Software countermeasures against the multiple instructions skip fault model

Vanthanh Khuat, Jean-Max Dutertre, Jean-Luc Danger

Software Countermeasures Against the Multiple Instructions Skip Fault Model

Vanthanh Khuat[a,c,*], Jean-Max Dutertre[b], Jean-Luc Danger[a]

[a] *LTCI, Télécom Paris, Institut polytechnique de Paris, France*
[b] *Mines Saint-Etienne, CEA, Leti, Centre CMP, F - 13541 Gardanne France*
[c] *Faculty of Information Technology, Le Quy Don Technical University, Hanoi, Vietnam*

Abstract

In this work, we proposed two software countermeasures (CMs) for the detection of multiple instructions skips caused by Fault Injection (FI). The first CM is based on code duplication and uses a hardware dedicated counter. The implementation of this method consists in the duplication of instructions previously turned into an idempotent form and the insertion of dedicated instructions incrementing a hardware counter in between the groups of duplicated instructions. The second CM is based on the insertion of Sensitive instruction (SI)s into a block of instructions as sensors of instruction skips. The SI is chosen based on the observed Fault Model (FM) at bit level. We experimentally validated the effectiveness of the two CMs in a 32-bit Microcontroller Unit (MCU) using Laser Fault Injection (LFI) and Electromagnetic Fault Injection (EMFI). First, the skip of multiple instructions was obtained with a fault rate of 100%. The FM at bit level was identified to be bit-reset rather than bit-set. Second, we carried out LFI and EMFI experiments to the protected codes to validate the effectiveness of the CMs. In both cases, the results showed that the proposed methods are effective to detect multiple instructions skip faults.

Keywords: Fault Injection; Laser Fault Injection; Electromagnetic Fault Injection; Microcontroller; Instruction Skip; Software Countermeasure

*Corresponding author: Vanthanh Khuat, email:van-thanh.khuat@lqdtu.edu.vn, Faculty of Information Technology, Le Quy Don Technical University, Hanoi, Vietnam

1. Introduction

MCUs carry plenty of valuable information such as password, account number, identity, critical data, etc. Attackers have hence turned their attention to study how to attack and steal the information they contain. One of the most powerful attack techniques that poses a significant threat to MCU security is FI.

FI is an active attack technique in which the attacker uses a physical tool to disturb the target, inducing faults or errors into the target for the purpose of extracting the secret information they may contain. The most common techniques used for FI attack are: Clock or voltage tampering [1, 2, 3], Electromagnetic FI (EMFI) [4, 5, 6, 7], and Optical FI [8, 9, 10].

Recently, many researches have been conducted to study the threat that FI poses to the MCU. In [11], the authors reported the EMFI-induced skip of instructions and used it to bypass a verify PIN function, allowing a potential adversary to access the system without being authorized. In [12], the authors used EMFI for zeroing and setting an Advanced Encryption Standard (AES) key, which allows an attacker to retrieve the plain text without having the original key. Recently, [13] experimentally proved that instructions can be faulted at several points along the instruction channel, from the Flash interface to the execution pipeline, and also up to the fetch and execution stages in the pipeline itself using LFI.

What is called a Fault Model (FM), is the description of the faults main characteristics. At bit level, common FMs are: bit-set, bit-reset, and bit-flip [14]. While at instruction level, the FMs include modification of instructions, skip of instructions, and replay of instructions. Among which, skip of instruction(s) FMs draw great attention from attackers because it is as if some instructions were erased on-the-fly from the program, making it interesting for exploitation purposes. Based on the number of faulted instructions, the skip FM can be classified into single instruction skip and multiple instructions skip. Obviously, multiple instructions skip is more threatening as compared to the single instruc-

tion skip. Dutertre et al. [10] reported on a powerful multiple instructions skip obtained with LFI. Recently, [13] and [15] reported the ability to respectively achieve multiple instruction skips with both LFI and EMFI.

In response to the increasing threat from the attack side, a great deal of effort has been put into the development of CMs for device protection. CMs can be classified as either hardware or software. Hardware CMs can be very effective when implemented on a specific device against a specific type of fault based on its physical properties. However, it may be ineffective for another type of fault and it is impossible to change (or update). For example, in [16], the author proposed a hardware CM at Integrated Circuit (IC)-level that monitors the unusual bulk currents induced by LFI. However, these techniques may not be able to offer protection against EMFI. While software CMs are more attractive because they can be updated constantly; and more interestingly, they bring no redesign cost to the existing devices[17, 18, 19].

Many of the existing software CMs are developed considering FMs at instruction level, which includes: instruction skip, and instruction modification. In [18], the authors proposed CM based on redundancy. In [17], the authors proposed a CM against the single instruction skip FM ; they also formally verified its effectiveness. However, these methods mainly consider single instruction skip fault. In this work, we experimentally demonstrated that it can be extended to the multiple instructions skip FM, and proposed two CMs for detection of multiple instruction skip. We implemented both CMs in a 32-bit MCU and experimentally tested their effectiveness.

This paper is organized as follows. Section 2 provides the information about the target, experimental setups, test procedure, and FMs definitions. Section 3 provides a survey on the state-of-the-art of the existing CMs against FI. Section 4 introduces the principle of the two proposed software CMs. Section 5 reports on the implementation of the two proposed CMs in the target and theirs effectiveness when exposed to LFI and EMFI attacks. In section 6, we discuss the limitations of the two CMs. Finally, section 7 provides the main conclusions and perspectives.

2. Experimental setup and methodology

2.1. Device under test

Our test target is a 32-bit MCU: a SAMD21G18A [20] which uses an ARM Cortex-M0+ core (2-stage pipeline). This core implements an ARMv6 thumb architecture and the Thumb-2 ISA of which most of the instructions have a 16-bit length [21]. The MCU is equipped with a 256 Kb Flash, and a 32Kb SRAM; the data transfer between the memories and the processor is performed via 32-bit AHB bus. A cache memory of 8× 64-bit lines is included to improve the performances of the MCU. For all the reported experiments, the MCU was configured to work at 12 MHz with zero wait states which guarantees that there is no delay during the data read operations from the Flash memory.

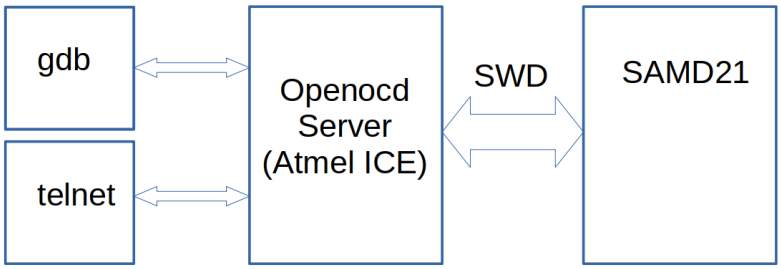

Figure 1: Schematic of debugging target using Openocd

The MCU was debugged using an Atmel-ICE Debugger which allows stopping the program at a breakpoint and collecting register data for further analysis. Fig. 1 shows the block diagram of the debugging system we used using Atmel-ICE. In this system, an Openocd server is run with Atmel ICE which is connected to the SAMD21 via a SWD interface. The servers accepts connections from client such as GDB and telnet. In our test, a telnet client was used to send commands to and receive data from the Openocd server.

2.2. Experimental setup

During the reported experiments we carried out EMFI and LFI on the device under test.

2.2.1. EMFI bench

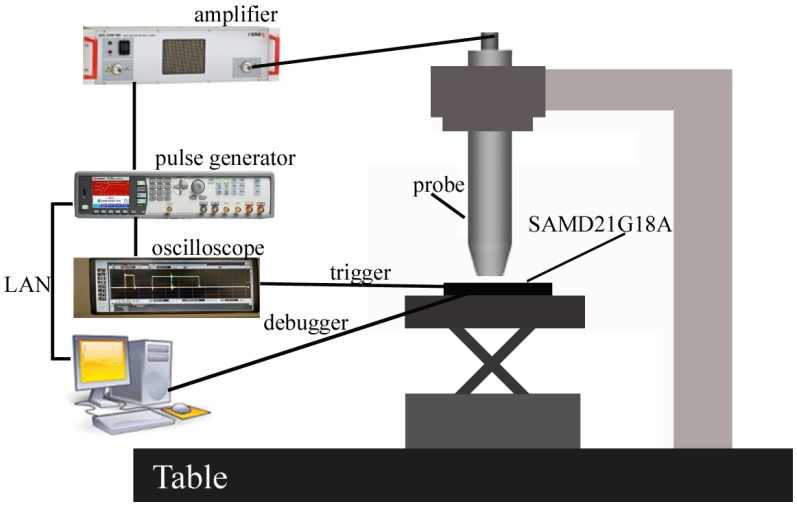

Figure 2: Schematic of the EMFI experimental setup

Fig. 2 depicts the experimental setup used for conducting EMFI on the MCU target. It consists of an oscilloscope, an arbitrary waveform generator (AWG), a power amplifier, an injection probe, a computer, and the device under test. The AWG is the Keysight Pulse generator 81160A, which is capable of generating a voltage pulse as short as 1.5 ns with a rising time of 1.0 ns. Its output is fed to a CBA 400M-260 Power Amplifier, which delivers an amplified voltage pulse to a handmade injection probe. The latter is built with four loop turns of a 150 μm insulated copper wire around a ferrite core designed as a circular truncated cone, with a top diameter of 1.5 mm and a bottom diameter of 0.8 mm. As a result, a strong EM perturbation is created in the vicinity of the injection probe, which may in turn induce faults into the device under test at run time (it is placed at a distance of a few hundreds of μm from it). Synchronization between the operations of the target (the execution time of the test code being under EMFI) and the EM perturbation onset is made thanks to trigger signal generated from the device under test prior to the start of the test code. The trigger signal is used to trigger the AWG pulse with a tunable delay. The oscilloscope is used

to observe and set the synchronization.

2.2.2. Laser bench

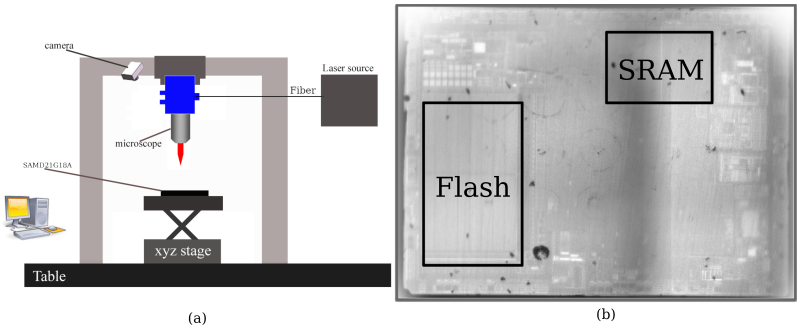

Figure 3: LFI experimental setup: (a) laser bench schematic, (b) target backside image taken using an IR camera

Fig. 3 shows the experimental setup we used for conducting LFI on the MCU. The laser platform, as shown in Fig. 3(a), consists of a laser source, a microscope, an `XYZ` stage, an IR camera, and a computer. The laser source can produce laser pulses with a wavelength of 1,064 nm which allows the light to pass through several hundreds of μm of silicon. The laser Pulse Width (PW) is tunable in the range from 50 ns to 1 s. In addition, the laser source allows obtaining a programmable delay, and a power ranging from 0 to 3 `W`. The light is conducted to and focused by a microscope. In our experiments, we used a $5\times$ objective to focus the laser beam on the transistors of the device under test. The diameter of the laser spot was 20 μm. The device under test was mounted on a `XYZ` stage which allows controlling the position of the laser spot with an accuracy of 0.1 μm. The Infrared (IR) camera was used to observe the active part of the device and the location of the laser spot through the bulk (Fault Injection Attacks (FIA) were carried out through the target backside). The computer was used to control the laser pulse parameters as well as communicate with the device under test.

The MCU was unpackaged from the backside to ensure that the light is able to reach the transistor layer. Notice that the laser power is strong enough to

reach the circuit layer of the MCU without the requirements of thinning it down. Fig. 3(b) shows the image taken with the IR camera from the backside of the MCU. The positions of the Flash and SRAM memories are marked with black rectangular shapes. Other structures in the circuit layer can also be seen.

2.3. Test procedure

A scan of the chip surface was performed to find the positions sensitive to FI. The probe , or laser spot, position was then fixed at an optimal position that provides the highest fault rate. All the registers were initialized to a known value at the beginning of all the tests to ensure fault traceability. One test iteration follows three main steps: (1) the target is reset and all systems registers are initialized; (2) the trigger for the pulse generator (either EMFI pulse or LFI pulse) is set, and the test code is executed; (3) all the register values are collected as the program reaches a configured breakpoint, or when an interrupt routine is performed.

During the experiments, **100 FI tests using either laser or Electromagnetic (EM) with a specific configuration of power, delay time and pulse width were performed. And the attack position remained unchanged for all the experiments.** Before each test, we ran the considered test code without FI. we collected the value of all the registers to confirm the that the program functions correctly in the normal condition and used it as a reference to detect EMFI or LFI induced faults.

2.4. Instruction(s) Skip FM definitions

At software level, common FMs are: instruction replay, instruction modification and instruction(s) skip. Among which the instruction(s) skip FM poses a significant threat. In this FM, one or more instruction(s) are replaced by no-operation(s) (`nop`) instructions (or by another random instruction that has no effect on the executed code and thus can be considered as skipped [10]).

Based on the number of faulted instructions, the faults are classified into single and multiple instruction(s) skip. In single instruction skip FM, FI impacts

only one instruction. Effective CM has been proposed against it [17]. In multiple instructions skip FM, two or more consecutive instructions are impacted by FI. This FM is more complex to obtain, and so as the CM to thwart it.

Skips of multiple instructions were achieved with both EMFI and LFI. In [11], the authors were able to achieve up to six instructions with EMFI. While in [10], Dutertre et al. were able to obtain a skip of up to 300 instructions using a relatively long laser pulse width. Recently, [15] reported the skip of a block of up to four instructions using EMFI (faults were induced into the memory Flash interface buffer). Using LFI, the same authors were able to fault instructions when traveling from the Flash interface to the execution pipeline stages.

In this work, we focused on EMFI and LFI induced instructions(s) skips. The experiments we conducted were tuned in order to inject various forms of skips as shown in Table 1. It provides four test codes to exemplify the definition of the instruction skip FMs we used throughout this work.

Table 1: Skip FM fault definitions

Reference code	**skip** $i_5i_6i_7i_8$	**skip** i_5i_6	**skip** i_5
i_1. add r0, r0, #1	i_1. add r0, r0, #1	i_1. add r0, r0, #1	i_1. add r0, r0, #1
i_2. add r0, r0, #2	i_2. add r0, r0, #2	i_2. add r0, r0, #2	i_2. add r0, r0, #2
i_3. add r0, r0, #4	i_3. add r0, r0, #4	i_3. add r0, r0, #4	i_3. add r0, r0, #4
i_4. add r0, r0, #8	i_4. add r0, r0, #8	i_4. add r0, r0, #8	i_4. add r0, r0, #8
i_5. add r1, r1, #1	i_5. nop	i_5. nop	i_5. nop
i_6. add r2, r2, #1	i_6. nop	i_6. nop	i_6. add r2, r2, #1
i_7. add r3, r3, #1	i_7. nop	i_7. add r3, r3, #1	i_7. add r3, r3, #1
i_8. add r4, r4, #1	i_8. nop	i_8. add r4, r4, #1	i_8. add r4, r4, #1

The reference code consists in eight instructions (additions of immediate values in registers `r0` to `r4`). For convenience, these eight instructions are denoted as (i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8). Notice that concerning instruction(s) skip FM, we assume that a skipped instruction is replaced by an instruction equivalent to a `nop`.

- skip $i_5i_6i_7i_8$: four instructions (i_5, i_6, i_7, i_8) are turned into instructions

equivalent to (*nop*, *nop*, *nop*, *nop*).

- skip $i_5 i_6$: two instructions (i_5, i_6) are turned into (*nop*, *nop*).
- skip i_5: a single instruction (i_5) is turned into (*nop*).

It should be pointed out that to differentiate between the replay and skip FMs, `nop` instructions should not be used in test codes, because the replay and skip of a block of `nop` instructions are equivalent [15].

3. State-of-the-art - CMs against the single skip FM

In this section, we discuss the existing CMs that have been developed against FI and their main features.

3.1. ***Randomization***

The instruction skip FM assesses the ability for an attacker to skip a chosen instruction (or group of instructions) with a 100 % success rate provided an accurate synchronization of the injection process w.r.t. the target's activity is achieved. This is a strong requirement that suggests to use desynchronization as a CM. Indeed, introducing a certain level of randomization into a program execution [22] will deny an attacker the ability to target specifically any of its instructions. Randomization can be built from random numbers of dummy instructions inserted in the course of a program, or by jumping randomly between different of its subparts (when compatible with the considered algorithm). For example, inserting randomness in the PIN verification of [11] would have prevented it from succeeding.

However, randomization is not an absolute defense:

- it only delays an attack success if the attacker is allowed to carry out several attack attempts in a row (thus degrading the attack success rate),
- some attack schemes do not require an accurate timing of FI to succeed (e.g. the attack of a RSA algorithm reported in [23]).

3.2. ***Redundancy***

Redundancy is a long-established defense against faults [24, 22, 25, 26, 27, 28]. It consists in executing several times the same subprogram and to compare the outputs. It is based on the assumption that an attacker is not able to induce an identical fault into each execution. There exists several variants of redundancy, the more common are:

- code duplication with comparison that makes it possible to detect FI,
- code triplication with majority vote that provides the ability to correct faults injected in one execution,
- duplication with inverse computations (e.g. for an encryption algorithm: perform the encryption and then decrypt the obtained cipher, any difference from the initial input indicates that a fault was injected) that further reduces the ability of an attacker to induce several times the same fault because the target's computations are different.

However, redundancy-based CMs suffer from a high overhead in execution time which is at least doubled or tripled (and possibly in code size as for duplication with inverse computations). The previous state-of-the-art in FI reported an ability to skip two instructions sufficiently away in time (several ms for laser injection [23], and possibly less for EM injection given the $50\,\mu s$ repetition rate of the used voltage pulse generators).

As a result, duplication techniques applied at instruction level were introduced for the purpose of choosing carefully the protected instructions. In addition, it may save code overhead (all instructions are not equally sensitive), allow quasi-immediate detection or mitigation of the injected fault. These defenses were built on the assumption that a fault (an instruction skip in our case) is restricted to a single instruction.

Barenghi et al. introduced in [18] a framework based CMs solution applied at the instruction level of a program. Three techniques within the framework are considered: instruction duplication, instruction triplication, and a parity check

Table 2: Redundancy-based software defense against FI - FI detection [18]

Initial code	**Redundancy-based defense**
`ldr r1, [r0]`	`ldr r1, [r0]`
	`ldr r2, [r0]`
	`cmp r1, r2`
	`bne <error>`

method. The main principle of this defense is based on duplication and comparison. It is illustrated in Table 2 for a load instruction of a value stored in RAM into a register: `ldr r1, [r0]` . A second register `r2` is used in the duplicated instruction. Then two additional instructions are used to compare the values stored in the two registers and branch into an error handler if the content of the two registers differs (hence revealing a store instruction was faulted). This defense is designed to provide data and code integrity. The authors detailed these CMs with tests on a AES implementation and suggested that the framework is employable on the whole ARM processor family. **From the overhead point of view, execution time and code size are quite high. For example, as shown in Table 2, to protect a single `load` instruction three additional instructions are needed.**

Moro et al. proposed in [17] a variation on this CM based on duplication without detection to cope with the single instruction skip FM. It is exemplified in the top part of Table 3 for an instruction adding 1 to the content of a register and storing the result in a second register. A simple duplication of this instruction is sufficient to assure a mitigation against a single instruction skip (i.e. assuring that the correct result is stored in the destination register even if one of the two instruction is skipped). This defense was suitable for this specific instruction because its duplication did not change the result, a property called idempotence.

Duplication of the addition instruction in the lower part of Table 3 was not feasible without changing the result because the same register `r1` is used both as a source and a destination register. However, using an additional register `r3`

Table 3: Duplication-based software defense against the single instruction skip FM [17]

Initial code	**Idempotent instructions**	**Duplication-based defense**
`add r1, r0, #1`		`add r1, r0, #1`
		`add r1, r0, #1`
`add r1,r1,r2`	`add r3, r1, r2`	`add r3, r1, r2`
	`mov r1, r3`	`add r3, r1, r2`
		`mov r1, r3`
		`mov r1, r3`

with a move instruction turns it into an idempotent series of two instruction that can be safely duplicated (see resp. the mid and last columns of Table 3). The authors formally verified the efficiency of this countermeasure on a 32-bit ARM Cortex-M3 MCU, based on an ARMv7-M architecture that runs the Thumb-2 instruction set; the application tests were applied to the AES and SHA-0 algorithms. The overhead, when applying this countermeasure, is important as reported by the authors: it may be higher than 100 % in clock cycles number and 200 % in code size. Hence, they suggest that it shall be applied in selected sensitive parts only of the target program to reduce the overhead.

This countermeasure was applied by Barry et al. in [29] on a modified Low Level Virtual Machine (LLVM) compilation tool as a generic mechanism to protect a code. The authors introduced a new approach that generates for all instructions an equivalent idempotent instructions and then proceeds with the related duplication process. They also proposed an instruction scheduling mechanism that proceeds to rearrange the execution order of the modified instructions to ensure a better execution time and an increased resistance to instruction skips. With this approach, it was possible to reduce the execution speed overhead and code cost by approximatively the half from the results obtained by [17]. **However, the code with this CM may become more vulnerable to either FIA or side-channel attacks as discussed in[30]**.

3.3. ***Hardware-assisted software countermeasures***

Danger et al. provide in [31] a solution based on extra hardware implemented block named Code and Control-Flow Integrity (CCFI). The method is presented as a generic countermeasure and is not intrusive since it does not need any modification of the core. This extra block is composed of two parts: a first part is used to store metadata related to the code and control flow information, and the second module is needed for the integrity check of both the code and the control flow. This method was tested on an implementation of PicoRV32 based on RISC-V ISA (three-stage pipeline). Their evaluation showed that it provided protection against simulated fault attacks. While, in [32], Yuce et al. demonstrate a new and flexible strategy against FIA. The presented countermeasure is based on a hardware detector combined with a software block that handle the fault flag and run the application specific fault response. A low-level hardware checkpointing mechanism is used to recover from FI; and the application-specific response is enabled by a software secure trap. The evaluation was proceeded by comparison of the protected code to both its unprotected version and its protected version using the instruction duplication method.

These CMs, though interesting, require to modify the hardware of the protected device. Our choice in this work was to devise software CMs that can be used for already existing devices (with no need of hardware modification/upgrade) with a simple update of their program.

4. Proposed CMs against multiple instructions skip FM

4.1. CM based on code duplication and a dedicated counter

The CM proposed and formally verified in [17] is effective only for a single instruction skip (or distinct skips that cannot target simultaneously the duplicated instructions). The implementation of this method includes two steps: (1) transformation of instructions into idempotent form (as exemplified in Table 3), (2) duplication of the instructions obtained in step 1. However, the CM is not effective for multiple instructions skip, i.e skip of more than one instruction.

Table 4: Steps to implement the detection method based on code duplication and a hardware dedicated counter

Reference code	**Step 1**	**Step 2**	**Step 3**
i_1	$inst_1$	$inst_1$	$inst_1$
i_2	$inst_2$	$inst_1$	$cnt = cnt + 1$
i_3	$inst_3$	$inst_2$	$inst_1$
i_4	$inst_4$	$inst_2$	$inst_2$
		$inst_3$	$cnt = cnt + 1$
		$inst_3$	$inst_2$
		$inst_4$	$inst_3$
....	....	....	....

By introducing a hardware counter and inserting an instruction incrementing this counter in between the duplicated instructions, we obtained a CM that is effective for both single and multiple instructions skips. Table 4 depicts the implementation of the CM. Here, we consider a piece of code to be protected consisting of four instructions (i_1, ...,i_4) as shown in the first column of Table 4. **Step 1** is to transform all the instructions into idempotent instructions ($inst_1$, ...,$inst_4$) (noted that the transformation may not be a one-to-one mapping). In **Step 2**, the idempotent instructions are duplicated. After **Step 2**, the protected code is the same as the one proposed by Moro et al. [17], which was proved to be effective against single instruction skip FM. In **Step 3**, an instruction to increase the dedicated counter is added in between the duplicated instructions. The value of the counter is initialized at the beginning of the program and its final value is also collected at the end and compared with the expected value to detect any injected fault. Notice that the code to check the counter value can be implemented at several points in the program to timely detect the fault as well as to make sure that this part is not skipped by the attack.

Table 5 shows how the CM works against single and multiple instructions skip FM. In **Case 1**, if a single instruction skip targets an instance of the duplicated $inst_1$, there will be no effect on the result, the code is protected

Table 5: Working principle of the method based on code duplication and a hardware dedicated counter against skip(s) FM

Case 1	**Case 2**	**Case 3**	**Case 4**	**Case 5**
$inst_1$	$inst_1$	$inst_1$	$inst_1$	~~$inst_1$~~
$cnt = cnt + 1$	~~$cnt = cnt + 1$~~	~~$cnt = cnt + 1$~~	$cnt = cnt + 1$	~~$cnt = cnt + 1$~~
~~$inst_1$~~	$inst_1$	~~$inst_1$~~	~~$inst_1$~~	~~$inst_1$~~
$inst_2$	$inst_2$	$inst_2$	~~$inst_2$~~	$inst_2$
$cnt = cnt + 1$	$cnt = cnt + 1$	$cnt = cnt + 1$	$cnt = cnt + 1$	$cnt = cnt + 1$
$inst_2$	$inst_2$	$inst_2$	$inst_2$	$inst_2$
$inst_3$	$inst_3$	$inst_3$	$inst_3$	$inst_3$
....	....	....	....	....

without detection. **Case 2** reports how a counter increment instruction may be skipped. As a result, the final value of the counter is corrupted and finally detected when compared to the expected values (note that the program still produces an error-free result). In **Case 3**, if two instructions `cnt=cnt+1` and $inst_1$ are skipped, the final result of the `cnt` is changed and does no longer match with the expected value, therefore the fault can also be detected. In **Case 4**, if instructions `inst1` and `inst2` are skipped, the fault is corrected without detection. Finally, in **Case 5**, if the fault models are three instructions skip (or more) we can be sure that the value of the `cnt` is faulted and not matched with any expected value. Therefore it can be detected. Notice that the value of `cnt` can be checked at several points in the program to increase the effectiveness of the CM. In addition, an error may be introduced as illustrated by skipping both duplicated $inst_1$ instructions.

From the above analysis, we see that the proposed method is effective with both single and multiple instruction(s) skip. **Specifically, for the single instruction skip, the CM can either automatically correct the result or detect fault via checking counter's value and all the instructions including branches, calls can be protected. The dedicated counter provides a means for detecting the multiple instructions skip fault.**

The working mechanism of the dedicated counter here is quite the same with the signature monitoring for detecting control flow error in [26]. Therefore, it can also be use for detection of control flow error fault. However it is out of the scope of this paper. It should be also noticed that in here the counter is inserted in between the duplicated instruction while in [26] the counter is inserted after every instruction.

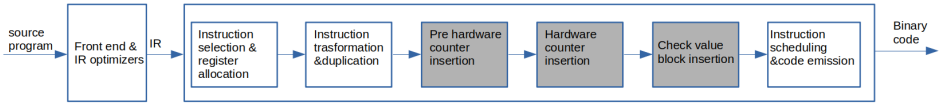

Figure 4: Proposed schematic for implement the CM using LLVM

The generalization and automation of this CM can be completed following the method described in [29] in which the author used modified LLVM compiler to transform instructions into idempotent form and duplicate it afterward. The procedure to implement this CM is shown in Fig. 4. Notice that the blocks marked with white color have already been implemented and verified in [29, 17]. We propose to add three blocks marked with grey color to implement this CM: (1) pre hardware counter insertion to check and reallocate the use of the registers, (2) hardware counter insertion to insert an instruction to increase the counter between the duplicated instructions, (3) check value block insertion to insert a block for checking the counter value. **As an example, we applied the CM to the verifyPIN program automatically using LLVM and ran it in an ARM MCU emulated with QEMU to evaluate the performance. The result shows that the code size overhead is 293% and the execution time overhead is 201%.**

4.2. Counter-measure using a SI as a sensor

In MCU, the program instructions are stored in the non-volatile memory such as an embedded Flash. Normally, several instructions are loaded into the Flash interface buffer before being transferred to the core for execution.

Unfortunately, the process related to loading instructions into the Flash interface buffer is quite sensitive to FI. And the block of instructions being loaded each time can be targeted as a whole resulting into a multiple instructions skip. In [6], the author reported the replay of a block of instructions caused by EMFI. In [33], the authors provided an in-depth analysis of EMFI-induced instruction buffer fault. Recently, the work in [34, 13, 15] reported on experimental basis the ability to fault the instruction buffers and to obtain multiple instructions skip with both EMFI and LFI. A closer inspection at bit level revealed that bits corruptions inside the buffer were induced. As a result, the opcodes of the instructions loaded into the buffer were changed into non-existent opcodes and hence not recognized by the core. In effect, they were considered as *nop*s resulting into a multiple instructions skip (which number was that of the buffer size expressed in number of instructions). The CM we propose takes into account that FM at block level by inserting so-called SIs. The idea here is to insert a SI into the block, and check its result to detect if the block of instructions was skipped. Notice that the SI is carefully chosen based on the FM at bit level. If the FM is bit-set, the SI should be an instruction having an opcode containing as many bits at `0` as possible. In contrast if the FM is bit-reset, the SI should be an instruction having an opcode containing as many bits at `1` as possible.

Table 6: Implementations of detection method based on SI as a sensor

Reference code	**Implementation 1**	**Implementation 2**
i_1	#`block1`	#`block1`
i_2	SI_1	SI_1
i_3	i_1	i_1
i_4	i_2	i_2
	i_3	SI_2
	#`block2`	#`block2`
	SI_2	SI_3
....	i_4	i_3
....	....	....

Table 6 illustrates the implementation of this method. Here we consider the case in which the size of the buffer corresponds to four instructions. The code needs to be protected is shown in the **Reference code** column. The second column **Implementation 1** shows the code with each block containing one SI, and the third column **Implementation 2** shows the code with each block containing two SIs.

The result produced by instructions SI_N (with N being 1, 2, 3..) is used to detect if there is any fault caused by the FI. And notice that it can be checked at several points in the program to timely detect the fault.

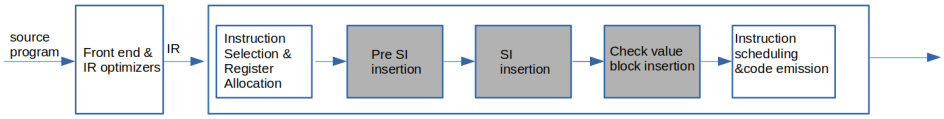

Figure 5: Proposed schematic for implement the CM using LLVM

The generalization and automation of this CMs can also be accomplished using the compiler such as LLVM. We proposed the steps to implement this CM using LLVM in Fig. 5. There are three additional boxes marked with grey color includes (1) pre SI insertion to check and reallocate the use of registers, (2) SI insertion to insert the SI into each block of instructions, (3) check value box insertion to check the value of the register used in SI. **We also used LLVM to generate the CM for verifyPIN program and ran it in an ARM MCU emulated with W to evaluate the performance. The result shows that the code size overhead is 200%, and the execution time overhead is 149.5%.**

5. Validation of the proposed CMs against LFI and EMFI on ARM cortex M0+

To evaluate the effectiveness of the CMs proposed in Section 4, we used LFI and EMFI with parameters that makes it possible to skip two and four successive instructions.

5.1. CM based on code duplication and dedicated counter

Table 7 shows the test codes we wrote for the evaluation of this CM. For

Table 7: Test codes used for evaluation of the CM based on code duplication and a dedicated counter on ARM Cortex-M0+

Code to be protected	**Code protected with CM proposed [17]**	**Protected code as proposed in Section 4.1**
i_1. `add r1, r0, #1`	i_1. `add r1, r0, #1`	i_1. `add r1, r0, #1`
i_2. `add r2, r0, #1`	i_2. `add r1, r0, #1`	i_2. `add r5, r5, #1`
i_3. `add r3, r0, #1`	i_3. `add r2, r0, #1`	i_3. `add r1, r0, #1`
i_4. `add r4, r0, #1`	i_4. `add r2, r0, #1`	i_4. `add r2, r0, #1`
	i_5. `add r3, r0, #1`	i_5. `add r5, r5, #1`
	i_6. `add r3, r0, #1`	i_6. `add r2, r0, #1`
	i_7. `add r4, r0, #1`	i_7. `add r3, r3, #1`
	i_8. `add r4, r0, #1`	i_8. `add r5, r5, #1`
		i_9. `add r3, r3, #1`
.....	.....	.....

simplicity, the four instructions were all chosen to be idempotent. The instructions are simple `add rx, r0, #1` adding operations, which adds `1` to `r0` register and stores the result into `rx` register, with `x` being `1`, `2`, `3`, `4`. The second column shows the test code protected with the CM proposed in [17]. The third column shows the test code protected with the detection method described in Section 4.1, in which register `r5` is used as the hardware counter. The counter `r5` is initialized at `0` at the beginning of the program (not shown in the CM code), and increased by `1` after each instruction.

These three codes were experimentally tested in terms of their ability to mitigate (or not) the skip of two instructions. Fig. 6 reports the obtained results (from left to right: unprotected code, code from [17], code implementing the duplication with the hardware counter increment CM). The LFI and EMFI were used to produced skip of first two instructions in each test code. LFI results

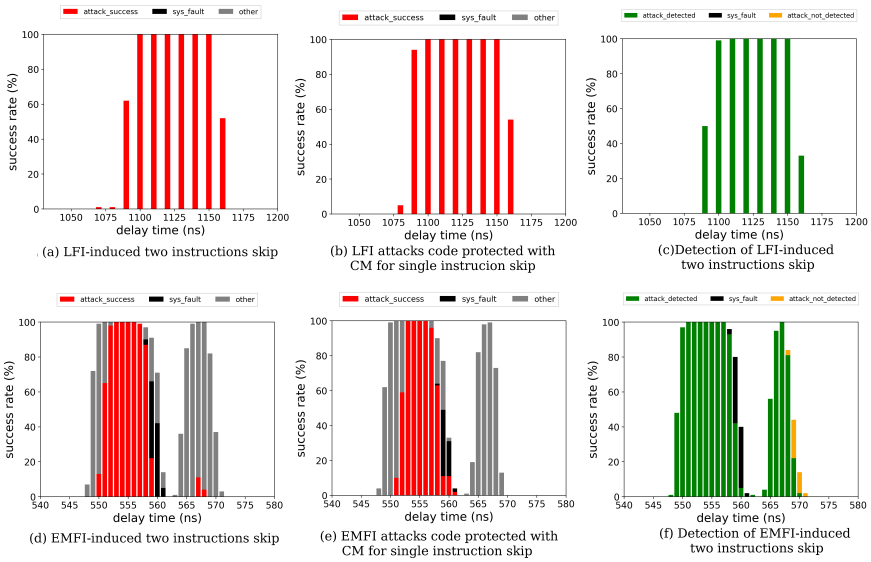

Figure 6: Test result with LFI-induced two instructions skip: (a) Skip of two instructions obtained with LFI-induced fault on data loaded into AHB bus, (b) Bypassing available CM with success rate of 100%, (c) Detecting the fault with dedicated counter, Test result with EMFI-induced two instructions skip: (a) Skip of two instructions obtained with LFI-induced fault on data load into Flash interface buffer, (b) Bypassing available CM with success rate of 100%, (c) Detecting the fault with a dedicated hardware counter.

are given in the figure top part, and EMFI results in its bottom part. We used the following denominations to identify the various results shown in Fig. 6:

- `attack_success`: the obtained result is faulted (incorrect) and the attack is undetected.
- `sys_fault`: the MCU stops working due to the attack.
- `other`: the faults are not the skip of two instructions.
- `attack_detected`: the attack is detected by the CM.
- `attack_not_detected`: the attack is not detected by the CM.

The result tested with LFI are presented in Fig. 6 (a), (b), and (c). The laser power was `1.5 W`, and the PW was `50 ns`. Notice that by using the LFI

to fault the data loaded into the AHB bus [13], we were able to achieve FM of skip of two instructions with a success rate of 100 % as shown Fig. 6 (a). It can also be used to bypass the CM proposed in [17] with a success rate of 100% as shown in Fig. 6(b). For the CM, it is quite obvious from Fig. 6 (c) that all the attacks were detected.

Notice that the skip of two instructions with a fault rate of 100% was obtained with EMFI by faulting the data loaded into Flash interface buffer [15]. The result tested with EMFI are presented in Fig. 6 (d), (e), and (f). The Pulse Amplitude (PA) was `0 dB`, while the PW was `7.0 ns`. As shown in Fig. 6 (d), skip of two instructions with a fault rate of 100% was obtained. As compared to the result obtained with LFI, the EMFI-induced fault are more complex, as shown in Fig. 6 (d), apart from the skip of two instructions, there are also the other fault and system fault. Similarly, as applied to the code with CM proposed in [17], for the the time interval which the two instruction skip occurs, the attack success rate can reach 100% as shown in Fig. 6 (e). For our CM, the detection rate at the corresponding time interval can also be 100%. However there are some faults that are not detected by our detection method (i.e at the delay time from 568 to 571 `ns` in Fig. 6. This is because these faults are related to the modification of register `r0`, resulting into the fault of the value of the registers `r1` to `r4`, while the value of the register `r5` is still correct. In short, we can see that the proposed CM is effective for instruction(s) skip FM caused by the LFI and EMFI.

5.2. Counter-measure using a SI as a sensor

The idea behind the CM using SI is to insert into every instructions block one or several instructions that are particularly sensitive to both LFI and EMFI. As a result, any FI attempt would likely corrupt these SIs providing evidence that an attack in under way. A preliminary work was to identify the relevant FMs. To do so, we wrote three test codes (given in Table 8). The first one consists in four instructions `add rx, rx, #1`. The second one consists in four `lsl r0,r0,#0` instructions (i.e. instructions which opcode is `0x0000`) in order

Table 8: Test codes for characterization of FMs at instruction level and bit level on ARM Cortex-M0+

Test code for instruction level FM detection	**Test code for bit-set fault detection**	**Test code for bit-reset fault detection**
i_1. `add r1, r1, #1`	i_1. `lsl r0, r0, #0`	i_1. `sub r7, r7, #0xff`
i_2. `add r2, r2, #1`	i_2. `lsl r0, r0, #0`	i_2. `sub r7, r7, #0xff`
i_3. `add r3, r3, #1`	i_3. `lsl r0, r0, #0`	i_3. `sub r7, r7, #0xff`
i_4. `add r4, r4, #1`	i_4. `lsl r0, r0, #0`	i_4. `sub r7, r7, #0xff`

to be sensitive to bit-set faults. The third one consists in four `sub r7,r7,#0xff` instructions (instructions which opcode is `0x3fff` i.e. instructions with as many bit at `1` as possible) in order to be sensitive to bit-reset faults.

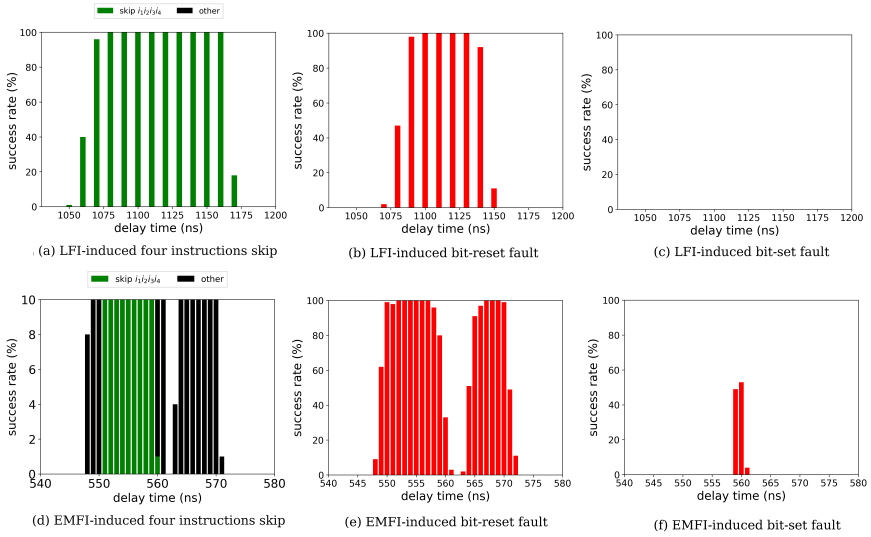

Figure 7: characterization of LFI and EMFI -induced fault at bit level and instruction level, LFI-induced fault: (a) skip of four instructions, (b) bit-reset fault rate, (c), bit-set fault, EMFI-induced fault:(d) skip of four instructions, (e) bit-reset fault rate, (f), bit-set fault rate

460 The laser power was set to 1.5 `W`, and the PW to 50 ns. The MCU was configured to work in cache enable mode, meaning that the Flash interface buffer size is 64-bits i.e. it contains a block of four instructions. Fig. 7 reports

the experimental results of LFI and EMFI on the **three** previous test codes. Fig. 7(a) provides the results at instruction level: the success rate in skipping the whole block of four instructions. The sensitivity window has a range of $\sim$ 125 ns (for a clock period of $\sim$ 83 ns) and it repeats itself with a period of four clock cycles. Fig. 7(b) shows the fault rate obtained when most of their bits in the buffer are `1`, many faults were observed (with a success rate of 100% for some timings). Fig. 7 (c) shows the fault rate obtained when the buffers were filled with bits at `0` , almost no faults were induced.

We also conducted the same test with EMFI to characterize its FM at instruction level and bit level. The results are shown in Fig 7(d), (e), and (f). We can see that at instruction level skip of four instructions can be achieved with EM pulse as shown in Fig 7(d). Concerning the FM at bit level, we can see that the fault mostly occurs when the buffers are filled with bits at one, while very few faults occur when the buffers are filled with zero bits. Through this test, we assume that: (1) at bit level, the faults induced by LFI and EMFI are bit-reset rather than bit-set, (2) at instruction level the fault is skip of instruction(s). In other words, instructions skip is caused by EMFI- or LFI-induced multiple bits reset on instructions' opcode loaded into buffer, and are fully reproducible as shown in Fig. 7 (b) and (e).

Based on the analysis above, we applied the CM proposed above to our target. For simplicity, we used four instructions corresponding to one block loaded into the Flash interface buffer as the code needed to be protected as shown in Table 9 (first column). After inserting the SI instructions, we obtained the protected code as shown in Table 9 (second column). Notice that now it becomes two blocks of four instructions. The `sub r7, r7, #0xff` is used as the SI. **At the beginning of `r7` is initialized, and the end, its value is check and compared with the expected value to detect whether there is a fault or not**. We used the LFI to target theses instructions. The fault are also classified into: (1) `attack_detected` , (2) `attack_not_detected`, (3) `sys_fault` . The result obtained with LFI are presented in Fig. 8 (a), (b) and (c). The laser PW was 50 `ns` and the laser powers were 1.5 `W`, 1.1 `W`, 0.9 `W`. In

Table 9: Implementation of the CM using a SI as a sensor on ARM Cortex-M0+

Code to be protected	**Protected code with CM proposed in 4.2**
	#initialization of r7
i_1. add r1, r1, #0x01	#block1
i_2. add r2, r2, #0x01	i_1. sub r7, r7, #0xff
i_3. add r3, r3, #0x01	i_2. add r1, r1, #0x01
i_4. add r4, r4, #0x01	i_3. add r2, r2, #0x01
....	i_1. sub r7, r7, #0xff
....	#block2
....	i_1. sub r7, r7, #0xff
....	i_2. add r3, r3, #0x01
	i_3. add r4, r4, #0x01
	i_1. sub r7, r7, #0xff
	#check value r7

most of the case, the attacks are detected by monitoring the value of register r7. No attack_not_detected was observed when tested with the LFI. The result strongly recommends that the CM is effective for detecting the multiple instructions skip.

The result obtained with EMFI are presented in Fig. 8 (d), (e), and (f). The PW was 7 ns and the PAs are 0 dB, -4 dB, -8 dB. It is also obvious that most of the attacks can be detected using our detection method. There is a very few attack_not_detected fault. In short, the result shows that the CM is effective for detecting either LFI and the EMFI.

Also notice that our method here differs from the one proposed in [26] in the way the SI is selected and inserted into to the instructions block. For the SI, it can be selected based on the FM at bit level. For example, in our case: (1) if the FM is bit-set the `lsl r0, r0, #0x00` **is used, (2) if the fault at bit level is bit-reset then the** `sub r7, r7, #0xff` **is used, (3) if the fault at bit level is bit-flip, both instructions can be**

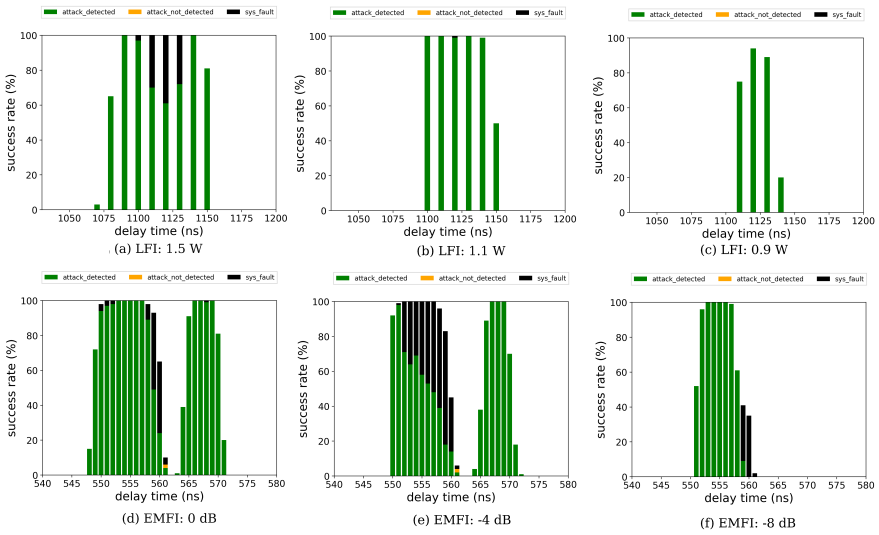

Figure 8: detection rate of CM based on SI, LFI: (a), 1.5 W, (b), 1.1 W (c) 0.9 W, EMFI: (d) 0 dB, (e) -4 dB, (f) -8 dB

used for detection. The block of instructions is formed based on the number of instructions loaded into buffer at one time which is highly related to the architecture of the device.

Concerning the code size overhead of this CM, we can see that a block of four instructions are formed by two SIs and two original instructions. In addition, blocks to check the value of the register related to SI are needed at some specific points in the program. Therefore, the code size overhead is approximately 100%. And the execution time overhead should be less than 100% because the execution time of the SI is only one clock cycle.

6. Limitations of the proposed CMs

The two CMs proposed in this paper show the their effectiveness for detection of multiple instructions skip caused by fault injection including LFI and EMFI. However, there are several limitations of the two CMs.

For the CM based on duplication and dedicated counter, the limitations includes the followings. (1) the implementation of this method is quite complex.

First, we need to implement the method proposed in [17] and then insert the counter in between the duplicated instructions, finally the code to check the counter value is also needed to implemented. (2) the code size over head of the protected code is quite heavy (approximately 300%). Indeed, we see that after the instruction transformation and duplication the code size is increased significantly. Here the insertion of the dedicated counter and the code to check its value also adds more code size overhead to the program.

For the CM based on SI instruction as the sensor, the limitation includes the followings. In this method, the SIs need to be found and this is a complex process due to the fact that instruction sets vary depending on the target's architecture. And this CM only works with the fault related to a block of instructions. Notice that the SIs and size of the block of instructions largely depend on the architecture of the devices and the type of attacks. **And this CM can not be used for detection of the attack in which the attackers are able to target specific instruction(s) in side one protected block.**

Additionally, the efficacy of the two CMs against other faults such as control flow error has not been tested.

7. Conclusions & perspectives

In conclusion, in this paper, we proposed two software CMs against multiple instructions skip FM. The first CM is based on instruction duplication and dedicated counter. In this CM, the instructions are first transformed into idempotent ones which is further duplicated. Finally, the dedicated counter is added in between the instructions. We experimentally proved the CM is effective for detection of multiple instruction skip. The second CM is proposed based on the bit level and instruction level FM. We noticed that the fault at bit level is bit-reset rather than bit-set. While the FM instruction level is related to a block of four instructions. Based on these facts, we constructed the CM by using SI instructions containing maximum bits of value 1 to detect the fault. Notice each faulted block needs to contain at least one SI. Experimental result shows that

the method is effective to detect skip of multiple instructions due to bit reset caused by FI.

Our future works will be centering on: (1) the validation of the methods proposed in this works on other devices, (2) the development of tool for automation of protected code generation, (3) **the more precise evaluation in code size and execution time overhead of the proposed CMs using statistical approaches**.

Acknowledgment

This work was partly funded by the SPARTA project, which has received funding from the European Union Horizon 2020 research and innovation programme under grant agreement number 830892.

References

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

[2] N. Selmane, S. Guilley, J.-L. Danger, Practical setup time violation attacks on aes, in: 2008 Seventh European Dependable Computing Conference, IEEE, 2008, pp. 91–96.

[3] J. Balasch, B. Gierlichs, I. 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, IEEE, 2011, pp. 105–114.

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

[5] R. Lashermes, M. Paindavoine, N. El Mrabet, J. J. Fournier, L. Goubin, Practical validation of several fault attacks against the miller algorithm, in: 2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, IEEE, 2014, pp. 115–122.

[6] L. Riviere, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer, L. Sauvage, High precision fault injections on the instruction cache of armv7-m architectures, in: 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), IEEE, 2015, pp. 62–67.

[7] A. Beckers, J. Balasch, B. Gierlichs, I. Verbauwhede, S. Osuka, M. Kinugawa, D. Fujimoto, Y. Hayashi, Characterization of em faults on atmega328p, in: International Symposium on Electromagnetic Compatibility. IEEE, 2019.

[8] S. P. Skorobogatov, R. J. Anderson, Optical fault induction attacks, in: International workshop on cryptographic hardware and embedded systems, Springer, 2002, pp. 2–12.

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

[10] J.-M. Dutertre, T. Riom, O. Potin, J.-B. Rigaud, Experimental analysis of the laser-induced instruction skip fault model, in: Nordic Conference on Secure IT Systems, Springer, 2019, pp. 221–237.

[11] A. Menu, J.-M. Dutertre, O. Potin, J.-B. Rigaud, J.-L. Danger, Experimental analysis of the electromagnetic instruction skip fault model, in: 2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS), IEEE, 2020, pp. 1–7.

[12] A. Menu, S. Bhasin, J.-M. Dutertre, J.-B. Rigaud, J.-L. Danger, Precise spatio-temporal electromagnetic fault injections on data transfers, in: 2019

Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), IEEE, 2019, pp. 1–8.

[13] V. Khuat, J.-L. Danger, J.-M. Dutertre, Laser fault injection in a 32-bit microcontroller: from the flash interface to the execution pipeline, in: 2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC), IEEE, 2021, pp. 74–85.

[14] C. Roscian, A. Sarafianos, J.-M. Dutertre, A. Tria, Fault model analysis of laser-induced faults in sram memory cells, in: 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, IEEE, 2013, pp. 89–98.

[15] V. Khuat, O. Trabelsi, L. Sauvage, J.-L. Danger, Multiple and reproducible fault models on micro-controller using electromagnetic fault injection, in: 2021 IEEE International Joint EMC/SI/PI and EMC Europe Symposium, IEEE, 2021, pp. 667–672.

[16] K. Matsuda, S. Tada, M. Nagata, Y. Komano, Y. Li, T. Sugawara, M. Iwamoto, K. Ohta, K. Sakiyama, N. Miura, An ic-level countermeasure against laser fault injection attack by information leakage sensing based on laser-induced opto-electric bulk current density, Japanese Journal of Applied Physics 59 (SG) (2020) SGGL02.

[17] N. Moro, K. Heydemann, E. Encrenaz, B. Robisson, Formal verification of a software countermeasure against instruction skip attacks, Journal of Cryptographic Engineering 4 (3) (2014) 145–156.

[18] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, F. Regazzoni, Countermeasures against fault attacks on software implemented aes: effectiveness and cost, in: Proceedings of the 5th Workshop on Embedded Systems Security, 2010, pp. 1–10.

[19] H. So, M. Didehban, J. Jung, A. Shrivastava, K. Lee, Chitin: A comprehensive in-thread instruction replication technique against transient faults,

in: 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, 2021, pp. 1440–1445.

[20] M. T. Inc, SAM D21/DA1 Family, in: SAM D21/DA1 Family.
URL https://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D21_DA1_Family_DataSheet_DS40001882F.pdf

[21] A. Limited, Arm®v6-m architecture reference manual, in: ARM Limited, ARM Limited, 2017.

[22] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, The sorcerer's apprentice guide to fault attacks, Proceedings of the IEEE 94 (2) (2006) 370–382.

[23] E. Trichina, R. Korkikyan, Multi fault laser attacks on protected crt-rsa, in: 2010 Workshop on Fault Diagnosis and Tolerance in Cryptography, IEEE, 2010, pp. 75–86.

[24] I. Koren, C. M. Krishna, Fault-tolerant systems, Elsevier, 2010.

[25] A. Barenghi, L. Breveglieri, I. Koren, D. Naccache, Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures, Proceedings of the IEEE 100 (2012) 3056 – 3076.

[26] J. Vankeirsbilck, N. Penneman, H. Hallez, J. Boydens, Random additive control flow error detection, in: Computer Safety, Reliability, and Security: 37th International Conference, SAFECOMP 2018, Västerås, Sweden, September 19-21, 2018, Proceedings 37, Springer, 2018, pp. 220–234.

[27] M. Didehban, A. Shrivastava, nzdc: A compiler technique for near zero silent data corruption, in: Proceedings of the 53rd Annual Design Automation Conference, 2016, pp. 1–6.

[28] M. Didehban, A. Shrivastava, S. R. D. Lokam, Nemesis: A software approach for computing in presence of soft errors, in: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), IEEE, 2017, pp. 297–304.

[29] T. Barry, D. Couroussé, B. Robisson, Compilation of a countermeasure against instruction-skip fault attacks, in: Proceedings of the Third Workshop on Cryptography and Security in Computing Systems, 2016, pp. 1–6.

[30] L. Cojocar, K. Papagiannopoulos, N. Timmers, Instruction duplication: Leaky and not too fault-tolerant!, in: Smart Card Research and Advanced Applications: 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13–15, 2017, Revised Selected Papers, Springer, 2018, pp. 160–179.

[31] J. Danger, A. Facon, S. Guilley, K. Heydemann, U. Kühne, A. Si Merabet, M. Timbert, Ccfi-cache: A transparent and flexible hardware protection for code and control-flow integrity, in: 2018 21st Euromicro Conference on Digital System Design (DSD), 2018, pp. 529–536. doi:10.1109/DSD.2018.00093.

[32] B. Yuce, C. Deshpande, M. Ghodrati, A. Bendre, L. Nazhandali, P. Schaumont, A secure exception mode for fault-attack-resistant processing, IEEE Transactions on Dependable and Secure Computing 16 (3) (2019) 388–401. doi:10.1109/TDSC.2018.2823767.

[33] O. T. Ltci, L. S. Ltci, J.-L. D. Ltci, Characterization of electromagnetic fault injection on a 32-bit microcontroller instruction buffer, in: 2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST), IEEE, 2020, pp. 1–6.

[34] V. Khuat, J.-M. Dutertre, J.-L. Danger, Analysis of a laser-induced instructions replay fault model in a 32-bit microcontroller, in: 2021 24th Euromicro Conference on Digital System Design (DSD), IEEE, 2021, pp. 363–370.