should be skipped to break memory protection.

2. **Fault intensity (+ injection location)**: The attacker must determine the accurate fault intensity to obtain desirable fault effects. In the fault-injection method with spatial freedom, the injection location also needs to be determined.

3. **Fault timing**: The attacker must count the clock cycles from the trigger signal to the target instruction to inject faults with proper timing. When a trigger signal is obtained immediately before the target instruction, the fault timing need not to be considered.

4. **Trigger signal**: The attacker must obtain a trigger signal as a reference to determine the fault injection timing. In general, (1) communication signals such as the universal asynchronous receiver/transmitter (UART) signal, (2) digital signals using a general-purpose input/output (GPIO) port, and (3) power consumption due to distinctive operations such as cryptographic operations are considered [TM17, MTW+18, BFP19].

[3] To manage sensitive data and to guarantee confidentiality, authenticity, and integrity of applications, trusted platform module, application encryption, and remote attestation should be used instead of TEE.

The target instruction provides the novelty of the attack. Then, the target instruction and attack scenario constrain how fault parameters and trigger signal are obtained. This study focuses on target instruction, fault intensity, and fault timing, as discussed in the next section.

3.2 Attack Scheme

In this section, we present the basic idea of the proposed attack and its challenges. Then, we propose an attack scheme that involves obtaining fault-injection parameters for exploitation. Hereinafter, we refer to the bypass attack of isolated execution using fault injection as the *proposed attack*, and distinguish it from the *attack scheme* that shows a series of attack procedures.

3.2.1 Basic Concept

The basic idea of the proposed attack is to bypass reconfiguring the PMP setting at the context switch, enabling partial inheritance of the PMP setting of the previous application, which is the attack target. The proposed attack calls the target application and injects a fault when it returns to the attacker application. To this end, the possible target instructions for skipping are limited to the following three instructions[4]:

```
CSR Write: csrw csr, rs
CSR Clear: csrc csr, rs
CSR Set:   csrs csr, rs
```

where the first operand, `csr`, is either `pmpcfg`i or `pmpaddr`i, and `rs` indicates a source register storing the value written to the first operand. The reason for this limitation is that the PMP, composed of CSRs, requires special instructions to change their values. In other words, one or more of these instructions must be executed as long as the PMP provides memory protection to RISC-V. Hence, our attack can generally be applied to a variety of RISC-V-based TEEs.

3.2.2 Challenges

It is necessary to determine the appropriate fault intensity and timing. Because the fault sensitivity varies from one instruction to another [BGV11], preliminary profiling of target instruction(s) is required. However, target instructions are privileged instructions that cannot be profiled on target devices operable only in the U mode. Therefore, fault intensity is determined in a cross-device environment using a profiling device.

Because triggering immediately before the target instruction was not possible[5], the attacker first needs to determine the fault timing. It is difficult to calculate the clock cycle using code analysis because applications, except for the attacker application, are unknown. Side-channel-based reverse engineering, as in [VWG07, BTG10, SBO+15, PXJ+18, YUZP19], is a promising solution. It is also compatible with the proposed attack because there are only three types of target instructions, and the number of execution times is small compared to general-purpose instructions. In determining the fault timing, the specific challenges include the high operating frequency of the target device, the large number of pipeline stages, and the requirement of cross-device deployment.

[4] More precisely, they are pseudo-instructions using `csrrw`, `csrrc`, and `csrrs`, respectively.

[5] When pattern triggers are used, PMP switching occurs multiple times in a series of attacks (cf. Figure 10), so it is necessary to choose which trigger to use.

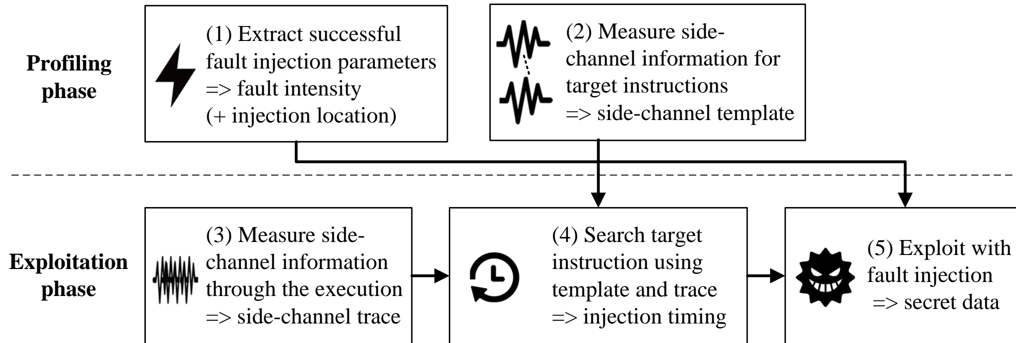


Figure 4: Proposed attack scheme.

3.2.3 Attack Scheme

The proposed attack scheme, based on the above-mentioned observations, is shown in Figure 4. The attack scheme consists of five steps divided into two phases: profiling and exploitation. In the profiling phase, a proper fault intensity (and injection location, if needed) is first extracted using a profiling device implemented on the same CPU as the target device. The side-channel information for each target instruction is then measured, and templates are created.

In the exploitation phase, the target device is used. First, a side-channel trace is captured, collecting information from the trigger signal to the execution of the target instruction. Then, the execution timing of the target instructions is identified using the templates and side-channel trace. Finally, the exploitation is performed with fault injection using the obtained fault intensity and timing. The concrete exploitation methods are described in the following sections.

4 Implementation of the Trusted Execution Environment

This section describes the PoC TEE targeted for the attack scheme. This PoC implementation is advantageous as it overcomes the inconveniences of existing TEEs. MultiZone has black-box components protected by patents and license agreements, making it difficult to analyze the success of our attack [Sec21]. Because Keystone requires relatively high-end devices running the Linux OS, further efforts are needed to apply power-based reverse engineering in a cross-device environment. Although the feasibility of the proposed attack is verified using Keystone, the application of the attack scheme is future work.

Our PoC TEE was implemented in a bare-metal manner (i.e., no OS) with the *Freedom Metal library* (v201908) developed by SiFive [SiF20]. We present the system structure, flowchart, and PMP usage in the PoC TEE. The detailed implementation of the PoC TEE is shown in Appendix A.

4.1 System Structure

Figure 5 shows the system structure of the PoC TEE. Comprising a monitor in the M-mode, three applications (APP1, APP2, and APP3) in the U-mode, and a shared library and memory that can be used in all modes. APP1 acts as a dispatcher and runs on the user commands via UART. It executes the command to send data to other applications, call other applications, and send processed results from other applications to the user. APP2 is a cryptographic application that executes the advanced encryption standard (AES) [Sma19] and has a secret key in its RAM region. APP3 is an attacker application

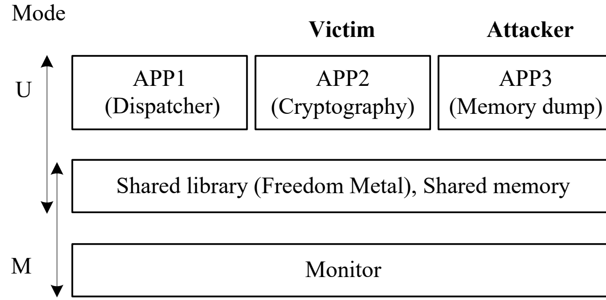


Figure 5: System structure of PoC TEE. Three applications are controlled by the monitor, and they share data with shared memory.

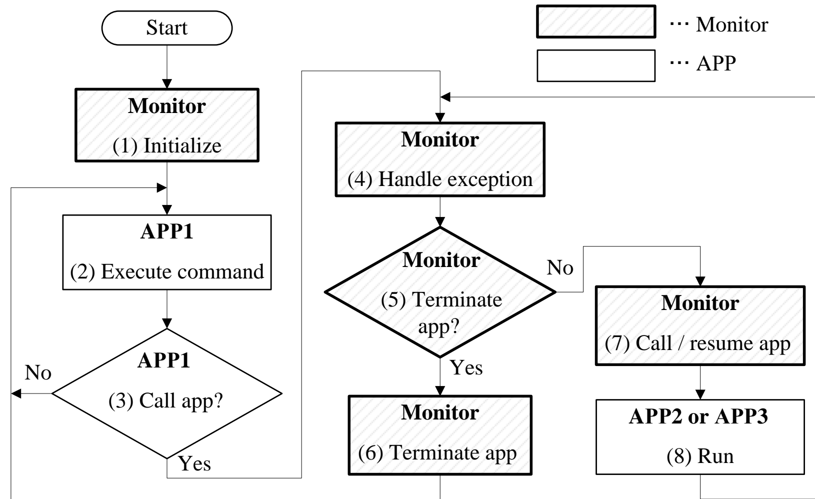


Figure 6: Flowchart of PoC TEE describing context switch handled by the monitor.

that dumps the RAM. More specifically, it obtains an address from the shared memory, reads data in the address, and stores the data in the shared memory.

The shared library is a subset of the Freedom Metal library. The PoC TEE mainly uses peripheral control functions for UART, GPIO, and PMP and exception handling functions. Shared memory is used to share data between isolated applications and those sending data to the monitor to call for other applications.

4.2 Flowchart

Figure 6 shows a flowchart of the PoC TEE behavior. In (1), the monitor first registers the exception handlers, initializes various variables, configures the PMP entries, and calls APP1. In (2) and (3), APP1 receives a user command and executes it. If required, it calls for another application using `ecall`. In (4), owing to the exception, the monitor runs the exception handler. During an environment call exception, the exception handler invokes the `ecall` handler registered in step (1). During a memory access fault exception, the monitor fills the data region of the shared memory with a value of 0xFF in hexadecimal, stops all running applications, and passes the control to APP1. In steps (5)–(7), the application call and finalization are executed. In (8), each application runs and returns to (4).

Table 1: PMP usage for rewriting method. N/A implies that PMP entry is disabled. All PMPi is configured by NAPOT.

PMP	APP1	APP2	APP3
PMP0	Shared library		
PMP1	Shared memory		
PMP2	ROM APP1	ROM APP2	ROM APP3
PMP3	RAM APP1	RAM APP2	RAM APP3
PMP4	UART	N/A	N/A
PMP5	N/A		
PMP6	N/A		
PMP7	N/A		

Table 2: PMP usage for switching method. The gray and white cells represent configurations with no accessibility (R=0, W=0, X=0) and all accessibility (R=1, W=1, X=1), respectively. In the case of "all region," PMP6 and PMP7 construct one entry with TOR. Apart from that, all PMP entries use NAPOT.

PMP	APP1	APP2	APP3
PMP0	ROM monitor		
PMP1	RAM monitor		
PMP2	ROM APP2	ROM APP2	ROM APP2
PMP3	RAM APP2	RAM APP2	RAM APP2
PMP4	ROM APP3	ROM APP3	ROM APP3
PMP5	RAM APP3	RAM APP3	RAM APP3
PMP6	All region	Shared library	
PMP7		Shared memory	

4.3 PMP Usage

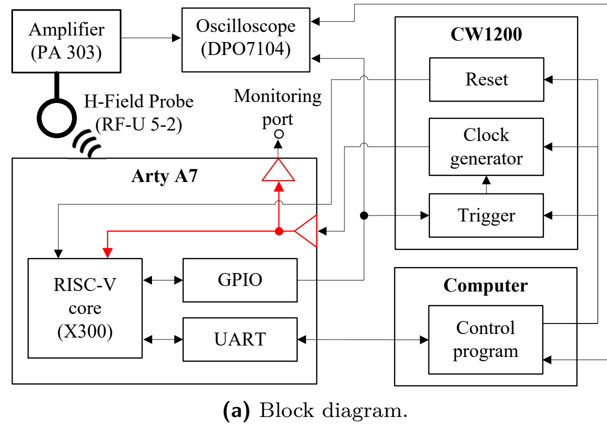
As shown in Figure 2, we implemented two types of PMP usage referred to as the rewriting and switching methods, respectively. The rewriting method, in Table 1, rewrites all PMP entries to realize isolated execution. The shared library and memory use two PMP entries. Each application uses two PMP entries for the isolation of ROM and RAM. If required, peripheral PMP entries are added. The switching method shown in Table 2 switches the permissions of PMP entries, that is, R, W, and X in `pmpcfg`, to provide isolation. We refer to [LKS+20] and consider APP1 as untrusted and APP2 and APP3 as trusted. The untrusted application can access all the memory regions as defined by PMP6 and PMP7 unless other PMPs forbid it. Only when the context switches from APP1 to APP2 (or APP3) or vice versa, PMP6 and PMP7 including `pmpaddr` are rewritten.

5 Experiment #1: Attack on PoC TEE

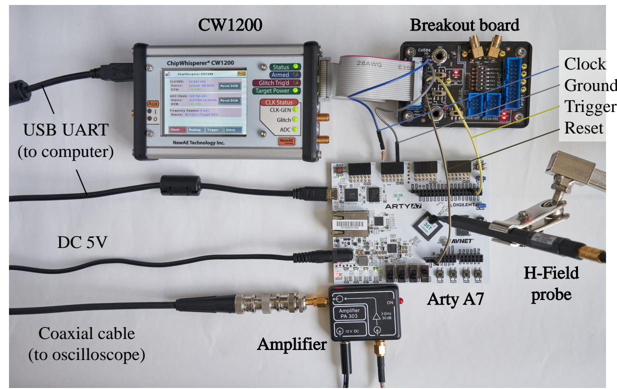
This section describes the experiments and the actual devices to validate the feasibility and effectiveness of the proposed attack scheme. First, the experimental setup is described. Then, we show two experimental results for extracting fault-injection parameters and exploitation based on the attack scheme in Figure 4.

5.1 Experimental Setup

This experiment employed clock-glitch injection as a fault injection technique because of its high repeatability and temporal resolution [YSW18]. A monitor generates a trigger signal before calling the attacker application for simplicity. However, an accurate fault-injection



(a) Block diagram.



(b) Overview.

Figure 7: Experimental setup. Computer is connected to Arty A7 and CW1200 with USB, and Arty A7 and CW1200 are connected with wires. The clock signal wire is equipped with a resistor of 100 Ω for impedance matching, and the clock signal is provided to the RISC-V core via input buffer. The computer and oscilloscope are connected via Ethernet, and the EM wave is acquired via the oscilloscope using an H-field probe. Modifications from the original system-on-chip design are shown in red, that is, clock input/output ports.

timing should be identified because the trigger is not necessarily generated immediately before the target instruction.

Figures 7(a) and (b) show the block diagram and overview of the experimental setup, respectively. An X300 RISC-V core (Hex Five) [Sec20], based on UCB's Rocket Chip [AAB+16], was implemented on an Arty A7 field-programmable gated array (FPGA) board to run the PoC TEE described in Section 4. The X300 RISC-V core supports the PMP and operates at an operational frequency of 65 MHz. For simplicity, we set up a port to provide an external glitchy clock (CW1200, NewAE Technology). A control computer communicated with Arty A7 and CW1200 via a universal serial bus (USB) UART. The computer calls an application and exchanges data in communication with Arty A7. The computer changes the glitch parameters (i.e., intensity and timing) and sends a reset command to Arty A7 in communication with CW1200. In addition, a DPO7104 (Tektronix), RF-U 5-2, and PA 303 (Langer EMV) measure EM leakage to determine the fault injection timing. According to the GPIO signal of Arty A7, DPO7104 transmits the measured EM leakage to the computer.

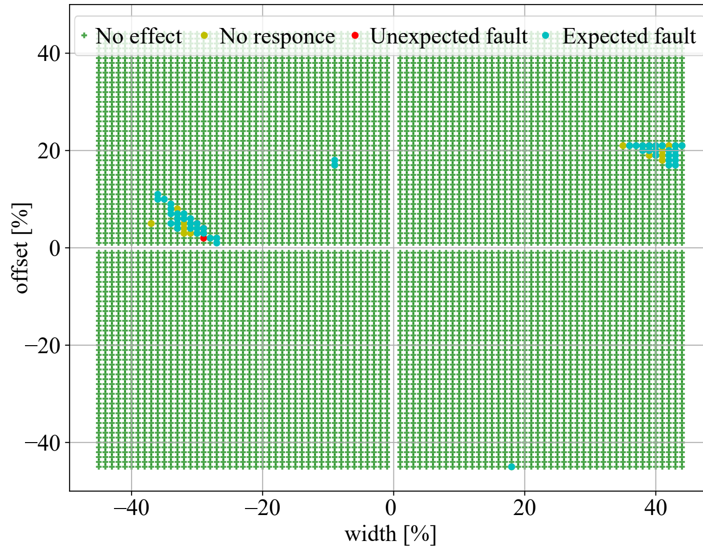


Figure 8: Characterization for searching proper glitch parameters. % represents ratio in percent of each parameter to the original clock width.

5.2 Experiment #1.1: Extracting Glitch Parameters

This section addresses steps (1)–(4) in the attack scheme shown in Figure 4. For the clock glitch provided by CW1200, the fault intensity and glitch timing are defined as the parameters of `width` and `offset`, and `external_offset`, respectively.

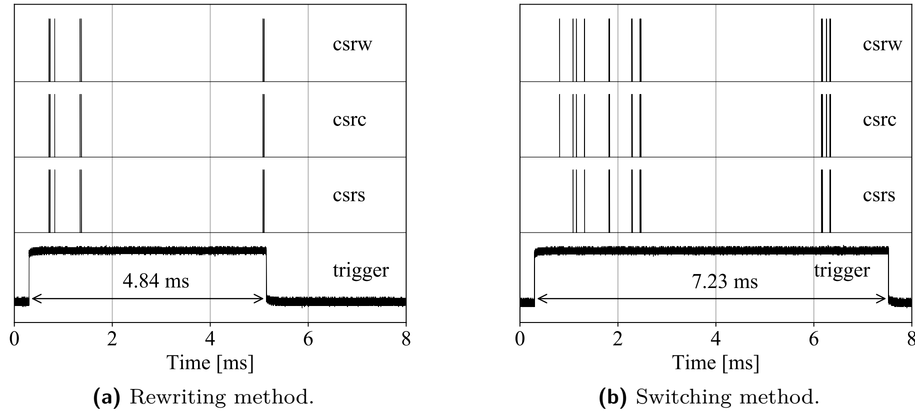
5.2.1 Fault Intensity

First, we experimentally obtained the `width` and `offset` using a test program to inject a fault into a profiling device. The program initializes GPIO, generates a pulse trigger signal, executes an instruction before and after a sufficient number of `nop`s, and sends the result of the attack. In this experiment, we assumed that the target instruction was "`csrw pmpcfg0, a5`," where register `a5` had a value of 0x1b1b1b1d. The attack results are given as the value of `pmpcfg0`. Thus, we obtain 0x00000000 and 0x1b1b1b1d as the success and failure to skip, respectively.

Figure 8 shows the experimental results, where the faults were injected 10 times for each glitch parameter. We changed the `width` and `offset` from -45% to +45% in steps of 1%. The fault results are overwritten on the graph in the order of no effect, no response, unexpected fault, and expected fault. The parameters plotted in blue indicate that the attack was successful at least once. The following exploitation experiment used all the parameters plotted in blue.

5.2.2 Glitch Timing

In this experiment, we obtained the `external_offset` by template matching of the EM leakage. Following assumption 5 in Section 3.1, we ran a PoC TEE code on the profiling device. For each of the three target instructions, EM waveforms for five clocks were obtained as templates, considering the pipeline size. In addition, we ran the target device and obtained EM leakage when the attacker application was running. The profiling and target devices are different to validate the effectiveness of cross-device template matching. The details of the template matching experiment is shown in Appendix B.3.



(a) Rewriting method. **(b)** Switching method.

Figure 9: Positional relation between trigger and execution timings for target instructions. The trigger signal becomes high-level logic during the attacker application running (cf. Section 5.3.1).

Figures 9(a) and (b) show the trigger signals and identified timings of the `csrw`, `csrc`, and `csrs` instructions, while running the PoC TEE with the switching and rewriting methods, respectively. The EM leakages are measured using an oscilloscope at a sampling rate of 1GS/s. Figure 9(a) shows that `csrw`, `csrc`, and `csrs` are executed eight times each during the trigger signal in the high-level logic state (several spike signals overlap). In Figure 9(b), `csrw`, `csrc`, and `csrs` are executed 16, 16, and 13 times, respectively. The elapsed time between the trigger and identified instruction timings for each target instruction is then used to calculate the number of elapsed cycles, corresponding to the `external_offset`. The following exploitation experiment used all the candidates obtained from the results.

5.3 Experiment #1.2: Exploitation

This section addresses step (5) in the attack scheme shown in Figure 4.

5.3.1 Operational Flow of Exploitation

Figure 10 shows the sequence diagram of the exploitation method. Again, the attacker cannot know the behaviors of APP1 and APP2. The computer first initializes CW1200 and sends a command to call APP3. APP1 receives the command and calls APP3 via the monitor. In this PMP reconfiguration, the monitor generates a trigger signal. APP3 stores the necessary data in shared memory in case it loses its RAM access permission. Then, APP3 directly calls the attack target (i.e., APP2) via the monitor. After encryption in APP2, the program flow returns to the caller application. CW1200 must inject faults at the proper timing with `external_offset` in this PMP reconfiguration for a successful attack. If the faults are induced correctly, APP3 obtains the target RAM access permission. APP3 reads the RAM data of APP2 and sends it to APP1 via shared memory. When APP3 completes its operations, the monitor moves the trigger signal to the low-level logic state. Finally, the computer sends the command to obtain the contents of the shared memory.

APP3 succeeds in dumping the target RAM data if the fault injection successfully bypasses the target instruction. The influence of faults on the exploitation is classified into the following four classes:

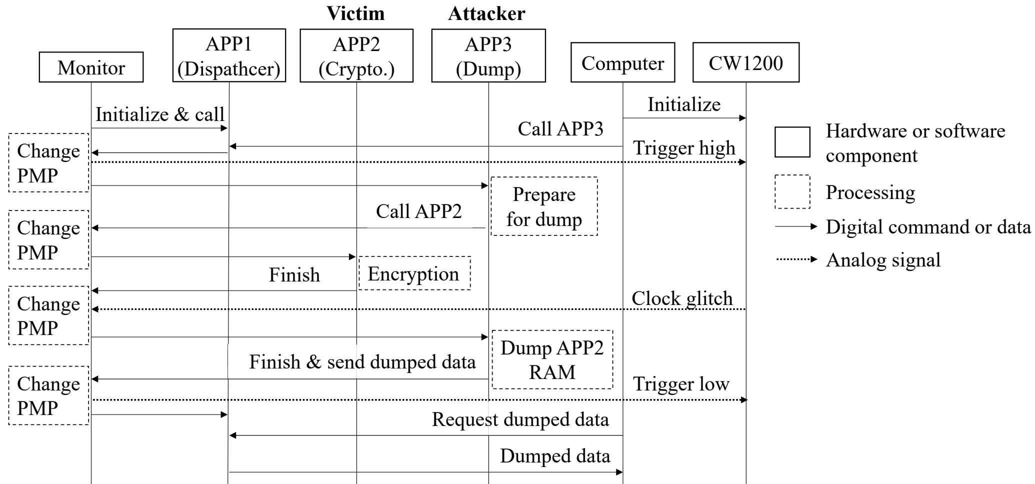


Figure 10: Sequence diagram for exploitation.

1. **No effect**: The CPU runs correctly, and the RAM access from APP3 to APP2 is handled as an access fault exception. Thus, we obtain 0xFFFF... because the shared memory is filled with 0xFF by the monitor.

2. **No response**: The CPU runs abnormally owing to excessive fault intensity. Thus, no results are obtained.

3. **Unexpected fault**: A fault is induced in the CPU, but the target instruction is not skipped. Thus, we obtain 0xFFFF... with no effect.

4. **Expected fault**: A fault is induced in the CPU, and the target instruction is skipped. Thus, we obtain the secret key held by APP2.

5.3.2 Exploitation in Rewriting Method

Attack attempts. We set all the fault parameters, that is, combinations of `width`, `offset`, and `external_offset`, obtained in Section 5.2 for the proposed attack. Here, we provided a margin of ± 50 cycles to the identified `external_offset` considering noise effects such as CPU pipeline and clock jitter. Finally, the secret key was obtained at 14 cycles after the identified 8th `csrw`.
Cause analysis. For the rewriting method, we skip the reconfiguration of PMP3, as shown in Table 1, to obtain the RAM permission for APP2 instead of APP3. This means that APP3 cannot use its stack memory. Hence, APP3 is written to avoid using local variables and function calls after calling APP2 or injecting the fault.

The attack requires the exchange of RAM permissions; therefore, `pmpcfg` does not change. Thus, the target instruction is only the reconfiguration of `pmpaddr`. The assembly code to reconfigure `pmpaddr` for PMP3 is as follows:

```
lw  a5,-64(s0)    // Load word (lw) on stack into a5
csrw pmpaddr3,a5  // Write addr value (a5) to CSR pmpaddr3
```

In this case, the target instruction is only `csrw`. If the `lw` instruction is skipped, register `a5` becomes undefined, and success or failure of the attack remains unknown. To summarize the above-mentioned observations, a successful glitch is considered to have skipped the `csrw` in the experiment.

Table 3: Fault intensity parameters and their success rates with 10 trials.

Fault intensity [%]		Success rate [%]		
Width	Offset	Profiling	Exploitation	
			Rewriting method	Switching method
-32	6	10	90	20
-31	5	50	90	20
-31	4	40	100	10
39	20	20	90	0
40	20	40	90	0
40	19	40	90	0
41	21	40	100	0
42	19	100	0	0
42	18	90	0	0

5.3.3 Exploitation in Switching Method

Attack attempts. We also performed an experiment to exploit the switching method. We successfully obtained the secret key through a fault injection attack using all the fault parameters extracted in Section 5.2. The successful `external_offset` was smaller by 42 cycles than the identified 13th `csrc`.

Cause analysis. For the switching method, we skip the reconfiguration of PMP3, as shown in Table 2, so that APP3 additionally obtains the permission of RAM for APP2. This PMP protection is weaker than that of the rewriting method. Therefore, we use the same code as in Section 5.3.2 (cf. Appendix A.6).

The switching method does not need to change `pmpaddr`; therefore, the target requires only the reconfiguration of `pmpcfg`. The assembly code used to reconfigure `pmpcfg` for PMP3 is as follows.

```
lw  a5,-24(s0)   // Load word (lw) on stack into a5
csrc pmpcfg0,a5  // Clear CSR pmpcfg0 with mask bit (a5)
lw  a5,-28(s0)   // Load word (lw) on stack into a5
csrs pmpcfg0,a5  // Set CSR pmpcfg0 with config bit (a5)
```

One `pmpcfg` has the configuration for four PMP entries, as shown in Figure 1. With the specification of the Freedom Metal library, `pmp`*i*`cfg` is cleared (`csrc`), and a new value is then set into `pmp`*i*`cfg` (`csrs`). Thus, the target instruction is limited to `csrc`. In summary, successful glitches are considered to have skipped `csrc`.

5.4 Evaluation of Glitch Parameters

This section evaluates the effectiveness of the proposed attack by comparing the experimental results of profiling and exploitation.

5.4.1 Fault Intensity

We performed the exploitation 10 times for each glitch parameter by fixing the value of `external_offset` when the attack was successful in each experiment as described in Sections 5.3.2 and 5.3.3. Table 3 summarizes a set of fault intensity values with success rates of 90% or more in each experiment, including the profiling experiment. The results show that (1) parameters with a high success rate differ between the profiling and exploitation experiments, and (2) parameters with a high success rate differ even in the exploitation experiments using the same device.

Observation (1) was based on individual differences. Even though such differences exist, the exploitation is successful. Observation (2) is due to the difference in the fault sensitivity for each instruction: the fault intensity targeting `csrw` was used to skip `csrc`, but additional experiments showed that the fault sensitivity of each instruction was different. See Appendix B.2 for details.

5.4.2 Glitch Timing

We also investigated shortening the time required for exploitation by identifying the timing of the target instructions. In the exploitation experiments, we expanded the `external_offset` by ± 50. Thus, 100 trials were performed for each candidate, resulting in 2,400 (100×24 candidates) and 4,500 (100×45 candidates) overall trials for the rewriting and switching methods, respectively. In contrast, for a brute-force attack, we need approximately 314,600 (65 MHz $\times$ 4.84 ms) and 469,950 (65 MHz $\times$ 7.23 ms) trials for each PMP usage, respectively. Thus, a reduction of more than 99% of the trials was achieved with the proposed attack scheme.

6 Experiment #2: Attack on Keystone

This section demonstrates the practicality of the proposed attack through experiments on Keystone. First, we describe the experimental setup. Next, we present the results of the exploitation experiment. In this experiment, steps (1)–(4) in the proposed attack scheme are completed in advance, and only the proposed attack is conducted.

6.1 Experimental Setup

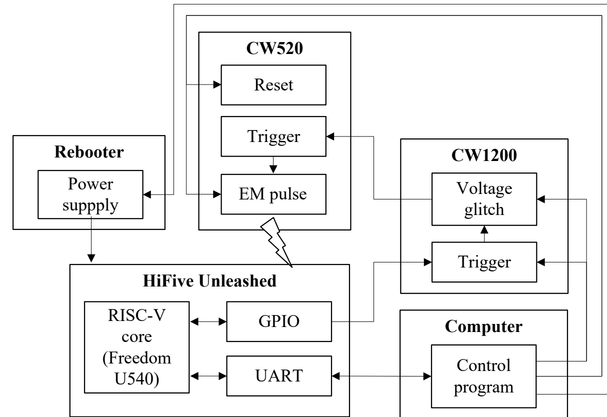
In this experiment, we employed EM injection without any device modification as a fault injection technique, while we modified the Keystone SM to generate a trigger signal immediately before the target PMP reconfiguration. Therefore, we omitted the adjustment of the fault-injection timing for simplicity.

Figures 11(a) and (b) show the block diagram and overview of the experimental setup, respectively. A HiFive Unleashed (SiFive) is equipped with a Freedom U540 RISC-V core (SiFive), and the official Keystone sample application (hello-native) [Lee21] is run. For an untrusted host application (happ), we added a process to access an enclave's memory after calling the enclave application (eapp). Meanwhile, the eapp is not changed.

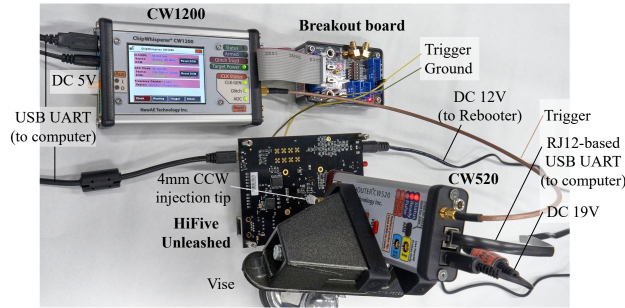
The EM pulses are injected by a CW520 (NewAE Technology) after receiving a trigger signal from a CW1200. Because HiFive unleashed has a CPU covered by a heat spreader, EM pulses are injected from the back side of the board. CW1200 generates the trigger signal after receiving a GPIO signal from HiFive Unleashed. A control computer communicates with HiFive Unleashed, CW540, and CW1200 via USB UART. The computer executes the happ and receives the result of accessing the enclave memory with HiFive Unleashed. The computer communicates with CW540 to change the fault-injection parameters (that is, voltage and pulse width) and reset CW540. In addition, a rebooter hard-reset HiFive Unleashed when an abnormal state occurs due to the EM fault injection.

6.2 Experiment #2.1: Exploitation

This experiment shows that the proposed attack enables the happ to access the protected area of the eapp.



(a) Block diagram.



(b) Overview.

Figure 11: Experimental setup. Computer is connected to HiFive Unleashed, CW520, and CW1200 with USB, and HiFive Unleashed and CW1200 are connected with wires. CW520 and CW1200 are connected with coaxial cable. Rebooter provides power for HiFive Unleashed. CW520 is fixed in position by vise. There is no hardware modifications.

6.2.1 Operational Flow of Exploitation

Figure 12 shows the sequence diagram for exploitation. First, the Keystone SM creates the enclave memory (E1). Then, the happ calls the eapp that stores the string "hello world" to the shared memory (U1). The eapp then stops, and the happ resumes. During this context switch, the PMP should restrict memory access for eapp (E1). This PMP reconfiguration is our target. Next, the happ accesses E1 and outputs the results. The happ then stops, and the eapp resumes, and the finalization process runs in the order of the eapp and happ. Finally, the Keystone SM destroys E1. Details of the target code are shown in Appendix C.

The happ succeeds in accessing E1 if the fault injection successfully bypasses the target instruction. As mentioned in Section 5.3.1, the influence of faults on exploitation is classified into four classes:

1. **No effect**: The CPU runs correctly, and accessing E1 is handled as an access fault exception.
2. **No response**: The CPU runs abnormally owing to the excessive fault intensity.
3. **Unexpected fault**: A fault is induced in the CPU, without skipping the target instruction. Thus, accessing E1 is handled as an access fault exception.

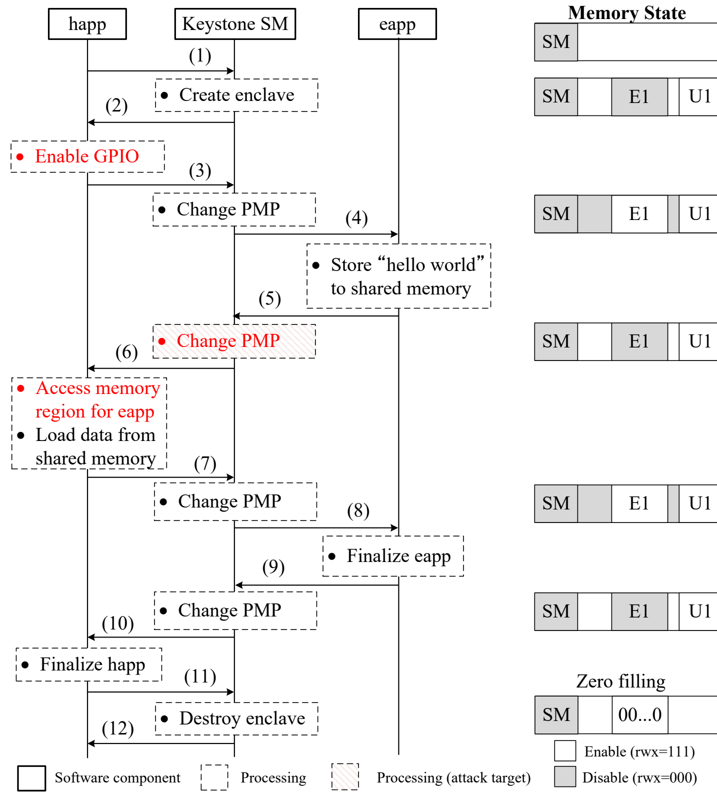


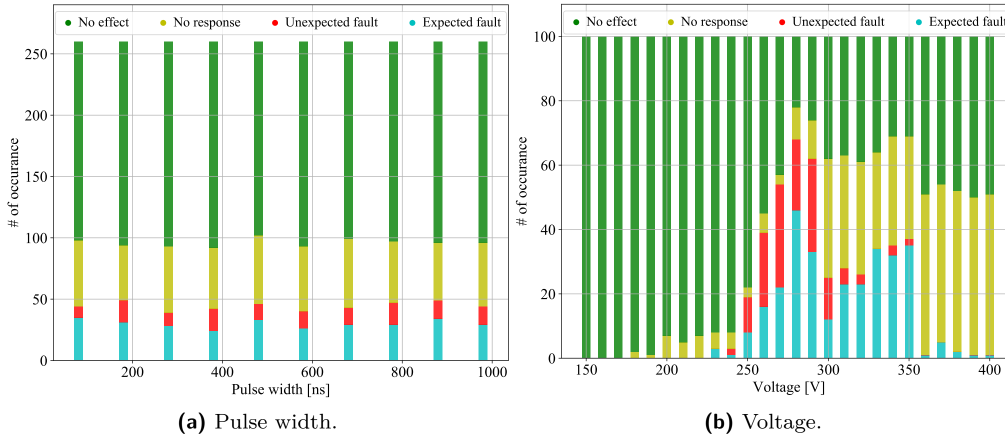
Figure 12: Sequence diagram for exploitation and memory state. Red text in the sequence diagram indicates changes from the original process. E1 and U1 in memory state represent memory for eapp and shared memory, respectively.

4. **Expected fault**: A fault is induced in the CPU, and the target instruction is skipped. As a result, we obtain the values stored in E1.

6.2.2 Result

In this section, we show the success rate of the proposed attack for each glitch parameter. First, we injected a fault while setting random glitch parameters to CW540 and identified a fault sensitive location. Next, the injection tip was fixed at the location (cf. Figure 11(b)), and attacks were performed 10 times for each glitch parameter (i.e., EM pulse width and voltage). Figure 13 shows the number of occurrences of each fault influence for the glitch parameter (summed for one glitch parameter). The pulse width was varied from 90 to 980 ns in 100 ns steps, and the voltage was varied from 150 to 400 V in 10 V steps.

Figure 13 shows that the expected fault was obtained, and the attack was successful. Figure 13(a) shows that the influence of faults is almost the same when the pulse width is changed. Figure 13(b) shows that the expected fault is obtained mainly at 250 to 350 V. However, above 300 V, the percentage of "no response" increases drastically. We fixed the width to 80 ns and employed the voltage of 250 to 290 V, which has less "no response" and more "expected fault." Table 4 shows the attack success rates for the optimized glitch parameters. For each glitch parameter, we performed our attack 100 times. In this experimental setup, we can expect a high attack success rate (expected fault) of about 30% to 40% at best.



(a) Pulse width. **(b)** Voltage.

Figure 13: Relationship between glitch parameters and observed fault influences and their number of occurrence.

Table 4: Fault intensity and probability of occurrence of each fault influence.

Width [ns]	Voltage [V]	No effect [%]	No response [%]	Unexpected fault [%]	Expected fault [%]
80	270	28	6	33	33
80	280	18	9	34	39
80	290	15	9	34	42

7 Countermeasure

This section proposes a software-based countermeasure against the proposed attack. Software-based countermeasures have advantages such as flexibility in making changes to devices and no additional hardware costs. However, existing countermeasures can only reduce the success rate of fault-injection attacks or increase the difficulty of the attacks. Therefore, hardware support has been considered essential for absolute countermeasures.

In this section, we briefly describe the issues associated with existing approaches. Next, we present the proposed countermeasure and then evaluate its attack resistance. Finally, the evaluation of the runtime overhead of the countermeasure is performed.

7.1 Existing Approaches

Memory encryption can prevent malicious applications from reading secret data. Keystone provides software-based encryption as a plugin for additional protection against physical attackers [LKS+20]. However, the encryption mechanism significantly impacts the execution speed. Executing protected instructions twice prevents a single instruction skip [YGS+16, WP17, MTW+18, BFP19] and raises the bar by requiring the attacker to have an advanced multiple fault injection capability. However, the attack is still possible in principle. Inserting a random delay makes it difficult for the attacker to identify the exact timing of fault injection [TSW16, WP17, MTW+18]. Even though the success rate of each trial decreases, the attack will succeed after repeated trials.

The proposed attack can evade certain major countermeasures, including protecting data with instruction duplication/triplication [YGS+16, MTW+18, BFP19], branch instructions or loop structures [NHH+17, PHBC17, WSUM19], and control flows with integrity checks

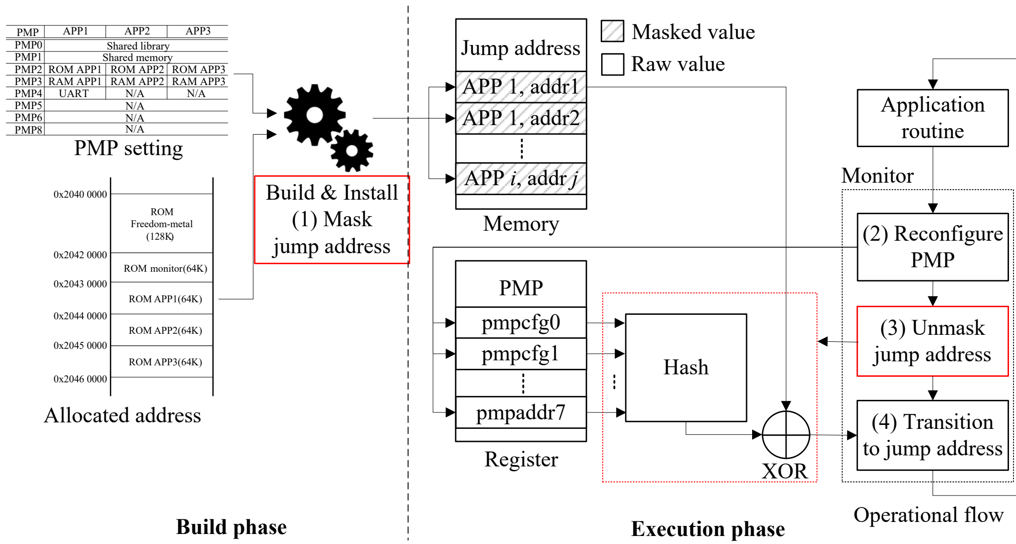


Figure 14: Operational flow of jump-address masking. Processes marked in red are added to the original build and execution process for countermeasure.

(i.e., CFI) [WP17, VTM+18, MTW+18, WSUM19]. This is because the proposed attack neither corrupts the data nor transfers the control flow to a malicious attack.

7.2 Proposed Countermeasure

This section formulates the proposed countermeasure and compares it with a similar one.

7.2.1 Overview

One of the key ideas in countermeasures against fault injection attacks is associating the process to be protected with the program control flow [NHH+17]. Therefore, the proposed countermeasure associates the validation of the PMP with the transition to an application. If an attacker maliciously changes the value of the PMP, the transition to the application will not be achieved, and the attack will fail. Therefore, it can be an absolute countermeasure, even if the attacker can skip multiple instructions.

To associate PMP validation with transitions to each application, we mask the jump addresses for each application, which are the entry point and return address. The operational flow of the proposed *jump-address masking*, in Figure 14, is divided into a build phase and an execution phase, with four processes. In the build phase, an executable file is generated and installed in the device. Then, (1) all jump addresses are masked based on the PMP setting and the address assigned to each application. More specifically, each jump address is XORed with the hash value of all the PMP registers (details are shown in section 7.2.2). In the execution phase, the installed application is run. When an application transition occurs, the CPU (2) reconfigures the PMP, (3) loads the masked jump address and unmasks it using the inverse masking procedure, and (4) jumps to the unmasked address.

7.2.2 Formulation

The list of variables for the proposed countermeasure is shown in Table 5 [Sec20, SiF21]. Masking ((1) in Figure 14), unmasking and hashing ((3) in Figure 14) can be formulated as follows:

Table 5: Variable definition for jump-address masking.

Variable	Description	Remarks
$addr_{jump}[i,j]$	Jump address for the jth address of application i.	$j=0$ indicates the entry point. If there are no other application calls, there is no return address; therefore $j=0$.
$addr_{mask}[i,j]$	Masked address corresponding to $addr_{jump}[i,j]$.	
$h[i]()$	Hash function for application i.	$h[i]() \in H, H$: Universal family. E.g., Toeplitz Hash.
N_{cfg}	Number of pmpcfg.	2 for X300 and 1 for Freedom U540.
N_{addr}	Number of pmpaddr.	8 for X300 and Freedom U540.
$pmpcfg_a[i]$	$pmpcfg_a$ for application i.	$0 \leq a \leq N_{cfg}-1$.
$pmpaddr_a[i]$	$pmpaddr_a$ for application i.	$0 \leq a \leq N_{addr}-1$.

Masking:

$$addr_{mask} = addr_{jump}[i,j] \oplus m[i]. \tag{1}$$

Unmasking:

$$addr_{jump} = addr_{mask}[i,j] \oplus m[i]. \tag{2}$$

Hashing:

$$\begin{aligned} m[i] &= h[i](x[i]), \\ x[i] &= x_1[i] \parallel ... \parallel x_k[i] \parallel ... \parallel x_{N_{cfg}+N_{addr}}[i], \\ \{x_1[i], ..., x_{N_{cfg}+N_{addr}}[i]\} &= \\ \{pmpcfg_0[i], ..., pmpcfg_{N_{cfg}-1}[i],\ &pmpaddr_0[i], ..., pmpaddr_{N_{addr}-1}[i]\}, \end{aligned} \tag{3}$$

where $\parallel$ represents the concatenation of bit sequences.
Hash function $h[i]()$ selection. $h[i]()$ selection is based on the following four requirements: 1) the function is selected after determining the PMP setting and allocated address; 2) a different function may be selected for each application i; 3) $addr_{mask}[i,j]$ must not point to the address allocated to application i, and 4) consider all the instruction skip patterns so that the tampered jump address will not point to the address allocated to application i. If requirements 3 and 4 are not met, the function must be reselected.

7.2.3 Feasibility Study on Universal Hashing

We show that requirements 3 and 4 in hash function selection are satisfied with high probability. The jump that results in a PMP reconfiguration is realized using `mret` to set the `mepc` register to the `pc` register. Thus, the space of the jump destination is equal to the bit width of the processor, n. The hash output, $m[i]$, and $addr_{jump}[i,j]$ tampered by the instruction skipping can be regarded as random numbers. The probability that an n-bit random number indicates an address space (m-bit) allocated to an application is

$$2^{m-n}, \tag{4}$$

where $n > m$. Therefore, the probability of satisfying requirement 3 when a certain hash function is selected is $p_1 = 1 - 2^{m-n}$. As described later in Section 7.3.2, the pattern of instruction skipping is $c = 2^{N_{cfg}+N_{addr}}$. Therefore, the probability that a hash function satisfies requirement 4 is $p_2 = (1-2^{m-n})^c$. Because c includes patterns without instruction skipping, p_2 also satisfies requirement 3. Therefore, the probability that a hash function

satisfies requirements 3 and 4 is $p = p_2$. Assuming Toeplitz hash function, we select a Toeplitz matrix with n rows, $m(N_{cfg} + N_{addr})$ columns, and $n + m(N_{cfg} + N_{addr}) - 1$ degree of freedom (number of bits in space). Therefore, the probability of searching the entire space and selecting a hash function that satisfies requirements 3 and 4 is

$$p_{success} = 1 - (1-p)^{2^{n+m(N_{cfg}+N_{addr})-1}}. \quad (5)$$

Case 1: PoC TEE. As shown in Appendix A.1, the PoC TEE provides each application with 64 KB (2^{16}) of code space. Thus, we have $n = 32, m = 16$, and $N_{cfg} + N_{addr} = 10$, $p = 0.984$ and $p_{success} \approx 1$.
Case 2: Keystone. Assume that the size of an enclave application is 128 MB (2^{27}) following the Intel SGX [CHKV19]. Thus, we have $n = 64, m = 27, N_{cfg} + N_{addr} = 9$, $p = 0.999$ and $p_{success} \approx 1$.

7.2.4 Related Work

The authors in [LNW+19] proposed a pointer authentication code (PAC) that prevents control flow hijacking and data corruption by authenticating pointers and return addresses based on hashes. In particular, it is similar to the proposed countermeasure in that code and data pointers are protected by associating them with hash values, and data pointers are calculated and stored during compilation. However, the expected values should be loaded into all the PMP registers to prevent the proposed attack, which is not possible with PAC. For example, even if the code pointer is protected, the PMP can be modified. In addition, the usage of the protected value is unmanaged, making the proposed attack successful by skipping the load instruction into the PMP. Furthermore, PAC protects the pointers by calculating hashes at runtime, but such an approach is likely invalidated by instruction skipping. Therefore, the proposed countermeasure associates data verification with control flow by jumping to unmasked addresses instead of detecting tampering by runtime calculation.

7.3 Attack Resistance of Proposed Countermeasure

In this section, we present and verify security claims.

7.3.1 Security Claims

The attacker model was inherited from Section 3.1. The requirements for the proposed countermeasure are as follows: 1) The proposed countermeasure is resistant to the proposed attack, which is **deterministically** successful through instruction skipping; 2) the attack is not successful even if the attacker knows the implementation of the TEE and countermeasure and can skip any such instructions multiple times; 3) the attack is not successful even if the attacker can control the address to which the application is allocated and the corresponding changes in the PMP value. Incidental successful attacks are out of scope, as in requirement 1, but such attack resistance is discussed in Section 8.4.

7.3.2 Evaluation

We present all possible attacks. Furthermore, we demonstrate that the proposed countermeasure is resistant to them.
Attack 1: Skip PMP reconfiguration. The proposed attack forces $x_k[i]$ to inherit the register value $x_k[i']$ from the previous application i'. Therefore, the mask value is

$$\begin{aligned} m'[i] &= h[i](x'[i]), \\ x'[i] &= x_1[i] \parallel ... \parallel x_k[i'] \parallel ... \parallel x_{N_{cfg}+N_{addr}[i]}. \end{aligned} \quad (6)$$

Table 6: Comparison of execution time (Arty A7).

Toeplitz hash	AES128 (enc. + dec.)	Context switch	
		Rewriting method	Switching method
127.75 µs	798.73 µs (80.00 + 718.73)	153.27 µs	110.97 µs

Thus, the address to jump to is

$$addr'_{jump}[i,j] = addr_{jump}[i,j] \oplus m[i] \oplus m'[i]. \quad (7)$$

Because $1 \leq k \leq N_{cfg} + N_{addr}$, the attacker can decide whether to skip or not for each $x_k[i]$; therefore, we define $c = 2^{N_{cfg}+N_{addr}}$ as the different instruction skipping patterns. However, from requirement 4 of hash function selection in Section 7.2.2, this attack is unsuccessful.

Attack 2: Skip hashing. The impact of this attack is implementation-dependent and is not handled in requirement 1.

Attack 3: Skip XOR operation for unmasking. This attack results in the following unmasking operation

$$addr_{jump}[i,j] = addr_{mask}[i,j]. \quad (8)$$

From requirement 3 of hash function selection, the attack is unsuccessful.

Attack 4: Manipulate allocated address. From assumption 6 in Section 3.1, the attacker can know $m[i]$ and $m'[i]$ in Eq. (6). Therefore, it is possible to manipulate $addr_{jump}[i,j]$ and $x_k[i]$ to acquire the desired $addr'_{jump}[i,j]$. However, based on requirement 1 of hash function selection, the hash function is newly selected based on the modified $addr_{jump}[i,j]$, making the attack unsuccessful.

7.4 Runtime Overhead

This section evaluates the runtime overhead of hash computation, which is the core of the proposed countermeasure. Furthermore, we compare the proposed countermeasure with memory encryption (cf. Section 7.1), which is a promising countermeasure against the proposed attack. Specifically, we implemented the Toeplitz hash as an example of a universal hash and used AES [Sma19] as an example of memory encryption, and also compared the time required for context switching in the PoC TEE and Keystone. The Arty A7 platform with X300 core and HiFive Unleashed were used for the evaluation.

Table 6 shows the execution time of the Toeplitz hash, AES, and context switch for the Arty A7 platform. The Toeplitz hash shows the result of the matrix operation on a 32×320 Toeplitz matrix (cf. Section 7.2.3) and a 320-bit input (10 registers from Table 5). AES shows the time required to encrypt and decrypt a 128-bit input. The context switch shows the time from the `ecall` to the start of the application. In all cases, the O2 option was used for optimization.

The Toeplitz hash is 6.25 times faster than AES128. In memory encryption, the number of operations increases as the data to be protected increases, whereas the proposed countermeasure is constant as long as the architecture (i.e., number of bits and PMP entries) remains the same. Therefore, it is a reasonable countermeasure against the proposed attack.

Meanwhile, when the hash calculation is added to the context switch, the rewriting and switching methods become 1.8 and 2.2 times slower, respectively. The acceptability of this overhead depends on the execution time of the application. As an example, the MultiZone SDK sample [Sec21] provides an execution time of 10 ms for one application.

Table 7: Comparison of execution time (HiFive Unleashed).

Toeplitz hash	AES128 (enc. + dec.)	Context switch			
		Run	Stop	Resume	Exit
42.22 µs	208.70 µs (32.02 + 176.68)	11.01 µs	3.43 ms	5.52 µs	2.74 µs

Compared to this, the hash calculation accounts for 1.3% of the total time, indicating no significant impact.

Table 7 shows the execution time for the HiFive Unleashed. Because the HiFive Unleashed is 64-bit architecture and has 9 registers for PMP (cf. Table 5), the Toeplitz hash shows the result of the matrix operation on a 64×576 Toeplitz matrix and a 576-bit input. The input size for AES is the same as the Arty A7 platform. The context switch shows the time required for run, stop, resume, and exit which corresponds to the steps (3) and (4), (5) and (6), (7) and (8), and (9) and (10) in Figure 12, respectively.

Similar to the results for the Arty A7 platform, the Toeplitz hash is 4.9 times faster than AES128. Meanwhile, the countermeasure makes the context switch 1.01 to 16.4 times slower. However, the processing time of Toeplitz hashing is much smaller than that of "stop". Furthermore, "stop" is an essential process for exchanging data between happ and eapp. Therefore, there is no significant impact from the perspective of context switching.

8 Discussion

This section further discusses the proposed attack and countermeasure.

8.1 Attack Applicability to TrustZone

ARM TrustZone is a well-known TEE-enabler technology for embedded devices. First, we describe the isolation mechanisms of TrustZone as in [ARM15, Yiu15, NMB+16, Yiu17, PS19, ARM19], and then show that the proposed attack can be partially applied to TrustZone-based TEEs. Here, we focus on the state-of-the-art TrustZone based on ARMv8-A (v8-A) and ARMv8-M (v8-M). The bare-metal implementation is assumed to be a RISC-V-based TEE, such as our PoC TEE. A detailed comparison between TrustZone and RISC-V-based TEE is shown in Appendix D.

TrustZone has the concept of the *world*, which divides CPU resources into secure and normal worlds, and isolates applications running in each world. Hereinafter, we refer to world-based isolation and isolation for applications as *world isolation* and *application isolation*, respectively. In v8-M, world and application isolations are realized by hardware units called memory protection unit (MPU) and software attribution unit (SAU), respectively. In v8-A, both isolations are realized by a hardware unit called the memory management unit (MMU). Therefore, MPU+SAU or MMU corresponds to the PMP in RISC-V. The usage of configuration values written to registers and memory to realize isolation is consistent with the PMP.

We can summarize the attack applicability as follows.

1. The proposed attack is applicable to application isolation in TrustZone because the hardware units and their configurations correspond to those of RISC-V. To perform the attack, we skip the reconfigurations of MPU and MMU in v8-M and v8-A, respectively.

2. The proposed attack is not applicable to world isolation in TrustZone. In v8-M, the SAU settings are not reconfigured after initialization. In v8-A, each world has its

MMU setting. Therefore, tampering with MMU settings in the normal world does not affect the secure world.

8.2 Attack Limitation

Although target instructions that change the PMP configuration can be identified and skipped, the success or failure of the attack depends on the implementation of each TEE.
The main determinants of success or failure are considered as follows.

- **Address-matching method in `pmpcfg`**: We mainly used NAPOT for the PoC TEE. In the rewriting method, NAPOT can be replaced with TOR using two PMP entries. The order of the PMP entries then differs from those shown in Tables 1 and 2. TOR covers the range between the two `pmpaddr`s; therefore, skipping the PMP configuration results in an increase or decrease in the target range. The former enables the attack to succeed, whereas the latter causes the attack to fail.
- **Order of PMP entry**: For example, if the order of PMP2 and PMP3 is reversed in the rewriting method, the access permissions of ROM and RAM are exchanged. ROM access is necessary to execute instructions for applications; thus, only the exchange of RAM for APP3 and ROM for APP2 is allowed. Although such an exchange breaks memory protection by the PMP, it also fails the original goal of obtaining the secret data. In contrast, an attack on the switching method would be successful even if the order of the PMP entries was changed because the attacker can obtain RAM permission for APP2 in addition to the original access permissions.
- **Calling other applications**: MultiZone adopts round-robin scheduling and allows each application (or zone) to run for a short time by controlling them with timer interrupts [Sec21]. Thus, attacker application cannot directly call a victim application. Therefore, the attacker can only target the application executed just before the attacker application. Meanwhile, Keystone allows untrusted host applications to invoke an enclave application at an arbitrary timing [LKS+20]. The proposed attack is applicable in such cases.

8.3 Applicability of Countermeasure

In this section, we show that the proposed countermeasure can be fully or partially applied to MultiZone and Keystone.
Prerequisites. The proposed countermeasure can be applied to: 1) bare-metal environments (e.g., PoC TEE or MultiZone), where the build process can be modified, and the allocated address for applications and PMP value are known, or 2) OS environments (e.g., Keystone), in which the address where an enclave application is deployed and PMP value are known.
MultiZone. MultiZone build tool is encrypted and is a black box for users; therefore, prerequisite 1 is not satisfied. However, because there is no technical problem, Hex Five Security, the provider of the tool, can implement our countermeasure.
Keystone. Since the address where the enclave application is deployed is managed by the OS, prerequisite 2 is not satisfied. Therefore, it is necessary to extend Keystone as follows: fix the address where enclave applications are deployed and set up a special region for enclave in the memory space. For this purpose, we have a table of IDs (indicating the number of enclave applications), raw addresses, and mask addresses. When deploying the application, the raw address is extracted from the ID, and the application is extracted. At the time of execution, after switching the PMP, the mask address is extracted from the ID, and the application jumps to the unmasked address according to the proposed countermeasure.

Although this approach can protect the entry point, it does not protect the return address. The approach for fully applying the countermeasure to Keystone and evaluating its resistance will be investigated in future work.

8.4 Resilience of Countermeasure to Other Attacks

Random jump attack. Even with the proposed countermeasure, an unexpected register corruption may cause a random jump with a modified PMP. If the jump destination points to the address where the attacker application is allocated, the attack code is executed. We refer to such an attack as a *random-jump attack*.

Although random-jump attacks are not covered by the proposed countermeasure, as shown in security claim 1, we discuss their success probability. From Eq. (4), the probability of a successful random-jump attack by corrupting a certain register is $1/2^{16}$ and $1/2^{37}$ for the PoC TEE and Keystone, respectively. Therefore, the proposed countermeasure alone is insufficient, and random-jump attacks should be protected by other countermeasures such as CFI [WSUM19].

Other bypassing attacks. The proposed countermeasure can protect against bypassing attacks on secure boots [VTM+18] and authentication [MTW+18, BFP19]. Because the hash value of the boot code is verified in the case of a secure boot, the transition address to the boot process can be masked by the expected hash value. In the authentication process, the transition address to the process after authentication can be masked by the expected value as a password or a response to a challenge.

9 Conclusion

We proposed an attack to bypass isolated execution realized by the PMP in RISC-V. Because the proposed attack targets the unique instructions required to construct TEEs, it applies to various RISC-V-based TEEs. We also proposed an attack scheme for determining the fault-injection parameters to conduct the proposed attack in a cross-device environment. The effectiveness of the attack scheme was demonstrated through experiments using a PoC TEE implemented with reference to existing RISC-V-based TEEs. The practicality of the proposed attack was also demonstrated by attacking Keystone. Furthermore, we proposed a software-based countermeasure that invalidates the proposed attack in principle.

From our experimental results (cf. Section 6) and discussion (cf. Section 8.2), we conclude that the rewriting method is relatively secure for fault-injection attacks. The switching method is easier to attack because the attacker can obtain permission from the victim RAM in addition to the original permissions. As mentioned in Section 8.2, this suggests that the attack should be effective even when the order of the PMP entries is changed.

We did not report the attack results to RISC-V-based TEE developers for two reasons. (1) There are obstacles to **real attacks**, and (2) TEE does **not** focus on invasive physical attacks such as fault-injection attacks. Although the attack on Keystone was successful, there are still challenges in applying the attack scheme (cf. Section 4). In addition, Keystone is still mainly used for research purposes. It should be noted that for a critical application requiring higher security, physical attacks should be considered, such as plugins provided by Keystone, even though TEE is generally intended to protect against software attacks, and most physical attacks are out of scope.

The following issues remain to be addressed in the future: (1) implementation of the proposed countermeasure and a demonstration of its attack resilience; (2) application of the attack scheme to Keystone; and (3) an evaluation of the proposed attack on TEEs based on another architecture such as ARM TrustZone (cf. Section 8.1).

References

[AAB+16] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[ARM15] ARM. ARM Cortex-A Series Programmer's Guide for ARMv8-A. https://developer.arm.com/documentation/den0024/a/, 2015. Accessed 5 July 2020.

[ARM16] ARM. Memory Protection Unit (MPU) Version 1.0. https://developer.arm.com/documentation/100699/0100/, 2016. Accessed 19 September 2021.

[ARM19] ARM. Learn the architecture: AArch64 Virtualization. https://developer.arm.com/documentation/102142/0100, 2019. Accessed 27 June 2021.

[BDL97] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

[BECN+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[BFP19] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 199–224, 2019.

[BGV11] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 105–114. IEEE, 2011.

[BRSK17] Swarup Bhunia, Sandip Ray, and Susmita Sur-Kolay. *Fundamentals of IP and SoC security*. Springer, 2017.

[BTG10] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 148–163. Springer, 2010.

[CHKV19] Somnath Chakrabarti, Matthew Hoekstra, Dmitrii Kuvaiskii, and Mona Vij. Scaling Intel® Software Guard Extensions Applications with Intel® SGX Card. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–9, 2019.

[ELG20] Mahmoud A Elmohr, Haohao Liao, and Catherine H Gebotys. EM Fault Injection on ARM and RISC-V. In *2020 21st International Symposium on Quality Electronic Design (ISQED)*, pages 206–212. IEEE, 2020.

[GA03] Sudhakar Govindavajhala and Andrew W Appel. Using Memory Errors to Attack a Virtual Machine. In *2003 Symposium on Security and Privacy, 2003.*, pages 154–165. IEEE, 2003.

[Gil15] Brett Giller. Implementing Practical Electrical Glitching Attacks. *Black Hat Europe*, 2015.

[Int20] RISC-V International. RISC-V International Members. https://riscv.org/members/, 2020. Accessed 15 Janualy 2021.

[KFG+20] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1445–1461. USENIX Association, August 2020.

[LBDPP19] Johan Laurent, Vincent Beroulle, Christophe Deleuze, and Florian Pebay-Peyroula. Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 252–255. IEEE, 2019.

[Lee21] Dayeol Lee. keystone-sdk. https://github.com/keystone-enclave/keystone-sdk/tree/master, 2021. Accessed 23 June 2021.

[LK18] Dayeol Lee and David Kohlbrenner. Welcome to Keystone Enclave's Documentation! http://docs.keystone-enclave.org/en/latest/index.html, 2018. Accessed 5 July 2020.

[LKS+20] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[LNW+19] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. {PAC} it up: Towards Pointer Integrity using {ARM} Pointer Authentication. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 177–194, 2019.

[MOG+20] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (S&P)*, 2020.

[MTW+18] Alyssa Milburn, Niek Timmers, Nils Wiersma, Ramiro Pareja, and Santiago Cordoba. There Will Be Glitches: Extracting and Analyzing Automotive Firmware Efficiently. *Black Hat USA*, 2018.

[NHH+17] Shoei Nashimoto, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji, and Takafumi Aoki. Buffer overflow attack with multiple fault injection and a proven countermeasure. *Journal of Cryptographic Engineering*, 7(1):35–46, 2017.

[NMB+16] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone Explained: Architectural Features and Use Cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451. IEEE, 2016.

[PHBC17] Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. Compiler-Assisted Loop Hardening Against Fault Attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(4):1–25, 2017.

[PS19] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys (CSUR)*, 51(6):130, 2019.

[PT17] Jungmin Park and Akhilesh Tyagi. Using Power Clues to Hack IoT Devices: The power side channel provides for instruction-level disassembly. *IEEE Consumer Electronics Magazine*, 6(3):92–102, 2017.

[PW17] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017.

[PXJ+18] Jungmin Park, Xiaolin Xu, Yier Jin, Domenic Forte, and Mark Tehranipoor. Power-based Side-Channel Instruction-level Disassembler. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[QWLQ19a] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching Trustzone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2019.

[QWLQ19b] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–6. IEEE, 2019.

[RRR+04] Srivaths Ravi, Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. Security in Embedded Systems: Design Challenges. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):461–491, 2004.

[SBO+15] Daehyun Strobel, Florian Bache, David Oswald, Falk Schellenberg, and Christof Paar. SCANDALee: A Side-ChANnel-based DisAssembLer using Local Electromagnetic Emanations. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 139–144. IEEE, 2015.

[Sec19] Hex Five Security. MultiZone API. https://github.com/hex-five/multizone-api, 2019. Accessed 5 July 2020.

[Sec20] Hex Five Security. X300. https://github.com/hex-five/multizone-fpga, 2020. Accessed 5 July 2020.

[Sec21] Hex Five Security. multizone-sdk. https://github.com/hex-five/multizone-sdk, 2021. Accessed 21 July 2021.

[SiF20] SiFive. Freedom Metal Machine Compatibility Library. https://github.com/sifive/freedom-metal, 2020. Accessed 5 July 2020.

[SiF21] SiFive. SiFive FU540-C000 Manual v1p4. https://sifive.cdn.prismic.io/sifive/d3ed5cd0-6e74-46b2-a12d-72b06706513e_fu540-c000-manual-v1p4.pdf, 2021. Accessed 2 October 2021.

[Sma19] SmarterDM. micro-aes. https://github.com/SmarterDM/micro-aes, 2019. Accessed 5 July 2020.

[TM17] Niek Timmers and Cristofaro Mune. Escalating Privileges in Linux using Voltage Fault Injection. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–8. IEEE, 2017.

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, 2017.

[TSW16] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM using Fault Injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. IEEE, 2016.

[VBOM+19] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1741–1758, 2019.

[VTM+18] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adele Morisset, and Sebastien Ermeneux. Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot. *IEEE Transactions on Computers*, 2018.

[VWG07] Dennis Vermoen, Marc Witteman, and Georgi N Gaydadjiev. Reverse Engineering Java Card Applets Using Power Analysis. In *IFIP International Workshop on Information Security Theory and Practices*, pages 138–149. Springer, 2007.

[WA19] Andrew Waterman and Krste Asanovic. The RISC-V Instruction Set Manual Volume II: Privileged Architecture. https://riscv.org/specifications/privileged-isa, 2019. Accessed 5 July 2020.

[WP17] Nils Wiersma and Ramiro Pareja. Safety!= Security: On the resilience of ASIL-D certified microcontrollers against fault injection attacks. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 9–16. IEEE, 2017.

[WSUM19] Mario Werner, Robert Schilling, Thomas Unterluggauer, and Stefan Mangard. Protecting RISC-V Processors against Physical Attacks. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1136–1141. IEEE, 2019.

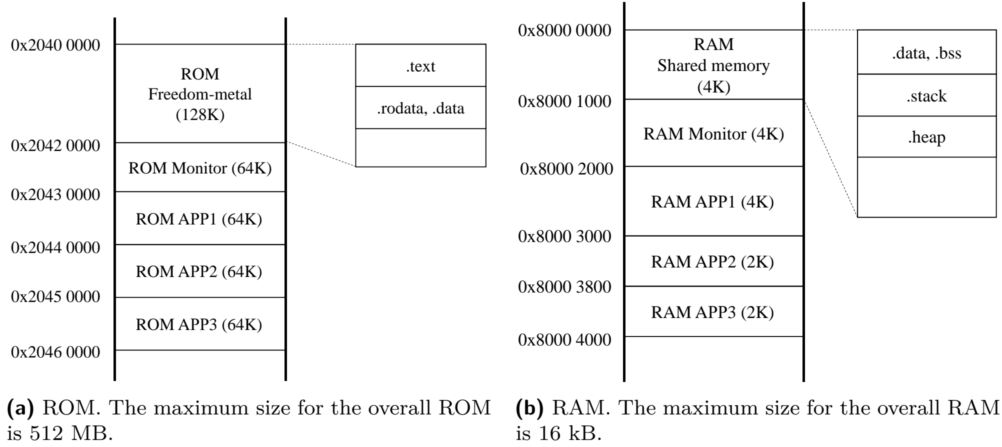
[YGS+16] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. Software Fault Resistance is Futile: Effective Single-Glitch Attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 47–58. IEEE, 2016.

[Yiu15] Joseph Yiu. ARMv8-M architecture technical overview. *ARM WHITE PAPER*, 2015.

[Yiu17] Joseph Yiu. Software Development in ARMv8-M Architecture. *embedded world 2017*, 2017.

[YSW18] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. *Journal of Hardware and Systems Security*, 2(2):111–130, 2018.

[YUZP19] Baki Berkay Yilamz, Elvan Mert Ugurlu, Alenka Zajic, and Milos Prvulovic. Instruction Level Program Tracking Using Electromagnetic Emanations. In *Cyber Sensing 2019*, volume 11011. International Society for Optics and Photonics, 2019.



(a) ROM. The maximum size for the overall ROM is 512 MB.

(b) RAM. The maximum size for the overall RAM is 16 kB.

Figure 15: Memory map.

A Specification and Implementation of PoC TEE

A.1 Memory Map

The memory maps indicating the ROM and RAM for the PoC TEE are shown in Figures 15(a) and (b).

A.2 Specification of Shared Memory

The shared memory, shown in Figure 16, separates its memory region into two sub-regions: one for application calls (0–11) and one for shared data (12–127). It is declared as a 128-byte array of `uint8_t`. `SP` and `RA` denote registers for stack pointers and return addresses, respectively. The monitor uses `caller ID` and `callee ID` to manage application calls. The monitor saves the context of the application with the caller ID and calls an application with the callee ID. `Cmd` (command) is used to determine the operation of each application. An example of `Cmd` usage is presented in Section A.6.

A.3 Function for Switching Applications

The function for switching applications involves the following three steps: (1) store `sp` and `ra` into the shared memory, (2) store caller ID and callee ID into the shared memory, and (3) transfer the control to the monitor in M-mode by an environment call exception caused by `ecall`. The original code of the function is as follows.

```
1  void call_app(uint8_t caller_id, uint8_t callee_id){
2    uintptr_t sp, ra;
3    uintptr_t *t;
4    __asm__ volatile ("mv %0, sp" : "=r"(sp));
5    __asm__ volatile ("mv %0, ra" : "=r"(ra));
6
7    t = (uintptr_t)&shared_buffer[SHARED_SP];
8    *t = sp; // [0:3]
9    t = (uintptr_t)&shared_buffer[SHARED_RA];
10   *t = ra; // [4:7]
11   shared_buffer[SHARED_CALLER] = caller_id;
12   shared_buffer[SHARED_CALLEE] = callee_id;
13   __asm__ volatile("ecall");
14 }
```

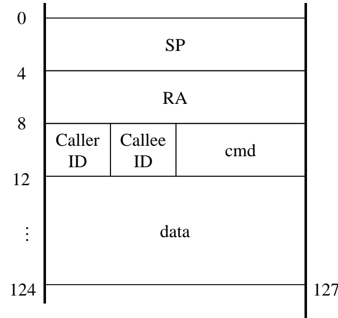


Figure 16: Structure of shared memory.

Table 8: Specification of commands for APP1.

Cmd	Class	Function	Size	Description
0x80	0x10	0x10	N/A	Echo cmd
		0x20	N/A	Get response
		0x30	size	Set buffer
	0x20	0x10	size	Set shared memory
		0x18	size	Get shared memory
		0x20	N/A	Call APP2
		0x30	N/A	Call APP3

A.4 Specification of Commands for APP1

APP1 acts a dispatcher and executes commands from users via UART. Table 8 summarizes the specifications of the commands. The command is given by a 4-byte array of `uint8_t`, and the bytes are interpreted as `Cmd`, `Class`, `Function`, and `Size`, respectively.

A.5 Implementation of APP2

APP2 is a victim application executing AES encryption, which is implemented with *micro-aes* [Sma19]. APP2 receives plaintext using the shared memory and then encrypts it. Finally, it stores the corresponding ciphertext in the shared memory. A user can obtain the ciphertext via APP1. A secret key used for encryption is declared as a static variable, and it is copied to the RAM. The original code of APP2 is as follows:

```
1  #define AES_BLOCK_SIZE 16
2
3  aes_128_context_t ctx;
4
5  static uint8_t key[AES_BLOCK_SIZE] = {0x00, 0x01, 0x02, 0x03,
6                                        0x04, 0x05, 0x06, 0x07,
7                                        0x08, 0x09, 0x0a, 0x0b,
8                                        0x0c, 0x0d, 0x0e, 0x0f};
9
10 void sep2_main() {
11   int i;
12   uint8_t block[AES_BLOCK_SIZE] = {0};
13
14   aes_128_init(&ctx, key);
15   for (i = 0; i < AES_BLOCK_SIZE; i++) {
16     block[i] = shared_buffer[SHARED_DATA + i];
17   }
18
19   aes_128_encrypt(&ctx, block);
20
21   for (i = 0; i < AES_BLOCK_SIZE; i++) {
22     shared_buffer[SHARED_DATA + i] = block[i];
23   }
24
25   call_app(CTX_SEP2, CTX_END); // finish
26 }
```

A.6 Implementation of APP3

APP3 is an attacker application that dumps the victim RAM. More specifically, it receives a base address and an offset, and then reads data from the address of ”base adder + offset.” During the execution of an attack (`case 0xf1`), APP3 stores the necessary data in the shared memory in case it loses access to its own RAM (cf. Section 5.3.2). The original code of APP3 is as follows:

```
1   void sep3_main() {
2     // for call_app() without using stack (RAM)
3     uintptr_t sp, ra;
4     uintptr_t *t;
5
6     uint8_t cmd[2];
7     uint16_t offset;
8     uint32_t reg_val;
9     static uint32_t base_addr = BASE_ADDR;
10
11    *((uint16_t *)cmd) = *((uint16_t *)&shared_buffer[SHARED_CMD]);
12
13    switch(cmd[0]) {
14    case 0x33:
15      switch(cmd[1]) {
16      // --- set base addr ---
17      case 0x10:
18        base_addr = *(uint32_t *)&shared_buffer[SHARED_DATA];
19        break;
20
21      // --- set offset & load data (without fault injection) ---
22      case 0x20:
23        offset = *(uint16_t *)&shared_buffer[SHARED_DATA];
24        reg_val = *((uint32_t *)(base_addr + offset));
25
26        *((uint32_t *)&shared_buffer[SHARED_DATA]) = reg_val;
27        break;
28
29      // --- exploit with fault ---
30      case 0xf1:
31        // TIP: save offset value from APP3 RAM to shared RAM
32        offset = *(uint16_t *)&shared_buffer[SHARED_DATA];
33        *(uint16_t *)&shared_buffer[SHARED_DATA+STR_ADDR] = offset;
34
35        call_app(CTX_SEP3, CTX_SEP2);  // <- fault here
36        // TIP: ecall without call_app() to avoid using stack
37        __asm__ volatile ("mv %0, sp" : "=r"(sp));
38        __asm__ volatile ("mv %0, ra" : "=r"(ra));
39
40        t = (uintptr_t)&shared_buffer[SHARED_SP];
41        *t = sp; // [0:3]
42        t = (uintptr_t)&shared_buffer[SHARED_RA];
43        *t = ra; // [4:7]
44        shared_buffer[SHARED_CALLER] = CTX_SEP3;
45        shared_buffer[SHARED_CALLEE] = CTX_SEP2;
46        __asm__ volatile("ecall");
47
48        // TIP: 16-byte memory access without using stack
49        *((uint32_t *)&shared_buffer[SHARED_DATA+0]) = *((uint32_t *) (
              BASE_ADDR_TGT + *((uint16_t *)&shared_buffer[SHARED_DATA + STR_ADDR
              ])+0));
50        *((uint32_t *)&shared_buffer[SHARED_DATA+4]) = *((uint32_t *) (
              BASE_ADDR_TGT + *((uint16_t *)&shared_buffer[SHARED_DATA + STR_ADDR
              ])+4));
51        *((uint32_t *)&shared_buffer[SHARED_DATA+8]) = *((uint32_t *) (
              BASE_ADDR_TGT + *((uint16_t *)&shared_buffer[SHARED_DATA + STR_ADDR
              ])+8));
52        *((uint32_t *)&shared_buffer[SHARED_DATA+12]) = *((uint32_t *) (
              BASE_ADDR_TGT + *((uint16_t *)&shared_buffer[SHARED_DATA + STR_ADDR
              ])+12));
53        // TIP: return without sp use
54        shared_buffer[SHARED_CALLER] = CTX_SEP3;
55        shared_buffer[SHARED_CALLEE] = CTX_END;
56        __asm__ volatile("ecall");
57        break;
58
59      default:
60        break;
```

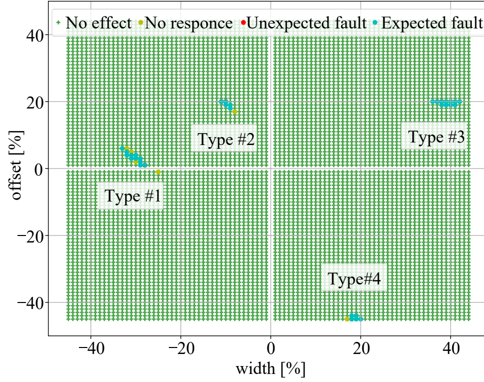
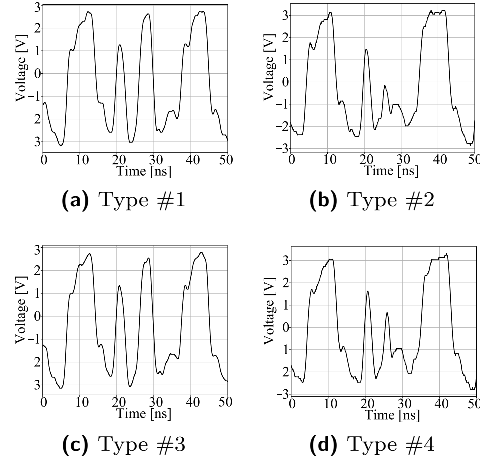


Figure 17: Profiling result for the exploitation device.



(a) Type #1 **(b)** Type #2 **(c)** Type #3 **(d)** Type #4

Figure 18: Successful glitches: (a) width=-32, offset=6, (b) width=-10, offset=20, (c) width=41, offset=21, (d) width=19, offset=-45. The clock signal split into two at the central position is a glitch.

```
61          }
62          break;
63
64        default:
65          break;
66        }
67
68        call_app(CTX_SEP3, CTX_END); // finish
69    }
```

B Supplement for Experiment #1

B.1 Fault Intensity for Target Device

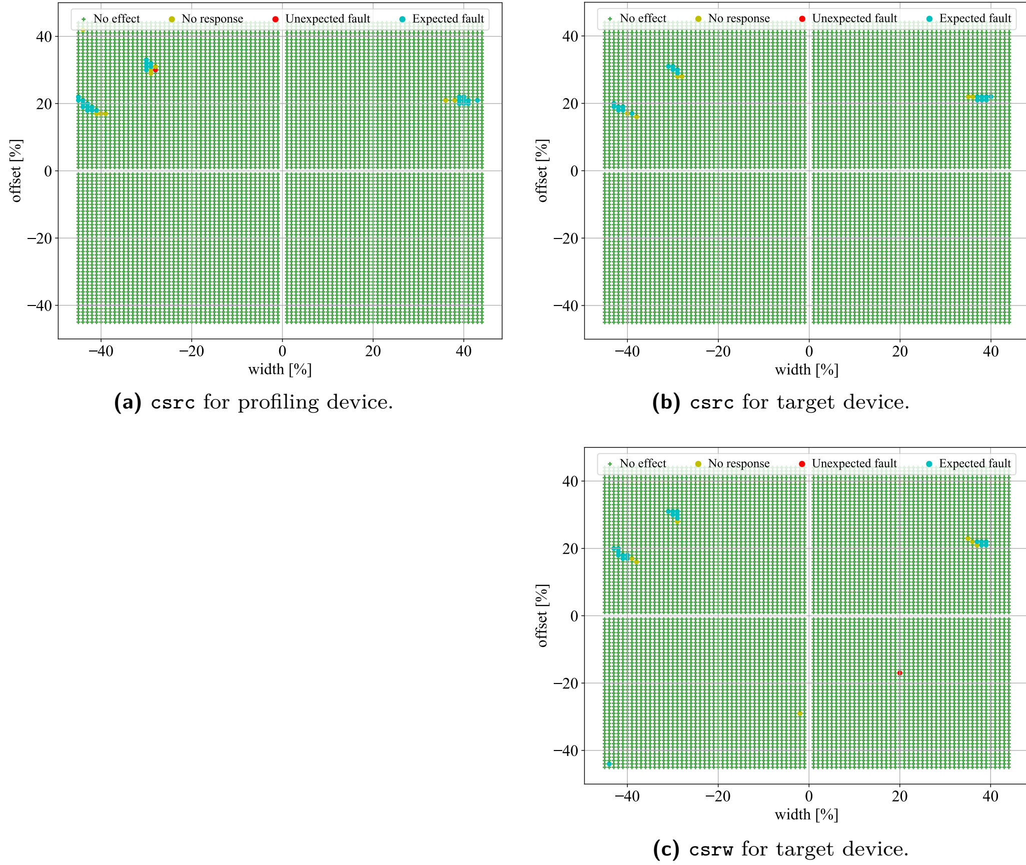
We performed the same profiling experiment using the target device. Figure 17 verifies the effectiveness of the method for extracting the fault intensity using a profiling device and suggests trends similar to that in Figure 8.

Successful clock glitches are divided into four types, as shown in Figure 17. Figure 18 shows representative waveforms for each type. They indicate that pairs of types #1 and #3 and types #2 and #4 show similar trends.

B.2 Fault Intensity for `csrc`

This section describes the profiling of `csrc` and clarifies the difference in fault sensitivity between `csrc` and `csrw`. We performed the same profiling experiments for `csrc` as in Section 5.2.1. Figure 19 shows the fault sensitivity of `csrc` in the profiling device and the target device, as well as the fault sensitivity of `csrw` in the target device[6]. For detailed comparison, Table 9 shows the parameters for which an expected fault was obtained with a probability of more than 60%.

[6]The result of `csrw` is the same experiment as Figure 17, however since it has been a long time since the previous experiment, it was re-experimented for comparison under the same conditions.



(a) `csrc` for profiling device.

(b) `csrc` for target device.

(c) `csrw` for target device.

Figure 19: Profiling results for `csrc`. Result for `csrw` is for reference.

Figures 19(b) and (c) show that the fault sensitivity of `csrc` and `csrw` are similar. Furthermore, Figures 19(a) and (b) show that for `csrc`, the fault sensitivity is similar for the profiling device and the target device. Table 9 shows that there is a difference between `csrc` and `csrw` in terms of attack success rate. Therefore, as shown in the experiment in section 5.4.1, the parameters for which the expected fault was obtained by the profiling device were not sufficient for the attack. However, the similarity in fault sensitivity shown in Figure 19 indicates that we can attack both `csrc` and `csrw` by extending the range of fault parameters obtained by profiling either `csrc` or `csrw`.

B.3 Template Matching

This section describes the details of steps (2)–(4) of the attack scheme shown in Figure 4. More specifically, we explain the template creation and matching methods and evaluate the accuracy of template matching.

B.3.1 Creating Side-Channel Template

We created side-chennel templates for each target instruction, that is, `csrw`, `csrc`, and `csrs`, following the steps below.

1. **Extract target waveform.** We ran the PoC TEE, including the target instructions

Table 9: Fault intensity parameters and their success rates with 10 trials.

Fault intensity [%]		Success rate [%]		
Width	Offset	`csrc` (profiling device)	`csrc` (target device)	`csrw` (target device)
-42	18	60	20	20
-41	18	10	60	70
-29	30	0	0	90
-29	31	80	0	10
38	22	0	60	40
40	21	80	0	0

on the profiling device, and acquired EM waveforms. The average of 100 waveforms was used as the target waveform.

2. **Extract reference waveform.** We replaced the target instruction with `nop` (no operation) and acquired EM waveforms. As in step (1), the average of 100 waveforms was used as the reference waveform.

3. **Extract side-channel template.** We calculated the difference between the target and reference waveforms and identified the execution timing of the target instruction. We then cut out the region around the identified position from the target waveform as templates.

Figures 20(a)–(c) show the waveforms for each step above. In the difference waveform in Figure 20(c), the target instruction was executed in the area where the difference is large, that is, spikes are observed. Therefore, eight spike regions were extracted as templates. In the template matching, all candidates were examined and the one with the highest matching score was adopted.

B.3.2 Matching Templates with Trace

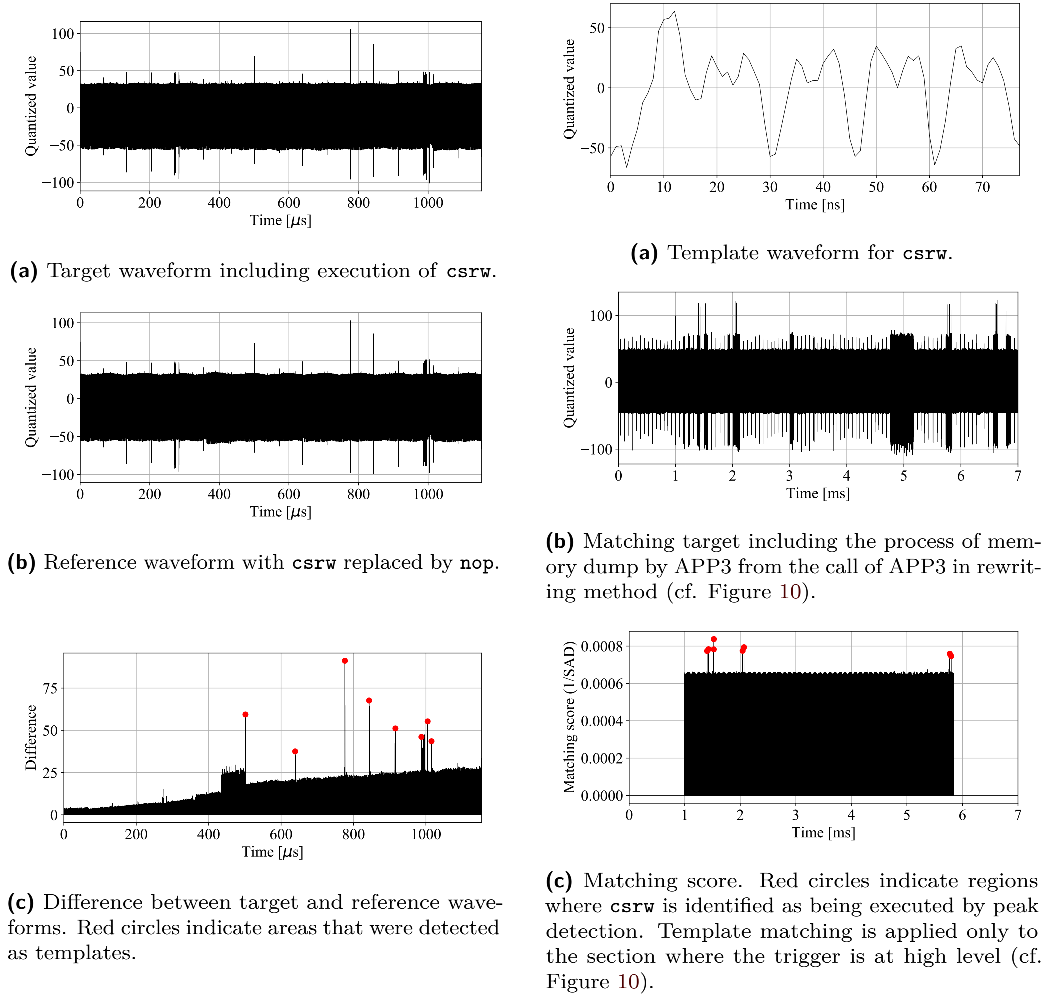
The sum of absolute difference (SAD), one of the simplest matching algorithms, matched the side-channel templates with the side-channel traces. From the viewpoint of visibility, the matching score was set to 1/SAD. Therefore, the higher the matching score, the more the target instructions are executed.

Figure 21(c) shows the results of matching with the `csrw` template (Figure 21(a)) against the waveform in which the attacker application was run in the PoC TEE with the rewriting method on the target device (Figure 21(b)). The template was cut out for five clocks from the candidate templates, taking into account that the X300 core has a five-stage pipeline [Sec20]. The average of 100 waveforms was used as the matching target. Figure 21(b) shows that eight high matching scores are observed in the form of spikes (some spikes overlap). The same matching was performed for the `csrc` and `csrs` templates, and the spike locations are summarized in Figure 9(a) (cf. Section 5.2.2). Similarly, the same process was applied to the switching method, and the results are shown in Figure 9(b).

B.3.3 Matching Accuracy

The execution timing of the target instructions identified by template matching was compared with the true execution timing. To acquire the true execution timing, GPIO control instructions were inserted into the target code[7]. Figures 22(a) and (b) show

[7] To avoid changing the fault-injection timing owing to the insertion of GPIO control instructions, these instructions were left inserted throughout the experiments.

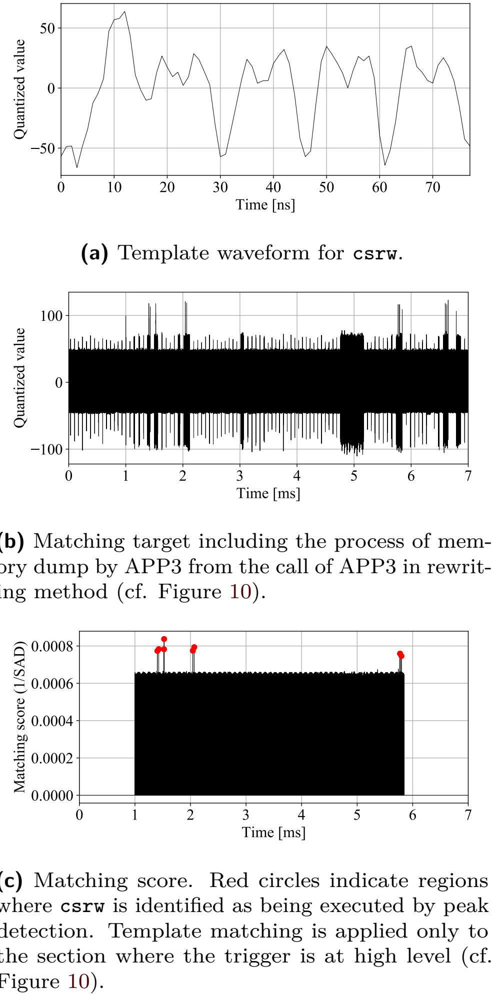


(a) Target waveform including execution of `csrw`.

(b) Reference waveform with `csrw` replaced by `nop`.

(c) Difference between target and reference waveforms. Red circles indicate areas that were detected as templates.

Figure 20: Side-channel template creation process.



(a) Template waveform for `csrw`.

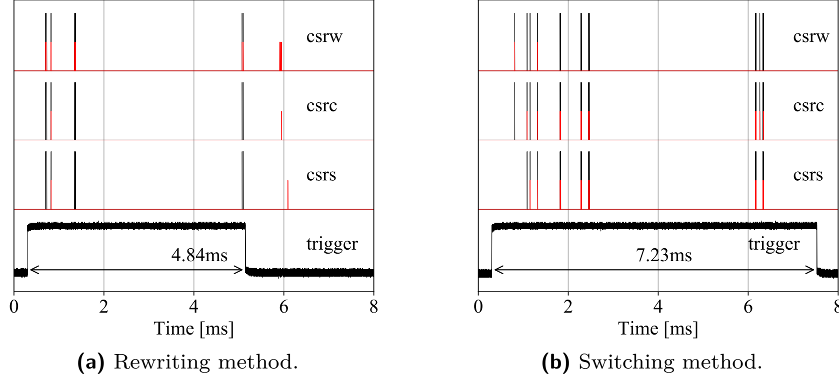
(b) Matching target including the process of memory dump by APP3 from the call of APP3 in rewriting method (cf. Figure 10).

(c) Matching score. Red circles indicate regions where `csrw` is identified as being executed by peak detection. Template matching is applied only to the section where the trigger is at high level (cf. Figure 10).

Figure 21: Template matching process.

the true execution timing of the target instructions and execution timing identified by template matching in rewriting and switching methods, respectively. The results show the occurrence of false positives, but no false negatives. Therefore, although the search space is increased, the successful fault-injection timing of the attack can be identified by testing all the candidates.

Based on Figures 22(a) and (b), the identified clock cycles are summarized in Tables 10(a) and (b), respectively. The number of cycles obtained by template matching is at most ± 50 cycles, which is different from that obtained from the GPIO signal. Therefore, the proposed method can effectively identify the execution timing of target instructions.



(a) Rewriting method. **(b)** Switching method.

Figure 22: Identified and true execution timing. Black and red lines show the results from template matching and GPIO, respectively.

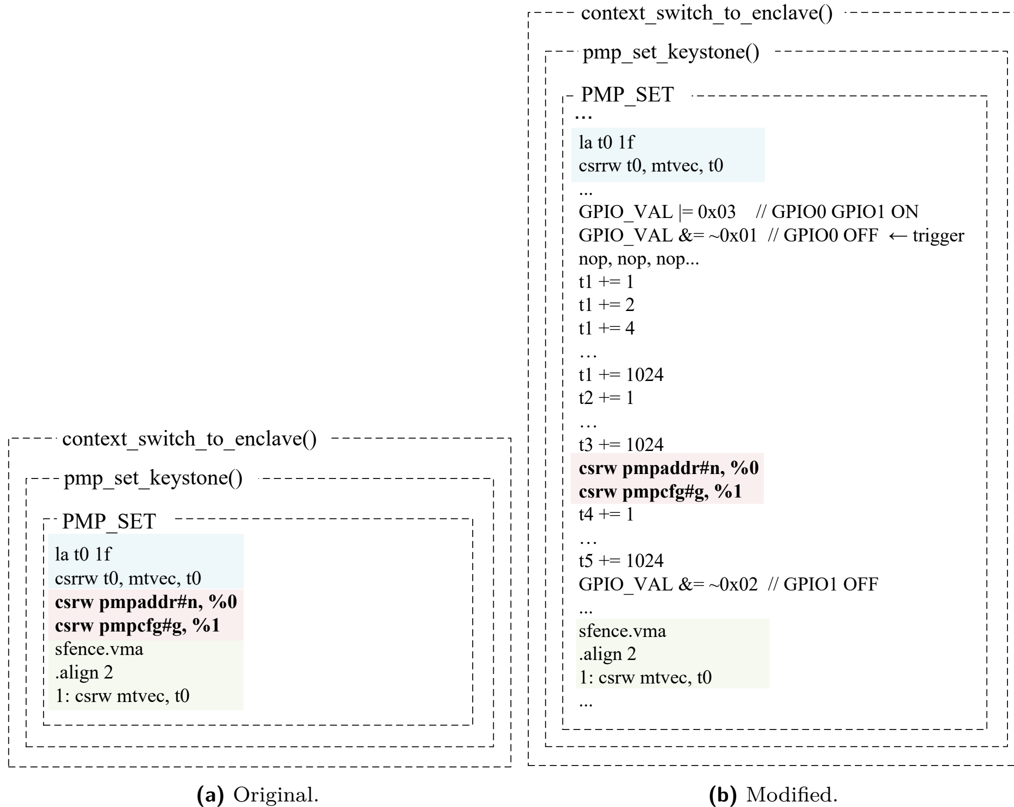
Table 10: Comparison of identified clock cycles.

(a) Rewriting method.

csrw			csrc			csrs		
Identified	GPIO	Diff.	Identified	GPIO	Diff.	Identified	GPIO	Diff.
26231	26239	8	26235	-	-	26260	-	-
27855	27862	7	27859	-	-	27884	-	-
34016	34032	16	34020	-	-	34045	-	-
34104	-	-	34108	34121	13	34182	34168	-14
67772	67798	26	67776	-	-	67801	-	-
69395	69402	7	69399	-	-	69424	-	-
310164	310190	26	310168	-	-	310193	-	-
311788	311793	5	311792	-	-	311817	-	-

(b) Switching method.

csrw			csrc			csrs		
Identified	GPIO	Diff.	Identified	GPIO	Diff.	Identified	GPIO	Diff.
33117	33141	24	33121	-	-	-	-	-
50991	-	-	50995	51002	7	51020	-	-
55345	-	-	55349	-	-	55374	55350	-24
66127	66156	29	66130	-	-	-	-	-
66214	-	-	66268	66226	-42	66293	66284	-9
98627	-	-	98680	98647	-33	98705	98683	-22
99459	-	-	99512	99468	-44	99537	99515	-22
128685	-	-	128738	128702	-36	128763	128739	-24
129514	-	-	129567	129525	-42	129592	129568	-24
139876	-	-	139830	139853	23	139855	139899	44
140996	-	-	140950	140955	5	140975	141021	46
380740	-	-	380694	380697	3	380719	380748	29
381806	-	-	381860	381817	-43	381835	381880	45
387239	-	-	387243	387263	20	-	-	-
391593	-	-	391597	-	-	391622	391598	-24
392662	-	-	392715	392673	-42	392740	392716	-24



(a) Original. **(b)** Modified.

Figure 23: Target code for Keystone.

C Target Instructions for Keystone

Figure 23 shows the target instructions for Keystone. Figure 23(b) shows the target code, which is a modified version of the original code shown in Figure 23(a). PMP reconfiguration is realized by `PMP_SET`, which is executed by Keystone SM calling the `context_switch_to_enclave()` and `pmp_set_keystone()`. Temporary registers, `t1` to `t5`, indicate the instruction that introduced a fault by assigning different values to each bit and serves as buffers for filling the lag between the trigger and injection of faults.

The only target instruction is `csrw pmpcfg`. This is because, as shown in Figure 12, the Keystone context switch does not change the address, but only the access attributes. Therefore, there is no effect even if `csrw pmpaddr` is skipped. For this reason, the experiment in Section 6.2.2 can also be regarded as a profiling experiment to investigate the fault intensity that can skip the `csrw` on the HiFive Unleashed.

Table 11: Comparison of RISC-V and TrustZone

Item	RISC-V	TrustZone v8-M	TrustZone v8-A
How to isolate	Check permissions at memory access.		Check permissions at address translation by MMU.
# of world	Any (PMP entries are at most 16)	2 (secure and non-secure (S/NS) states)	2 (secure and normal)
# of application		Any (MPU has up to 16 regions)	Any
Hardware units	PMA, PMP	IDAU, SAU, MPU	MMU
Privilege	M, H, S, U	Handler Mode (Privileged), Thread Mode (Privileged/Unprivileged)	EL3, EL2, EL1, EL0
How to configure world isolation	Configure PMP in M-mode.	Configure SAU from code in secure region.	Configure MMU (i.e., translation tables) in EL3 mode
How to configure application isolation		Configure MPU in Privileged mode.	Configure MMU in EL1 or EL2 modes
World switch	Exceptions transfer the control from U/S-mode to M-mode. M-mode reconfigures PMP and then returns the control to U/S-mode by `mret`.	Code in NS calls NSC function and then moves to S. Code in S calls a callback function and then moves to NS.	`SMC` instruction, exceptions, or interrupts, such as IRQ and FIQ, transfer the control from normal to secure. Secure returns to normal by `ERET`.
Application switch		Interrupts, such as IRQ and FIQ, transfers the control from non-privileged to privileged modes. Privileged mode reconfigures MPU and then returns to non-privileged mode.	Interrupts, such as IRQ and FIQ, transfers the control from EL0 to EL1 or EL2 modes. EL1 or EL2 mode reconfigures MMU and then returns to EL0 mode by `ERET`

Abbreviations

PMA: Physical Memory Attribute, IDAU: Implementation Defined Attribution Unit, MMU: Memory Management Unit, SAU: Software Attribution Unit, MPU: Memory Protection Unit, EL: Exception Level, NSC: Non-Secure Callable, IRQ: Interrupt ReQuest, FIQ: Fast Interrupt reQuest

D Comparison with TrustZone

Table 11 summarizes the comparison of TrustZone-based TEE and RISC-V-based TEE according to [ARM15, Yiu15, ARM16, NMB+16, Yiu17, PS19, ARM19]. RISC-V uses the PMP for both world isolation and application isolation; therefore, there is no separation in the column of RISC-V in Table 11. The major features of the TrustZone are as follows.

Hardware unit: The relation of PMA[8] and PMP in RISC-V corresponds to that of IDAU and SAU in v8-M. MPU in v8-M provides isolation based on the base address, size, and attribute, which is similar to PMP. Meanwhile, MPU is different from PMP in that MPU is defined in each world. MMU in v8-A has a richer function than MPU in v8-M in the sense that MMU can translate a virtual address into a physical address.

Privilege: The privileges in v8-A are defined as EL3 for secure monitor, EL2 for hypervisor, EL1 for OS, and EL0 for applications, which are the same as in RISC-V. Meanwhile, the privileges in v8-M are defined as handler and thread modes, which are different from RISC-V.

Configuration of world/application: TrustZone v8-M and v8-A perform the configuration of world(s) with SAU and MMU, and then perform the configuration of application(s) with MPU and MMU, respectively. The world configuration in v8-M does not change after initialization, whereas the application configuration can be changed. In v8-A, each world has its own translation tables for MMU, and they can be changed. Hence, the same virtual

[8]PMA is a H/W-defined unit for providing memory protection similarly to PMP.

address is translated to other physical addresses in each world.
World/application switch: Application switches resemble each other, although the world switch is different from the application switch. The world switch in v8-M employs a specific function called NSC to move the world from NS to S. Then, it employs a callback function located in the NS world to return from S to NS. The world switch in v8-A employs the `SMC` instruction, exceptions, or interrupts to move the world from normal to secure. Then, it employs the `ERET` instruction to return from secure to normal. This is similar to the operation of the monitor in RISC-V.