Constructing software countermeasures against instruction manipulation attacks: an approach based on vulnerability evaluation using fault simulator

Junichi Sakamoto^{1,2} · Shungo Hayashi¹ · Daisuke Fujimoto¹ · Tsutomu Matsumoto¹

Received: 23 September 2021 / Revised: 23 September 2021 / Accepted: 28 September 2021 / Published online: 23 November 2021
© The Author(s) 2021

Abstract

Fault injection attacks (FIA), which cause information leakage by injecting intentional faults into the data or operations of devices, are one of the most powerful methods compromising the security of confidential data stored on these devices. Previous studies related to FIA report that attackers can skip instructions running on many devices through many means of fault injection. Most existing anti-FIA countermeasures on software are designed to secure against instruction skip (IS). On the other hand, recent studies report that attackers can use laser fault injection to manipulate instructions running on devices as they want. Although the previous studies have shown that instruction manipulation (IM) could attack the existing countermeasures against IS, no effective countermeasures against IM have been proposed. This paper is the first work tackling this problem, aiming to construct software-based countermeasures against IM faults. Evaluating program vulnerabilities to IM faults is required to consider countermeasures against IM faults. We propose three IM simulation environments for that aim and compare them to reveal their performance difference. GDB (GNU debugger)-based simulator that we newly propose in this paper outperforms the QEMU-based simulator that we presented in AICCSA:1–8, 2020 in advance, in terms of evaluation time at most $\times 400$ faster. Evaluating a target program using the proposed IM simulators reveals that the IM faults leading to attack successes are classified into four classes. We propose secure coding techniques as countermeasures against IMs of each four classes and show the effectiveness of the countermeasures using the IM simulators.

Keywords Fault injection attack · Laser fault injection · Countermeasure · Instruction skip · Instruction replacement · Instruction manipulation

1 Introduction

Modern ICT (Information and Communication Technology) society is supported by many devices, such as smartphones and IC cards, containing confidential information. These devices provide confidentiality of internal data by using various kinds of security functions, such as access control and cryptography. Fault injection attacks (FIA) [1], which physically inject intentional faults into devices' operations, can bypass or corrupt the normal operation of the security functions. Increasing the number of devices around us in the IoT era enables attackers to obtain physical access to the target devices easily; therefore, protecting the devices from FIA becomes more critical to realize a secure information society.

Modeling possible faults [2] which attackers can inject is a necessary step to construct countermeasures against

✉ Junichi Sakamoto
sakamoto-junichi-ws@ynu.ac.jp

Shungo Hayashi
hayashi-shungo-kx@ynu.jp

Daisuke Fujimoto
fujimoto-daisuke-ht@ynu.ac.jp

Tsutomu Matsumoto
tsutomu@ynu.ac.jp

¹ Yokohama National University, 79-7 Tokiwadai, Hodogaya-ku, Yokohama 240-8501, Japan

² National Institute of Advanced Industrial Science and Technology, 2-3-26 Aomi, Koto-ku, Tokyo 135-0064, Japan

FIA. Previous studies report that many fault injection means can skip assembly instructions running on many types of architecture. Such a fault effect is modeled as instruction skip (IS) model and is widely studied as a possible threat to cryptography on actual devices [3, 4]. Some studies report that injecting IS at short intervals is difficult, and they propose anti-IS countermeasures under the assumption that IS does not occur at consecutive instructions [5, 6].

The instruction replacement (IR) model is another fault model on software, where the attackers can have the devices run faulty instructions, which are instructions replaced from originally programmed ones. Some experiment results report that IR faults were observed on some devices [7–9]. Most of the IR faults in these studies are uncontrollable; therefore, some studies conclude that the IR faults are not useful for establishing successful attacks [7]. On the other hand, recent studies [10–13] show that attackers can manipulate the instructions running on the devices partly but as attackers want, by laser fault injection (LFI). We call such faults *instruction manipulation (IM)* to distinguish them from the above (uncontrollable) IR. IM faults are pretty powerful than conventional IS and IR faults; however, the IM faults in the previous studies have some restrictions because the IM faults are injected by single-spot laser equipment. A single-spot laser—one of the significant restrictions—can manipulate only one specific bit of each target instruction. Even under this restriction, IM faults pose a significant threat to devices. For example, manipulating one bit of a branch instruction can cause a control flow alternation. To prevent such attacks by LFI, hardware countermeasures such as light detectors are essentially effective. Nevertheless, some manufacturers demand anti-FIA countermeasures without any hardware modification due to cost reasons.

1.1 Related works

In the general context of fault tolerance, faults like soft errors by radiations occur at random; therefore, time/space redundancy is effective since the probability that faults into redundant parts occur at the same time is considered negligible. Some compiler-based and formally provable fault mitigation methods make good results in such cases [14, 15]. On the other hand, attackers can inject faults into vulnerable points intentionally in the context of FIA. One challenging point to construct anti-FIA countermeasures on software is that the countermeasure part of the software can also be fault-injected by attackers. Previous studies related to anti-FIA countermeasures take care of the faults into the countermeasure part by restricting the number of faults or the frequency of faults. Moro et al., [6] propose an anti-IS countermeasure under the assumption where the attackers

can skip only one instruction. Barengi et al., [5] propose another anti-IS countermeasure under the assumption where attackers can skip instructions, but not consecutive instructions. On the other hand, Sakamoto et al., [11] report that attackers can inject multiple IM faults into consecutive instructions with laser since the laser effects on the electric circuit last during the laser duration. Constructing countermeasures against the IM is considered a more challenging task than the conventional software countermeasure constructions.

We presented a part of work in this paper in advance in [16], where countermeasures against IMs and evaluation results using QEMU-based IM simulator are shown. The QEMU-based IM simulator was useful to construct countermeasure against IM. We proposed four secure coding schemes against IM vulnerabilities found by the QEMU-based IM simulator. Meanwhile, the QEMU-based IM simulator has the drawback that the vulnerability evaluation takes a long time. It took approximately 132 min to evaluate a short (15 instruction) program. Reducing the computation time for IM simulation is a critical milestone to evaluate realistic application programs that contain a large number of instructions. This paper is the extended version of the previous work; therefore, it includes proposals of effective simulation environments for evaluating IM vulnerabilities and a computational complexity reduction technique in addition to the contents in [16].

1.2 Contributions

To make problems to be solved to construct anti-IM countermeasures clear, we first restrict IM attacker's ability and formally define it as the one-bit-set instruction manipulation (OSIM) model. This model is based on the fact that existing studies [10, 11] related to IM use single-spot lasers onto flash memories for injecting IM faults. Colombier et al., [10] explain that laser irradiation onto a flash memory causes only bit-set faults and that single-spot laser affects only one bit. Kumar et al., observe one-bit-reset IM (ORIM) faults on an AVR microcontroller [13]. Section 7 discusses the efficiency of our anti-OSIM countermeasure against ORIM. We do not care about faults injected into more than two bits in this paper in order to relax countermeasure construction. The bit adjacent ordering is device-dependent and can be too complicated because of some reasons, for example, bus scrambling. We construct the anti-OSIM countermeasure through an attack-based approach. We first investigate how the OSIM model attacks an existing anti-FIA countermeasure. For this purpose, we propose three simulation environments to simulate the effects of the OSIM model faults, and then we attack the existing anti-FIA countermeasure by the simulators. Among the simulators, the GDB-based simulator

outperforms the QEMU-based simulator presented in previous work in [16] in terms of evaluation time. However, the GDB simulator's performance is insufficient for evaluating large programs because the evaluation time increases exponentially to the number of instructions. Furthermore, we propose a computational cost reduction technique for OSIM simulation. The OSIM model can manipulate instruction bits only to '1' but not to '0'; therefore, we can eliminate simulating some IMs such as manipulations from '1' to '1'. This method reduces the computational complexity of the IM simulation from approximately the power of n to the power of $n/2$, where n is the number of instructions. Using the computation complexity reduction technique and the GDB-based simulator, we achieve approximately $\times 400$ faster evaluation when $n = 15$ compared to the results in [16].

A data integrity verification (DIV) scheme, which is often used as a common countermeasure against fault injection attacks (FIA) [1, 5, 17] is chosen as the attack target. The attack results are categorized into four classes. We propose coding methods as countermeasures to prevent each of these four attacks. The proposed coding methods improve the DIV, and the improved DIV implementation is attacked on the IM simulator to evaluate the efficiency of the proposed countermeasure.

The remainder of this paper is organized as follows. In Sect. 2, we provide works related to FIA and preliminaries for explaining the proposed methods. A formal description of the OSIM model and its computational complexity for simulation is presented in Sect. 3. A reduction technique of the computational complexity is also described in Sect. 3. Three different OSIM model simulation environments are presented in Sect. 4. In Sect. 5, a target program is evaluated using the three simulation environments, where the evaluated results show that the GDB-based simulator outperforms other simulators. Based on the evaluation results, the coding method as a countermeasure against the OSIM model are proposed in Sect. 6. The simulators also evaluate the countermeasure-applied implementation to confirm the efficiency of the proposed countermeasures. Discussions are given in Sect. 7, and finally, Sect. 8 concludes this paper.

2 Preliminaries and related works

2.1 Target instruction set architecture (ISA): ARM 16-bit Thumb ISA

We chose ARM 16-bit Thumb instruction set architecture (ISA) [18, Chapter 3] as a target for constructing a countermeasure against the OSIM model. ARM is one of the most successful processor architectures, and Thumb ISA is

implemented on devices after the ARMv4T family. 16-bit ISA is preferable for our aim of constructing a countermeasure against the OSIM model because a countermeasure on long instructions becomes too complex to construct. Many applications on modern ARM devices such as ARMv7 are written in 32-bit Thumb-2 ISA, which is a superset of Thumb ISA. In Thumb-2 ISA, 16-bit and 32-bit instructions can be mixed in a single program; therefore, the applications of the countermeasures written in Thumb are wide-ranging. Thumb ISA can access eight 32-bit registers $r0$ – $r7$, and has five condition flags, N (negative), Z (zero), C (carry), V (overflow), and Q (saturation), for representing the states of computational results. According to the states of these flags, conditional branch instructions determine whether to branch, for example, beq (branch if values are equal) instruction branches if the Z flag is set to 1. The assembly language codes shown in the rest of this paper are written in Thumb ISA.

2.2 Fault injection attack (FIA)

Fault attacks are physical attacks that inject intentional computation errors (so-called 'faults') and use them to compromise security. Since fault attacks can disturb the operation of normal algorithm flow, they can cause information leakage from secure implemented devices. Fault injection is performed by various means, such as clock glitches [7, 17, 19], underpowering [20], electromagnetic perturbation [8, 21, 22], laser irradiation [9, 11, 23], and body biasing [24]. As a simple example of a fault attack, Anderson and Kuhn introduce an attack that forcibly outputs sensitive data by changing addresses of output data [25].

Differential fault analysis (DFA) [26, 27] is a sophisticated technique for attacking cryptosystems. When injecting faults during cryptosystem computation, incorrect ciphertexts are output from the device. DFA computes the candidates of a secret key of the cryptosystem with sets of a correct (fault-free) and an incorrect (fault-injected) ciphertext.

2.3 Fault model

No correct operations are ensured on devices exposed to the threat of fault attacks; therefore, assuming realistic attackers' ability and making it the basis of trust is necessary for constructing countermeasures against fault attacks. Such a process is called 'fault modeling' [2]. Fault models strictly define the faults attackers can inject. Defining appropriate fault models based on an assumed attacker, we can construct secure countermeasures against the target fault model. While various levels of fault models

can be considered, defining suitable fault models is desirable depending on the required security level due to the higher cost of countermeasures against stronger fault models. The level of faults to be injected depends on the attackers' knowledge and their fault injection equipment. Instruction skip (IS) and instruction replacement (IR) models are known as popular fault models that attackers can inject into actual devices. Data integrity verification (DIV) schemes are widely used as countermeasures against DFA. The DIV schemes output no data if the verification is failed so that attackers cannot perform DFA.

2.3.1 Instruction skip (IS) model

IS model allows an attacker to skip an assembly instruction. Previous studies have reported the occurrence of IS faults for several fault injection means on various devices, such as the clock glitch on the 8-bit AVR ATmega163 [7] and LEON3 [19], electromagnetic irradiation on the 32-bit ARM Cortex-M3 [8] and the 32-bit ARMv7-M [21], and laser irradiation on the 32-bit ARM Cortex-M3 [9]. Actual key-retrieve attacks based on IS faults are reported by several papers [12, 19]; constructing countermeasures against IS is a major topic in fault attack studies.

2.3.2 Instruction replacement (IR) model

IR model allows an attacker to replace a running instruction with another. IS model can be considered part of IR model, wherein a target instruction is replaced with a NOP instruction; therefore, IR model faults represent a more significant threat than IS model faults. Some experiment results reported that IR faults were observed on some devices [7, 8, 12]. Balasch et al. reported the occurrence of IR faults on an ATmega163 smartcard via a clock glitch at the time of instruction pre-fetch [7]. In their study, some instructions, e.g., EOR (Exclusive OR) and SER (SEt all bits in Register), were targeted, and the instructions after replacement depend on the period of the clock glitch. However, Balasch et al. concluded that controlling IR faults is too complex to use for actual attacks. Similar results were observed by Moro et al. on an ARM Cortex-M3 micro-controller through electromagnetic glitches [8]. Similar to Balasch et al., Moro et al. could not control IR faults. Few studies focus primarily on IR, and none consider developing countermeasures against it.

2.3.3 Instruction manipulation (IM) model

On the other hand, recent studies [10, 11, 13] show that means of accurate fault injection, such as laser irradiation, can control IR faults and launch practical attacks. We call such faults *instruction manipulation (IM)* to distinguish

them from (uncontrollable) IR. In these studies, flash memories on the target devices are irradiated with lasers. This laser irradiation injects faults into the instruction fetching process; the bit position to be fault-injected depends on the irradiated points. If an attacker records the irradiated points and the corresponding faulty bit positions in advance, the attacker can replace a target instruction with another one. However, the authors in [10] describe that IM faults injected by laser irradiation pose some limitations. One of the major restrictions is that a single-spot laser can manipulate only one specific bit of each target instruction. Even under this restriction, IM faults pose a threat to devices.

2.3.4 One-bit set instruction manipulation (OSIM) model

OSIM model allows an attacker to set one bit of a machine code to 1. According to [10], only bit-set faults are observed on their target device; thereby, we assume that attackers can set a bit to 1 but not to 0. Moreover, we assume that the attacker uses single-spot laser equipment. This assumption limits the number of bits that can be fault-injected simultaneously. We assume that a single-spot laser can inject faults into only a single bit because it can illuminate only one spot at a time. These assumptions reflect the actual faults that attackers can inject. Note that the OSIM model does not define the number of faults that can be injected. Sakamoto et al., [11] shows that the number of laser-injected IM faults increases with a longer laser duration.

2.3.4.1 OSIM model attacker's ability We summarize an attacker's ability to inject OSIM faults with single-spot laser equipment as follows:

1. The attacker can set one bit per machine code to 1.
2. The attacker can inject the bit-set faults into multiple instructions.
3. The attacker cannot inject the bit-set faults into different bit positions during program execution.

We define the OSIM model formally according to the above conditions in the next section.

3 OSIM model and its computational complexity

This section describes how much computational complexity the OSIM simulator has and proposes its reduction technique. We first define the OSIM model formally and then evaluate the computational complexity. The computational complexity of the OSIM model is exponential to

the number of instructions. The reduction technique becomes more effective as the instructions increase.

3.1 Formal definition of the OSIM model

Definition 1 (Instruction code) Let the bit length of an instruction be w . An instruction code ins is represented as a w -dimensional row vector,

$$ins = (b_{w-1}, b_{w-2}, \dots, b_0), b_i \in \{0, 1\}$$

Definition 2 (Running program) A running program P is an $n \times w$ matrix consisting of n instructions,

$$P = \begin{pmatrix} ins_1 \\ ins_2 \\ \vdots \\ ins_n \end{pmatrix} = \begin{pmatrix} b_{1,w-1}, b_{1,w-2}, \dots, b_{1,0} \\ b_{2,w-1}, b_{2,w-2}, \dots, b_{2,0} \\ \vdots \\ b_{n,w-1}, b_{n,w-2}, \dots, b_{n,0} \end{pmatrix}, b_{i,j} \in \{0, 1\}$$

Note that P is not the program stored in ROM. The IM faults in previous studies inject bit-set faults into program memory read-out circuits rather than into ROM.

Definition 3 (One-bit-set instruction manipulation (OSIM) model) A fault vector fv is an n -dimensional column vector,

$$fv = \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix}, e_i \in \{0, 1\}$$

OSIM faults into the c th bit are represented as following;

$$P \parallel_c fv = \begin{pmatrix} b_{1,w-1}, b_{1,w-2}, \dots, b_{1,c} \parallel e_1, \dots, b_{1,0} \\ b_{2,w-1}, b_{2,w-2}, \dots, b_{2,c} \parallel e_2, \dots, b_{2,0} \\ \vdots \\ b_{n,w-1}, b_{n,w-2}, \dots, b_{n,c} \parallel e_n, \dots, b_{n,0} \end{pmatrix} = P',$$

where the operator $\parallel_c$ represents the component-wise OR operation between the right operand and the c th column of the left operand, and $\parallel$ represents bit OR operation.

Here, c corresponds to the irradiated position irradiated by laser—namely which bit position is targeted—and $e_i = 1$ corresponds to the laser irradiation timing. For example, the laser turns on at the timing of the fetching of the 1st, 3rd, and n th instructions when e_1 , e_3 , and e_n in a fv are equal to 1. Executing P' different from P by attackers' manipulation can cause confidential information leakage from the devices.

3.2 Computational complexity of the OSIM model

The abstract workflow of the OSIM model simulators presented in Sect. 4 is shown in Fig. 1. The .text section of the target executable file is manipulated according to all possible OSIM model faults by the python program, and the python program generates all the manipulated executables. Let w be the instruction length and n be the number of instructions to be targeted by the faults. In Thumb instruction, $w = 16$. The number of all possible OSIM model faults is $N_{all} = w \times 2^n$ because 2^n possible fault vectors can be injected into w bit positions. We can obtain the N_{all} results from the simulator and analyze them for constructing countermeasures. Since N_{all} is exponential to the number of instructions n , increasing n makes OSIM simulation difficult.

3.3 Computational complexity reduction technique

As shown above, the number of OSIM faults increases exponentially to the number of instructions n . Simulating all OSIM faults becomes more challenging by increasing the number of instructions in the examined program. We propose a technique that reduces the number of OSIM faults we must simulate. The idea is based on the OSIM model can manipulate instruction bits to '1' but not '0'; therefore, some fault vectors cause no effects on the program execution. Fig. 2 shows the classification of all possible fault vectors (FVs) injected to the first column of an examined program. In this situation, the number of all possible FVs is $N_{all} = 2^4 = 16$. Distinct 16 FVs, $fv_0, \dots, fv_{15}$, are classified into two groups: four ineffective FVs, which produce no change onto the program binaries, and effective FVs, which produce some changes onto the program binaries. Let $n_{i,1}$ be the number of '1' bits in the i th column. There are $2^{n_{i,1}}$ ineffective FVs in the i th column such as the figure. Clearly, we do not need to simulate the ineffective FVs. Remains (these are $2^n - 2^{n_{i,1}}$ FVs) are the effective FVs that can change the program's behavior. Actually, we do not need to simulate some effective FVs as we can know some of them to lead to the same results. For example, the four FVs ($fv_4, fv_6, fv_{12}, fv_{14}$) produces the same manipulated binaries because all of e_2 and e_4 are identical, which contribute the instruction manipulations. By simulating an FV in such the same group, we can know the execution results from the rest of the FVs without simulation. There are $(2^n - 2^{n_{i,1}})/2^{n_{i,1}} = 2^{n-n_{i,1}} - 1$ such the groups for every column: therefore, we can get the all 2^n results generated by every FV only simulating $2^{n_{i,0}} - 1$ executions. For entire program, we can get all $w \times 2^n$

Fig. 1 OSIM model simulator workflow

Fig. 2 Examples of effective and ineffective FVs

results by simulating only $\sum_{i=0}^{w-1} (2^{n_{i,0}} - 1)$ executions. Assumed '1' and '0' ratio in target programs as 1:1, the number of FVs we have to simulate becomes $w \times (2^{n/2} - 1)$, which is close approximately to $2^{n/2}$ by increasing the number of instructions n .

4 OSIM model simulator and its variations

This section describes some constructions of the OSIM model simulators. The OSIM model simulator workflow is already shown in Fig. 1, there are some variations by the execution environment of the workflow. The main computational bottleneck of the workflow is the execution time of the Thumb instruction set simulator (ISS). Our previous work uses the user-mode QEMU as the Thumb ISS, which takes approximately 10 ms for simulating each binary executable. This section proposes three different approaches for OSIM simulators, the user-mode QEMU-based, the full-system QEMU-based, and the GDB-based.

4.1 User-mode QEMU-based OSIM simulator

This approach uses user-mode QEMU (arm-qemu-static) as the Thumb ISS as shown in Fig. 3. The QEMU emulator has two execution modes, user-mode emulation and full-system emulation. The former achieves high performance by omitting to emulate the low-level parts of the system such as hard disk and Ethernet controllers. We adopted the user-mode QEMU ISS for our IM simulator presented in [16]. The concrete system description are given in Fig. 3. The python script (InstructionManipulation.py) executed on Ubuntu 20.04 Linux manages the entire OSIM simulation workflow. The python script launches arm-qemu-static as a child process of the python and passes the manipulated binary executable to the child process as an argument. Although user-mode emulation can quickly execute the passed binary executables, the child process generation is a large performance bottleneck because it involves launching a large arm-qemu-static system.

4.2 Full-system QEMU-based OSIM simulator

This approach uses full-system QEMU (runqemu core-image-minimal) as the Thumb ISS as shown in Fig. 4.

The full-system QEMU virtual machine is emulated on Ubuntu 20.04, and we can run arbitrary processes on the virtual machine. The python script for managing IM simulation is executed on the virtual machine. The python script directly launches the manipulated binaries as child processes on the virtual machine. Although this approach has the disadvantage that the emulation is slow, the

Fig. 3 User-mode QEMU-based OSIM simulator

Fig. 4 Full-system QEMU-based OSIM simulator

overhead by the process launching is expected relatively low because it launches only manipulated executables, not the entire QEMU system as the user-mode QEMU-based simulator.

4.3 GDB-based OSIM simulator

This approach uses GDB (GNU debugger) as the Thumb ISS as shown in Fig. 5.

Two processes, the python process for managing IM simulation and the GDB process for executing Thumb binary executables, run on Ubuntu 20.04. The python script passes the manipulated binaries to the GDB process via inter-process communication, and the GDB executes the received binaries and sends back the executed results. The GDB process is stationed in Ubuntu. The python script controls the GDB interactively with respect library. An advantage of this approach is that it requires no process launching for each binary execution. Therefore, this approach is expected as a promising solution to improve OSIM simulation performance. However, this approach has some disadvantages as well. First, this approach is considered unsuitable for simulating large systems such as applications on Linux as we have to prepare Linux kernel and shared libraries ourselves. On the other hand, this approach has sufficient abilities for simulating bare-metal applications and applications on small operating systems,

which are often used in embedded devices. Second, the arm simulator on the GDB (GDB 10.1) was not able to handle some exceptions such as undefined instruction and aborts exceptions. We made possible the exception handling by modifying some source codes.

5 OSIM simulators comparison

This section compares the different OSIM simulator approaches described before section by examining (attacking) a target program.

5.1 Plaintext integrity verification with default fail (PIVD)

We chose plaintext integrity verification with default fail (PIVD) [17] as the attack target for the OSIM simulators. PIVD, a DIV scheme, is designed as a countermeasure against IS faults. Listing 1 shows an example of the PIVD implementation written in Thumb. This program outputs CTXT (the encrypted outcome of PTXT) only if two plaintexts, PTXT and PTXT', are equal to prevent DFA. The encryption and decryption processes are executed by the EncAndDec function at line 11. We specify no cipher algorithm but implicitly take into account a famous block cipher such as AES-128. To verify a 128-bit block at one time, 32-bit hashed values of PTXT and PTXT' are stored in the addresses r3 and r3 + 0x04, respectively. This hash process is not essential. We avoid multiple-word comparison to ease anti-OSIM countermeasure construction. OutCTXT is a function that outputs CTXT (ciphertext). FaultHandler is a function that securely aborts the program to prevent fault attacks. The attacker tries to inject DFA faults during the encryption process to obtain faulty ciphertext; therefore, it is the attacker's goal to execute the OutCTXT function under the state that PTXT != PTXT'.

Listing 1 An example implementation of the plaintext integrity verification with default fail (PIVD) in Thumb.

```

1 .extern FaultHandler
2 .extern OutCTXT
3 ;OutCTXT function is located at a lower address from here.
4 PIVD:
5     b LBL_MAIN
6 LBL_CMP:
7     cmp r3, r1 ; Only if r3 == r1,
8     beq OutCTXT ; output CTXT.
9     b FaultHandler
10 LBL_MAIN:
11     b EncAndDec ;Encrypt PTXT to CTXT, and decrypt CTXT to PTXT'.
12     ldr r1, [r3, #0]; Load PTXT to r1.
13     ldr r3, [r3, #4]; Load PTXT' to r3.
14     mov r7, #1 ;Init Z flag to 0.
15     bne LBL_CMP
16 ;FaultHandler function is located at an upper address from here.
17 .endfunc

```

Fig. 5 GDB-based IM simulator

This program is written in an anti-IS manner called default fail [28], where important functions (OutCtxt) or instructions (at LBL_SUCCESS) are placed at a lower address than the conditional branch for integrity verification (at line 8). The default operation of the branch (LBL_FAIL and FaultHandler) is placed at a higher address. The default fail manner is secure against skipping the beq instruction because IS can increment a program counter but not decrement it. Note that this program is written to be intentionally vulnerable to the IM, to find as many as vulnerabilities for the attack-based countermeasure construction.

5.2 PIVD evaluation results by the OSIM model simulators

The results of the attacks on the PIVD by the simulators are shown in Table 1. In the attacks, PTXT and PTXT' are set to different values under the assumption that the attacker already injected faults during the encryption process. The parameters $n = 8$. The branch instruction at line 11 is excluded from the target of the OSIM faults because we assume that the faults are already injected during the execution of the EncAndDec function.

The table shows that the results of the user-mode QEMU-based and the full-system QEMU-based simulators are the same except for evaluation time. On the other hand, the GDB-based simulator exhibits few different results. The differences are considered originated from the difference that the GDB-based simulator executes programs as bare metal applications, but the QEMU-based simulators execute them as Linux applications (for example, program loading addresses and startup routines differ). The GDB-

based simulator poses approximately $7\times$ faster evaluation time compared to the user-mode QEMU-based simulation presented in [16]; therefore, it is suitable to evaluate bare metal and small os applications. Moreover, we apply the computation complexity reduction technique described in Sect. 3.3 to the GDB-based simulator. The evaluation results are shown in the last row in Table 1. The evaluation time is reduced to 1/5 compared to before applying the technique and 1/40 compared the user-mode QEMU-based simulator.

We found that the faults that lead to faulty ciphertext outputs can be categorized into four patterns. In the next section, we describe how the four-fault patterns cause faulty ciphertext outputs and propose secure coding methods as countermeasures against the OSIM faults.

6 OSIM model attacks on the PIVD and the proposal of countermeasures against them

This section describes what OSIM faults cause to defeat the PIVD and proposes coding methods secure against such OSIM faults.

6.1 Operand register manipulation (ORM)

This is an attack manipulating the operand register of the cmp instruction at line 7 as shown in Fig. 6. The cmp instructions in Thumb take two operands, a source register Rs and a destination register Rd. In the original program, two distinct r1 and r3 are used for the source register Rs and destination register Rd, respectively. When the contents of the two registers differ, the PIVD program detects a fault injection and enters the fault handler function during normal flow. Attackers can have the cmp instruction compare the same register by injecting the OSIM fault shown in Fig. 6. In this case, the cmp instruction always sets the Z flag to 1; therefore, the integrity verification process passes for any values. Similar faults are used to increase the number of attack trials by changing a counter register to another one [10].

Table 1 Evaluation results on original PIVD (Listing 1) by different OSIM simulator approaches

Simulator approach	Faulty ciphertext	Fault detected	EXP4*	EXP11*	EXP19*	Evaluation time
User-mode QEMU-based [16]	528	2368	288	848	64	60 s
Full-system QEMU-based	528	2368	288	848	64	180 s
GDB-based	400	2304	352	976	64	8 s
GDB-based w/computational complexity reduction	400	2304	352	976	64	1.5 s

*EXP4, EXP11, and EXP19 represent undefined instruction, segmentation fault, and timeout exceptions, respectively

Fig. 6 Cmp instruction machine code format in Thumb ISA and an IM attack example for bypassing integrity verification process

6.1.1 Countermeasure against ORM

To prevent the above attack on the cmp instruction, we need to properly choose the fields of Rs and Rd. The OSIM-model attackers cannot inject faults into more than two bits; therefore, the attackers cannot do $Rs == Rd$ if the two registers satisfy the following expression;

$$HD(bin(Rs), bin(Rd)) > 1, \quad (1)$$

where $HD(,)$ is a function that evaluates the Hamming distance of the two input operands, and $bin(reg)$ means the binary expression of the reg .

6.2 Branch address manipulation (BAM)

This is the attack that forcibly executes the OutCTXT function by manipulating the address of the unconditional branch instructions at line 9. Unconditional branch instruction in Thumb ISA indicates the branch address by two's complement program counter (pc) relative addressing as shown in Fig. 7. FaultHandler function is located at a higher address than the PIVD since it follows the default fail manner; therefore, the sign bit (the 10th bit) of the unconditional branch instruction is zero. The OSIM fault at the sign bit changes the branch address to a lower address than the PIVD. If the location of the OutCTXT function coincidentally corresponds with the changed address as shown in Fig. 7, the attackers can obtain faulty ciphertexts.

6.2.1 Countermeasure against the BAM

To prevent the above attack on the address of the branch instruction, a FaultHandler function is placed at the address

Fig. 7 Unconditional branch machine code format in Thumb ISA and an IM attack example for bypassing the integrity verification process

Fig. 8 Load instruction machine code format in Thumb ISA and an IM attack example for bypassing the integrity verification process

between the OutCTXT function and PIVD. By placing a FaultHandler function at a lower address than the PIVD, the bit-set fault in the 10th bit of the branch instruction affects nothing because the 10th bit is already one before the fault injection. Other OSIM faults in the address field (the 9th–0th bits) change the branch destination to higher addresses than FaultHandler function. OutCTXT function is placed at a lower address than FaultHandler function to prevent unexpected reaches by such address manipulations.

6.3 Load address manipulations (LAM)

Figure 8 shows the attack that changes the value to be compared by manipulating the load address of the load instruction (ldr) at line 12. The ldr instruction loads the value at the address indicated r3 into r1. Since the address of the PTXT is stored in r3, PTXT is loaded into r3 in the case without any fault injections; however, manipulating the ldr machine code as shown in Fig. 8 changes the operation of the ldr instruction to loading PTXT'. As a result, the PIVD program compares the same value and always outputs ciphertexts, even if faults are injected.

Figure 9 shows another example of the ldr instruction manipulation leading to successful attacks. This is an example of the attack by a multiple-instruction manipulation. From the OSIR model assumption, the attacker can inject bit-set faults into the same bit position of multiple instructions. The faults are injected into the 7th bits of the ldr instructions (offset fields) so that unrelated values are loaded to both registers for integrity verification, r1, and r3. Although we don't know what values are stored at unrelated addresses, such as those shown in the figure ($r3 + 8$

Fig. 9 Load instruction machine code format in Thumb ISA and an IM attack to bypass integrity verification by manipulating the load addresses

and $r3 + 1$), zeros were loaded in the experiment. As a result, the PIVD program always outputs ciphertexts. This is because the simulator environment (QEMU) initializes the values at unused addresses to zeros. The fault at the 7th bit is one example of this type of attack. The faults at other bits of the offset, Rb, and Rd fields can lead to this type of attack.

6.3.1 Countermeasure against LAM

Choosing proper offset in binary, similar to the countermeasure against ORM, is effective against the first LAM attack. To prevent the OSIM faults from manipulating the address of PTXT to the address of PTXT', the offsets from the base address must satisfy the following expression:

$$HD(\delta_{PTXT}, \delta_{PTXT'}) > 1, \quad (2)$$

where $HD(,)$ is a function that evaluates the Hamming distance of the two input operands, and δ_{PTXT} represents the offset of PTXT from the base address.

We propose unused memory randomization as a countermeasure against the second LAM. This randomization requires that different values are stored in all unused addresses, but strong randomness is not required for the values. This countermeasure might consume a lot of time if the device has a vast address space; however, the necessity of this countermeasure is device-dependent because the treatment of unused addresses differs by device. We consider that unused memory randomization is unnecessary for most embedded devices because the SRAM values become unstable when booting.

6.4 Branch condition manipulation (BCM)

This is the attack that manipulates the branch condition field of the beq instruction at line 8. An actual BCM attack is performed on an ARM SC100 device [11]. The conditional branch is one of the most important instructions for determining software control flow. Manipulating the operation of the conditional branch can lead to remarkable information leakage. As shown in Fig. 10, the branch instruction in Thumb ISA determines the branch condition by the four-bit condition field (the 11th–8th bits). When the condition field is 0b0000, it is decoded to beq instruction, which branches if $Z = 1$. The beq instruction is used in the PIVD for jumping the OutCTXT function only if PTXT is equal to PTXT'. When an attacker injects a fault into the 8th bit of the beq instruction, the beq instruction changes to a bne instruction holding the same branch address. The bne instructions branch if $Z = 0$; therefore, this IM changes the PIVD program to a program that outputs CTXT when PTXT is not equal to PTXT'. Bit-set faults at other bits of

the condition field cause similar effects. For example, a fault at the 9th bit changes the beq instruction to a bcs instruction which branches if $C = 1$. The C flag is set to 1 by the cmp instruction at line 7 when $r1 \geq r3$. We consider that the attacker cannot control the PTXT' value, and half of the cmp executions set C flag to 1. Since the IM to the bcs instruction outputs CTXT with high probability, constructing countermeasures against the IM to bcs (and also bmi and bhi) is necessary for preventing DFA, not only the IM to bne.

6.4.1 Countermeasure against the BCM

The BCM attacks change the opcode part of the instructions, unlike other attacks which change the operand part; therefore, a countermeasure against BCM naturally requires a tricky method, unlike countermeasures against other OSIM faults. As shown in Listing 2, the countermeasure against BCM requires some extra instructions. The instructions at lines 14–16 are a comparison process instead of a cmp instruction, and the branch instruction at line 17 is responsible for determining if verification succeeds or fails. The following instructions in lines 8–12 are responsible for detecting BCM attacks. Consider if the beq instruction at line 17 is manipulated to a bne instruction as an example. If the beq instruction at line 17 is changed to other conditional branches by BCM, the identical conditional branch can detect the fault and jumps to the FaultHandler function. Consider if the beq instruction at line 17 changes to a bcs instruction through the 9th-bit manipulation. In this case, the attacker can pass the verification at line 17 when the C flag is one; however, the attacker cannot reach the OutCTXT function because of another bcs instruction at line 9. Moreover, the attacker cannot inject faults into the second bcs instruction because its 9th bit is originally one.

Fig. 10 Conditional branch instruction machine code format in Thumb ISA and IM attacks to bypass integrity verification by manipulating the branch condition

Listing 2 An example of the anti-OSIM PIVD implementation.

```

1 .extern FaultHandler
2 .extern OutCTXT
3     ;OutCTXT function is located at a lower address from here.
4     ;FaultHandler function is located at an upper address from here.
5 anti_OSIM_PIVD:
6     b LBL_MAIN
7 LBL_COND_BRANCH:
8     bne FaultHandler
9     bcs FaultHandler
10    bmi FaultHandler
11    bhi FaultHandler
12    beq OutCTXT
13 LBL_CMP:
14    mvn r3, r3        ; Only if r3 == r5,
15    add r3, r5        ;
16    mvn r3, r3        ; set Z flag to 1.
17    beq LBL_COND_BRANCH
18    b FaultHandler
19 LBL_MAIN:
20    b EncAndDec        ;Encrypt PTXT to CTXT, and decrypt CTXT to PTXT'.
21    mov r5, #1        ;Init Z flag to 0.
22    ldr r5, [r3, #0]
23    ldr r3, [r3, #12]
24    bne LBL_CMP
25 .endfunc

```

Note that the program in Listing 2 outputs ciphertexts only if the condition flags $(Z, C, N) = (1, 0, 0)$; therefore, the comparison process must satisfy this condition when the values in the compared two registers are equal. Cmp instruction is unsuitable for this purpose because it makes $(Z, C, N) = (1, 1, 0)$ when the values in the two registers are equal. The instruction sequence at lines 14–16 satisfies this condition¹. This sequence first calculates one's complement of r3, and then adds it to r5, and finally, calculates one's complement of r5. The outcome of r5 becomes zero only if $r3 == r5$ from the definition of one's complement.

6.5 Evaluation of the proposed countermeasures

A PIVD implementation written in the proposed anti-OSIM manners is shown in Listing 2.

The anti-OSIM PIVD implementation is evaluated by two OSIM simulators. Table 2 shows the evaluation results by the user-mode QEMU-based simulator. Anti-OSIM PIVD implementation causes no faults though it has a large number of extra instructions ($n = 15$). This result shows that the proposed countermeasures are effective in corresponding attacks. On the other hand, the evaluation time also largely increases by increasing the number of instructions. For increasing the number of instructions just twice, the evaluation time approximately becomes 132 times.

Table 3 shows the evaluation results by the GDB-based OSIM simulator with computational complexity reduction technique. The evaluation time gets approximately $400\times$

faster than the user-mode QEMU-based simulator for $n = 15$.

7 Discussion

7.1 Limitation of OSIM model simulator

Although our OSIM model simulator simulates the fault effects by manipulating the bits of executable files, actual OSIM faults are injected into running codes. Therefore, the simulator cannot simulate the OSIM model perfectly; for example, the cases that a manipulated instruction in a loop is executed multiple times. In this case, the simulator can simulate only the situation when faults are injected into all of the looped instructions. The actual OSIM model can choose which instructions are fault-injected into. We do not care about such cases because the target PIVD implementation has no loop.

Moreover, our proposed countermeasures do not ensure 100% fault tolerance because whether faulty ciphertexts are output is dependent on the initial device states such as register and memory values. Simulating all possible register and initial memory values is impossible. Even in the current settings (simulating only one initial value), the simulation takes approximately 1.5 and 20 sec for $n = 8$ and 15, respectively. While we achieve a large evaluation time reduction in this paper, we consider that the vulnerability evaluation by the OSIM simulators is not applicable for large programs since the simulation time increases the order of exponentiation for the number of instructions n . We estimate that evaluating an $n = 50$ program takes about a day. Mitigating the attacker's model, for example, restricting the number of instructions the attacker can

¹ Eor instruction also satisfies this condition; however, we found that eor is vulnerable to OSIM faults. Bit-set faults in the 7th bit of eor changes eor to lsr (logical shift right) instruction. Lsr instructions set Z flag to 1 in most cases.

Table 2 Evaluation results on original PIVD (Listing 1) and proposed anti-OSIM PIVD (Listing 2) using the user-mode QEMU-based OSIM simulator

Target implementation	Faulty ciphertext	Fault detected	EXP4*	EXP11*	EXP19*	<i>n</i>	Evaluation time
Original PIVD	528	2368	288	848	64	8	60 s
Anti-OSIM PIVD	0	346112	45056	116736	16384	15	132 min

*EXP4, EXP11, and EXP19 represent undefined instruction, segmentation fault, and timeout exceptions, respectively

Table 3 Evaluation results on original PIVD (Listing 1) and proposed anti-OSIM PIVD (Listing 2) using the GDB-based OSIM simulator with computational complexity reduction technique

Target implementation	Faulty ciphertext	Fault detected	EXP4*	EXP11*	EXP19*	<i>n</i>	Evaluation time
Original PIVD	400	2304	352	976	64	8	1.5 s
Anti-OSIM PIVD	0	342528	44032	121344	16384	15	20 s

*EXP4, EXP11, and EXP19 represent undefined instruction, segmentation fault, and timeout exceptions, respectively

manipulate at once and improving simulator performance is effective in simulating large programs.

7.2 Performance evaluation

The proposed countermeasures against the ORM, LAM, BAM, which modify the operand of instructions, have almost no effect on device performance. We need to take the performance decline by the countermeasure against BCM and the unused memory randomization into account. The countermeasure against BCM adds some instructions into the original PIVD implementation. These extra instructions make the program footprint and the execution time worse. However, the overhead of the extra instructions can be negligible compared to the encryption and decryption process.

Unused memory randomization countermeasure might consume a lot of time if the device has a vast address space; however, the necessity of this countermeasure is device-dependent because the treatment of unused addresses differs by device. We consider that the unused memory randomization is unnecessary for most embedded devices because SRAM values become unstable when booting.

7.3 Application to other programs and ISAs

Although the proposed countermeasure targets to protect only the narrow code snippet, the data verification process is a basis for other security mechanism; therefore, we consider the proposed method can be embedded in other programs. One example is an instruction duplication scheme [5]. This scheme executes all ldr instructions in AES encryption twice and verifies all the loaded values. Since the verification processes are almost the same as the

PIVD, we consider our proposed method is also applicable to the instruction duplication scheme. Automating finding out the parts vulnerable to the OSIM faults in large programs and replace them with the proposed anti-OSIM code remain as future works.

While we developed an anti-OSIM countermeasure for ARM Thumb ISA, Kumar et al., report that IM faults on an AVR microcontroller [13]. We suppose that the proposed countermeasures against ORM, BAM, and LAM attacks are effective in protecting other ISAs because they require only operand modification. On the other hand, the proposed countermeasures against the BCM attack, which requires operand modification, likely not to apply other ISAs in the same manner. How operand modification influences a program behavior is strongly ISA-dependent and hard to predict; therefore, we must carefully investigate the effects using a simulator.

Furthermore, Kumar et al. observe one-bit-reset IM (ORIM) faults on an AVR device rather than OSIM faults. This implies that the direction of manipulation (set/reset) is device-dependent. We must use different countermeasures depending on the fault models. For example, the inequality signs in inequalities (1) and (2) must be reversed to make the proposed countermeasures effective to anti-ORIM faults.

8 Conclusion

We propose secure coding schemes as countermeasures against the OSIM model, which is one of the IM models. Constructing countermeasures against IM model on software is a big challenge because the attackers can inject faults into the countermeasure part of the software. We construct countermeasure against the OSIM model by

attack-based approach. Three simulation environments for simulating the OSIM faults has been constructed, and we investigated which parts of the software are vulnerable to the OSIM faults. We have proposed the countermeasure schemes according to the simulation results and shown that the effectiveness of the countermeasures by the attacks using the simulator.

In addition, we propose OSIM simulation complexity reduction technique and the GDB-based OSIM simulator to For outcome of the improvements, and evaluation time is reduced by at most $400\times$.

Author contributions All authors contributed equally to the final dissemination of the research investigation as a full article. All authors have read and agreed to the published version of the manuscript.

Funding A part of this work was supported by the Cabinet Office (CAO), Cross-ministerial Strategic Innovation Promotion Program (SIP), “Cyber Physical Security for IoT Society”, JPNP18015 (Funding agency: NEDO).

Data availability Not applicable.

Code availability This paper uses a modified version of GDB (GNU debugger) v10.1, which is an open source code provided by <https://www.gnu.org/software/gdb/download/>.

Declarations

Conflict of interest Not applicable.

Ethical approval Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer’s apprentice guide to fault attacks. *Proc. IEEE* **94**(2), 370–382 (2006). <https://doi.org/10.1109/JPROC.2005.862424>
- Verbauwhede, I., Karaklajic, D., Schmidt, J.: The fault attack jungle—a classification model to guide you. In: 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 3–8. (2011). <https://doi.org/10.1109/FDTC.2011.13>
- Dehbaoui, A., Mirbaha, A. P., Moro, N., Dutertre, J.-M., Tria, A.: Electromagnetic glitch on the AES round counter. In: *Constructive Side-Channel Analysis and Secure Design*, pp. 17–31. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-40026-1_2
- Schmidt, J., Herbst, C.: A practical fault attack on square and multiply. In: 2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 53–58. (2008). <https://doi.org/10.1109/FDTC.2008.10>
- Barengi, A., Breveglieri, L., Koren, I., Pelosi, G., Regazzoni, F.: Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In: *Proceedings of the 5th Workshop on Embedded Systems Security, WESS ’10*. ACM, New York (2010). <https://doi.org/10.1145/1873548.1873555>
- Moro, N., Heydemann, K., Encrenaz, E., Robisson, B.: Formal verification of a software countermeasure against instruction skip attacks. *J. Cryptogr. Eng.* **4**(3), 145–156 (2014). <https://doi.org/10.1007/s13389-014-0077-7>
- Balasch, J., Gierlichs, B., Verbauwhede, I.: 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, FDTC, pp. 105–114. (2011). <https://doi.org/10.1109/FDTC.2011.9>
- Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In: 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 77–88. (2013). <https://doi.org/10.1109/FDTC.2013.9>
- Trichina, E., Korkikyan, R.: Multi fault laser attacks on protected CRT-RSA. In: 2010 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2010, Santa Barbara, California, 21 August 2010, pp. 75–86. (2010). <https://doi.org/10.1109/FDTC.2010.14>
- Colombier, B., Menu, A., Dutertre, J.-M., Moëllic, P.-A., Rigaud, J.-B., Danger, J.-L.: Laser-induced single-bit faults in flash memory: instructions corruption on a 32-bit microcontroller. In: 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 1–10. (2019). <https://doi.org/10.1109/HST.2019.8741030>
- Sakamoto, J., Fujimoto, D., Matsumoto, T.: Laser-induced controllable instruction replacement fault attack. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* (2020). <https://doi.org/10.1587/transfun.2019CIP0028>
- Kumar, D. S., Patranabis, S., Breier, J., Mukhopadhyay, D., Bhasin, S., Chattopadhyay, A., Baksi, A.: A practical fault attack on ARX-like ciphers with a case study on chacha20. In: 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 33–40. (2017). <https://doi.org/10.1109/FDTC.2017.14>
- Kumar, D.S.V., Beckers, A., Balasch, J., Gierlichs, B., Verbauwhede, I.: An in-depth and black-box characterization of the effects of laser pulses on atmega328p. In: *Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018*, vol. 11389 of *Lecture Notes in Computer Science*, pp. 156–170. Springer (2018). https://doi.org/10.1007/978-3-030-15462-2_11
- Martinez-Alvarez, A., Cuenca-Asensi, S., Restrepo-Calle, F., Palomo Pinto, F.R., Guzman-Miranda, H., Aguirre, M.A.: Compiler-directed soft error mitigation for embedded systems. *IEEE Trans. Dependable Secure Comput.* **9**(2), 159–172 (2012). <https://doi.org/10.1109/TDSC.2011.54>
- Fey, G., Staber, S., Bloem, R., Drechsler, R.: Automatic fault localization for property checking. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **27**(6), 1138–1149 (2008). <https://doi.org/10.1109/TCAD.2008.923234>

16. Sakamoto, J., Hayashi, S., Fujimoto, D., Matsumoto, T.: How to code data integrity verification secure against single-spot-laser-induced instruction manipulation attacks. In: 2020 IEEE/ACS 17th International Conference on Computer Systems and Applications (AICCSA), pp. 1–8. IEEE Computer Society, Los Alamitos (2020). <https://doi.org/10.1109/AICCSA50499.2020.9316521>
17. Endo, S., Homma, N., Hayashi, Y., Takahashi, J., Fuji, H., Aoki, T.: A multiple-fault injection attack by adaptive timing control under black-box conditions and a countermeasure. In: COSADE, vol. 8622 of Lecture Notes in Computer Science, pp. 214–228. Springer (2014). https://doi.org/10.1007/978-3-319-10175-0_15
18. ARM: ARM architecture reference manual Thumb-2 supplement (2015). <http://class.ece.iastate.edu/cpre288/resources/docs/Thumb-2SupplementReferenceManual.pdf>
19. Yuce, B., Ghalaty, N. F., Santapuri, H., Deshpande, C., Patrick, C., Schaumont, P.: Software fault resistance is futile: effective single-glitch attacks. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC, pp. 47–58. (2016). <https://doi.org/10.1109/FDTC.2016.21>
20. Barenghi, A., Bertoni, G. M., Breveglieri, L., Pellicoli, M., Pelosi, G.: Low voltage fault attacks to AES. In: 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 7–12. (2010). <https://doi.org/10.1109/HST.2010.5513121>
21. Rivière, L., Najm, Z., Rauzy, P., Danger, J.-L., Bringer, J., Sauvage, L.: High precision fault injections on the instruction cache of armv7-m architectures. In: 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 62–67. (2015). <https://doi.org/10.1109/HST.2015.7140238>
22. Menu, A., Bhasin, S., Dutertre, J.-M., Rigaud, J.-B., Danger, J.-L.: Precise spatio-temporal electromagnetic fault injections on data transfers. In: 2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 1–8. (2019). <https://doi.org/10.1109/FDTC.2019.00009>
23. Rodriguez, J., Baldomero, A., Montilla, V., Muijal, J.: LLFI: Lateral laser fault injection attack. In: 2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 41–47. (2019). <https://doi.org/10.1109/FDTC.2019.00014>
24. Beringuier-Boher, N., Lacruce, M., El-Baze, D., Dutertre, J.-M., Rigaud, J.-B., Maurine, P.: Body biasing injection attacks in practice. In: Proceedings of the Third Workshop on Cryptography and Security in Computing Systems, CS2 '16, pp. 49–54. ACM, New York (2016). <https://doi.org/10.1145/2858930.2858940>
25. Anderson, R., Kuhn, M.: Tamper resistance: a cautionary note. In: Proceedings of the Second USENIX Workshop on Electronic Commerce—Volume 2, WOECC'96, pp. 1–1. USENIX Association, Berkeley (1996)
26. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Advances in Cryptology—EUROCRYPT '97, pp. 37–51. Springer, Berlin (1997). https://doi.org/10.1007/3-540-69053-0_4
27. Giraud, C.: DFA on AES. In: Advanced Encryption Standard—AES, pp. 27–41. Springer, Berlin (2005). https://doi.org/10.1007/11506447_4
28. Witteman, M.: Secure application programming in the presence of side channel attacks (2018). https://www.riscure.com/uploads/2018/11/201708_Riscure_Whitepaper_Side_Channel_Patterns.pdf

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

based fault attack. He is currently interested in remote side-channel attacks and hardware implementation of homomorphic encryptions.

Shungo Hayashi is a master course student working on information security at Yokohama National University, Japan. He received B.S. degree in Information Technology from Yokohama National University in 2016. He is currently working on researches against embedded systems.

security cores. He is a member of IEEE and IEICE.

Daisuke Fujimoto received B.E., M.E., and Ph.D. degrees from Kobe University, Japan, in 2009, 2011 and 2014, respectively. He is currently an assistant professor at the Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan. He is also a visiting assistant professor in the Institute of Advanced Sciences, Yokohama National University. His research interests include hardware security and the implementation of

Tsutomu Matsumoto is a professor of the Faculty of Environment and Information Sciences, Yokohama National University, and the Director of the Cyber Physical Security Research Center at the National Institute of Advanced Industrial Science and Technology. He received a Doctor of Engineering degree from the University of Tokyo in 1986. He has been interested in research and education of Embedded Security Systems such as IoT Devices,

Cryptographic Hardware, In-vehicle Networks, Instrumentation and

Control Security, Tamper Resistance, Biometrics, Artifact-metrics, and Countermeasure against Cyber-Physical Attacks. He serves as the chair of the Japanese National Body for ISO/TC68 (Financial Services) and the Cryptography Research and Evaluation Committees (CRYPTREC) and as an associate member of the Science Council of Japan (SCJ). He received the IEICE Achievement Award, the DoCoMo Mobile Science Award, the Culture of Information Security Award, the MEXT Prize for Science and Technology, and the Fuji Sankei Business Eye Award.