Inferred Fault Models for RISC-V and Arm: A Comparative Study

Ihab Alshaer, Ahmed Al-Kaf, Valentin Egloff, Vincent Beroulle

Inferred Fault Models for RISC-V and Arm: A Comparative Study

Ihab Alshaer*, Ahmed Al-kaf*,Valentin Egloff*, Vincent Beroulle*
*Univ. Grenoble Alpes, Grenoble INP, LCIS, 26000 Valence, France

***Abstract*—With the widespread adoption of embedded systems, security issues became a major concern. In particular, such systems are vulnerable to various kinds of physical attacks, and fault injection is one of the main physical attacks. Designers and developers require fault models to predict the effects of the fault injection, so that they can analyze possible vulnerabilities and develop countermeasures against such attacks. Thus, understanding the effects of fault injection is essential to provide realistic fault models. Moreover, many of the systems currently in use or planned for future deployment incorporate either Arm or RISC-V processors. In this paper, voltage glitch campaigns have been carried out on two microcontrollers that are widely used in the embedded system market. One embeds a RISC-V core, where the other embeds an Arm Cortex-M4 core. As a result, we provide comprehensive analysis for the obtained faulty behaviors using a set of inferred fault models. We show that the presented fault models are able to explain more than 99% of the observed faulty behaviors. We also show that some of these models are applicable to both cores. Furthermore, we illustrate that some of the presented models are also comparable to state-of-the-art models that are proposed as a result of clock glitch campaigns. The presented fault models enable better understating of the fault injection effects, and thus, easing the process of analyzing vulnerabilities, and developing cost-effective countermeasures against fault attacks.**

***Index Terms*—fault injection attack, RISC-V, Arm Cortex-M, fault model**

I. Introduction

The widespread use of embedded systems open the doors to various considerations that focus on performance, cost, complexity, and most importantly, security. Such systems are under questions when it comes to their immunity to physical attacks, with fault injection attacks being among the most significant.

In the context of hardware security, fault injection can be defined as a powerful active physical attack, where the attacker tries to perturb the normal behavior of a system by introducing faults. The potential faulty outcome is then examined for a possible vulnerability to be exploited. There are different techniques to perform the injection. The most known ones are: applying perturbations to the clock signal, which is known as clock glitch [1], applying variations to the power supply, which is known as voltage glitch [2], and exposing the device to electromagnetic (EM) pulses [3] or to laser beams [4].

For the sake of protecting embedded systems against fault attacks, a comprehensive understanding of the fault effect is required to provide fault models. Fault models are abstract representations of the physical fault effects. These models provide description for the effects of the faults at different levels of system abstraction. Hardware designers and software developers need such fault models, so that, they can predict the possible effects by applying these fault models. Thus, they will be able to identify possible vulnerabilities in software codes and hardware designs. Consequently, they will be be able to develop/design the most suitable countermeasures. Nevertheless, having inaccurate fault models would lead to proposing either excessive protections, which affects the cost/performance ratio, or insufficient countermeasures, which means potential vulnerabilities are still exploitable.

979-8-3503-6688-4/24/$31.00 ©2024 European Union

A. Related works

Numerous studies [2]–[9] have focused on analyzing the effects of fault injection at the instruction set architecture (ISA) level, leading to the proposal of various fault models. For example, instruction skip. [4], [6], [8], [9], instruction replay [4], [6], instruction corruption [2], [3], [7], [9], and register corruption [2], [5], [7]. These models appear to be quite generic, lacking precision in accurately portraying the actual consequences of fault injection. The term ”corruption” lacks clarity in describing the fault’s impact. Thus, there will be a difficulty in identifying vulnerabilities solely based on this information. As a result, this may lead to developing over- or under-protections.

In [1], [10], the authors provided a comprehensive analysis and rationale for the experimental findings, resulting from clock glitch fault injection campaigns on 32-bit microcontrollers embedding Arm Cortex-M3 and Arm Cortex-M4 cores. They demonstrated how the alignment of instructions in memory can influence the resulting faulty behaviors. Building on this insight, they introduced two fault models at the instruction encoding level: *Skip* specific number of bits and *Skip and repeat* specific number of bits. This number of bits is related to the flash memory access size or a cache line size, such as 32 bits or 64 bits. Additionally, they proposed another fault model, called *Partial update* [11], where some bits, while the data are propagated from memory to pipeline stages, will be updated correctly, and others will be updated in a faulty way. This faulty part received its update either from the previous value or from a precharge value. These fault models allowed explaining a large set of the observed faulty behaviors at ISA level. Nevertheless, the authors were not sure if such fault models would be applicable on different microcontrollers, or when employing an injection technique that is different from clock glitch. Table I summarizes the related works.

TABLE I
SUMMARY OF RELATED WORKS ON FAULT INJECTION EFFECT CHARACTERIZATION AND MODELING.

Reference	Injection type	Target	Fault model
[2]	voltage	ARMv7-A	instruction corruption
[3]	EM	Arm Cortex-M3	instruction replacement
[5]	EM	Arm Cortex-M4	register, xPSR corruption, opcode or operand substitution
[6]	EM	Arm Cortex-A9	register corruption, operand substitution, control-flow corruption
[7]	EM	Arm Cortex-A53, Intel i3-6100T	register corruption, instruction corruption
[4]	laser	Arm Cortex-M0+	instructions skip, skip and replay
[8]	laser	Arm Cortex-M4	multi-instruction skip
[9]	EM	RISC-V E31	instruction corruption, instruction skip
[1], [10], [11]	clock	Arm Cortex-M3, Arm Cortex-M4	Skip # of bits, Skip and repeat # of bits, Partial update
This work	voltage	RISC-V E31 Arm Cortex-M4	Skip # of bits, Skip & repeat # of bits, Non-sequential skip & repeat, Partial update, Combination, Skip with forwarding

B. Contributions

In this paper, we present a set of inferred fault models that are able to describe and explain more than **99 %** of the obtained faulty behaviours from voltage glitch fault injection campaigns on two different 32-bit microcontrollers embedding RISC-V and Arm Cortex-M4 processors. We show that some of these models are applicable to both devices. This includes *Skip*, *Skip & repeat*, and *Partial update* fault models, which are proposed in the literature for clock glitch campaigns on Arm targets. Furthermore, we illustrate *new* fault models that are strongly related to specific features supported by the target devices. This encompasses *Skip with forwarding* for RISC-V target, and *Non-sequential Skip & repeat* For Arm Cortex-M4 target. Other observed faulty behaviors required proposing another fault model: *Combination*. More details in Section IV.

C. Outline

The rest of this paper is organized as follows: Section II presents the followed inference methodology to explain the obtained results. Section III illustrates the experimental setup, then the definitions of the inferred models and the experimental results are reported and discussed in Section IV. The paper is concluded along with perspectives in Section V.

II. FOLLOWED APPROACH

To derive fault models capable of describing the effects of fault injection, we utilized a methodology akin to that of previous works in [3], [11], [12]. The essence of our analysis involves comparing the results of a comprehensive and tailored set of programs executions, encompassing both simulations and physical fault injections, across different levels of digital system abstraction. In this study, we concentrate our analysis on the instruction execution level and the encoding level.

Fig. 1 illustrates the followed approach for the sake of inferring and confirming the use of fault models at encoding and execution levels. Initially, at step 1, physical fault injection campaigns are conducted on a target device running a set of target programs, comprising assembly instructions. Subsequently, in step 2, fault models are deduced from the results of these physical fault injections. The inferred fault models are then applied, as mutants to the encoding level, to simulate the execution of the same target program utilized in step 1 (step 3 in Fig. 1). The last step involves comparing the outcomes of physical fault injection with those of software executions to validate the inferred fault models, along with the ability to explain the obtained faulty behaviors from the physical fault injection (step 4 in Fig. 1). This comparison is conducted based on the output values of the processor's general-purpose registers, each of which has a predetermined initial value, allowing for the detection of any alterations as a result of the fault injection.

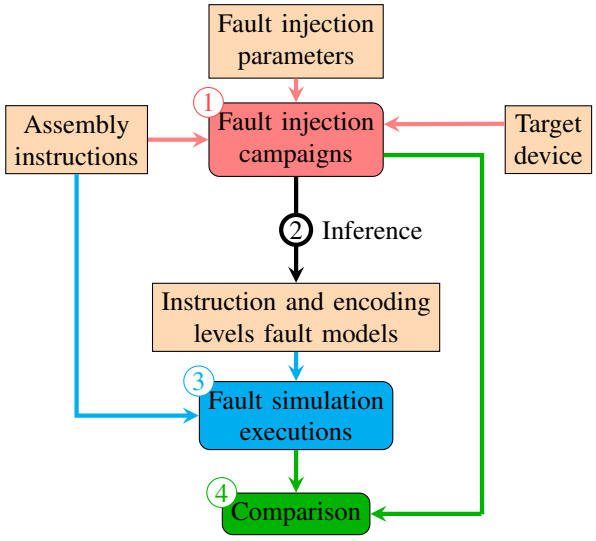

Fig. 1. Followed approach for inferring and confirming fault models

III. EXPERIMENTAL SETUP

Physical fault injection experiments were conducted to examine the effects of fault injection when targeting 32-bit microcontrollers embedding different cores. The objective is to establish reliable and realistic fault models for the observed faulty behaviors for each core. This endeavor aims to enhance the description of faulty behaviors at the ISA level, thereby offering a more comprehensive understanding. Furthermore, it is imperative to assess whether state-of-the-art fault models, typically applied when targeting Arm-based microcontrollers using clock glitch, remain applicable when targeting RISC-V and Arm-based microcontrollers using voltage glitch.

The following subsections present the target devices we have used, the target programs, and the fault injection technique we have employed.

A. Target devices

Two different 32-bit microcontrollers have been employed as target devices. The *first device* is a SiFive 32-bit microcontroller (FE310-G002) that embeds an E31 RISC-V core. The E31 core has a 5-stage pipeline: fetch, decode and register fetch, execute, data memory access, and register writeback. It has 32 general-purpose 32-bit registers, `X0` to `X31`. E31 core is based on RISC-V architecture and supports the RV32IMAC instruction set. Therefore, it supports the standard Multiply (M), Atomic (A), and Compressed (C) RISC-V extensions. Supporting the compressed extension makes RV32IMAC a variable-length instruction set that offers two encoding lengths: 16 and 32 bits. The instruction has a 32-bit encoding if the least significant two bits of a 32-bit word have `0b11` value. Otherwise, the least significant 16 bits belong to a 16-bit instruction [13]. Thanks to the observed faulty behaviors in Section IV, it seems that E31 core supports *operand forwarding*, which is used to resolve data hazards by bypassing data from one pipeline stage to another without writing to and reading from the register file.

This RISC-V device flash memory access size is 32 bits, allowing for the simultaneous retrieval of either one 32-bit instruction or two 16-bit instructions. Furthermore, since the instruction set encoding is variable in length, it permits fetching misaligned instructions in various configurations, as elaborated in [1]. For instance, during a clock cycle, the first half of a 32-bit instruction may be fetched, with the second half retrieved in the subsequent clock cycle.

The *second device* is an STM32L4 microcontroller. It is a 32-bit microcontroller that embeds an Arm Cortex-M4 core. Arm Cortex-M4 has a 3-stage pipeline: fetch, decode, and execute. It has 13 general-purpose 32-bit registers, `R0` to `R12`. Arm Cortex-M4 is based on ARMv7-M architecture and supports the Thumb2 instruction set, consisting of variable-length instructions: 16-bit and 32-bit instructions. The instruction has a 32-bit encoding if the most significant five bits of a 32-bit word have either `0b11101`, `0b11110`, or `0b11111` value. Otherwise, the most significant 16 bits belong to a 16-bit instruction. This Arm Cortex-M4 device supports cache lines of 64 bits. Therefore, in this case, the flash memory access size is 64-bit wide, allowing fetching misaligned instructions in different configurations as detailed in [10].

B. Target programs

Listing 1 shows the RISC-V target program instructions along with their encoding in hexadecimal format. All these instructions are 32-bit instructions. Thus, a complete 32-bit instruction will be fetched at a given clock cycle, and hence, the code in this case is aligned.

Listing 2 presents the Arm Cortex-M4 target program instructions along with their encoding in hexadecimal format. These instructions are the same as the instructions used for RISC-V, but clearly, they have different syntax and encoding as the supported instruction set is Thumb2. Moreover, we added a dummy 16-bit instruction at the beginning of the target program: `MOVS R0, R0`, whose encoding is `0x0000`. This is done to make the code misaligned and to showcase that the provided fault models in this work apply to the observed faulty behaviors regardless of the alignment in memory.

```
1 ADDI x28, x28, 0x3b  // 0x03be0e13
2 ADDI x29, x29, -0xc  // 0xff4e8e93
3 ADDI x7, x29, 0x27   // 0x027e8393
4 OR   x6, x28, x7     // 0x007e6333
5 XORI x6, x6, 0xf     // 0x00f34313
6 ADDI x31, x31, 0xd   // 0x00df8f93
```

Listing 1. RISC-V target program with its encoding in hex. format

```
1 MOVS R0, R0        // 0x0000f103
2 ADD r3, r3, 0x3b   // 0x033bf1a4
3 SUB r4, r4, 0xc    // 0x040cf104
4 ADD r2, r4, 0x27   // 0x0227ea43
5 ORR r1, r3, r2     // 0x0102f081
6 EOR r1, r1, 0xf    // 0x010ff106
7 ADD r6, r6, 0xd    // 0x060d0000
```

Listing 2. Arm Cortex-M4 target program with its encoding in hex. format

These instructions serve as examples in this paper, as the presented fault models hold regardless of the target instructions. Moreover, they are selected for different reasons. First of all, any software application has arithmetic and logical instructions. Additionally, they streamline the process of characterizing the fault injection effects, allowing identifying potential faulty behaviors. Thus, the tractability of a faulty behavior and the applicability of a specific fault model are all achievable with high probability. Upon completion of the regular execution, each register holds a distinct value compared to the others, thereby enhancing the detectability of any faults that may occur.

During the experiments, the processors undergo a predetermined setup before each fault injection, achieved through initialization instructions positioned ahead of the target instructions. Subsequently, following each execution, the data stored in the general-purpose registers is transmitted to a control computer via serial communication for analysis of the outcomes.

C. Voltage glitch fault injection

Introducing variations to the power supply that feeds an embedded system is an effective and low-cost fault injection technique. This method provides acceptable controllability in terms of timing accuracy. However, determining which part of the system is affected by the injection can be challenging.

In this work, ChipWhisperer [14] environment has been employed to perform the voltage glitch fault injection. In this setup, the perturbation to the power supply (Vcc) is performed by underpowering to ground while configuring three parameters as shown in Fig. 2:

- Width: the period of time in which the variation on the power supply is applied.
- Shift: the offset of the starting point of the glitch to the rising edge of the targeted clock cycle.
- Delay: the duration between the rising edge of a trigger signal and the rising edge of the targeted clock cycle.

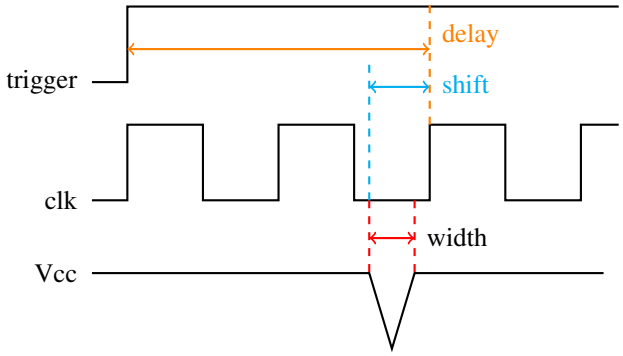

Fig. 2. Voltage glitch parameters

Table II shows the employed values for Shift and Width parameters. The values are expressed as a percentage of one clock period. The negative value for the shift means that the glitch is injected before the rising edge of the targeted clock cycle. Two different delay values are used for each injection campaign. The delay values depend on the number of initialization instructions before the target code and the start of the trigger signal within the code. For each combination of Shift, Width, and Delay, the experiments are repeated 20 times. Thus, the number of executions for each injection campaign is 20 000 (= *10 widths * 50 shifts * 2 delays * 20 repetitions*).

The Width values have been tuned to maximize the number of faults, as it has been observed that the probability to observe faults is much higher in this range. For Shift values, a wide range has been employed to make sure that most of the possible faults can be covered by the proposed models. Two different delay values has been used to target different locations within the target program, also to ensure the applicability of the proposed fault models. In summary, we wanted to ensure that our proposed fault models can cover the obtained faulty behaviours, regardless of the employed parameters. It should be mentioned that, by tuning the parameters, it was possible to maximize the observability of a specific faulty behavior, as we were able to do so.

TABLE II
PARAMETERS SETUP FOR BOTH DEVICES

Parameter	Values
Width	[40,49]
Shift	[-49,0]
repetitions	20
Total number of executions for each device	20 000

IV. EXPERIMENTAL RESULTS AND ANALYSIS

The outcome of a glitch injection in an execution leads to one of the following classes:

- Silent: the execution outcome is equivalent to a normal execution outcome without an injection.
- Crash: a reset or a crash occurs as a result of the injection.
- Fault: the execution outcome is different from the normal execution outcome.

The obtained classifications for both injection campaigns, *i.e.,* when targeting RISC-V device and Arm device are presented in Table III.

TABLE III
PERCENTAGE OF SILENT, CRASH & FAULT OVER THE TWO CAMPAIGNS.

	Target device	
Class	RISC-V	Arm Cortex-M4
Silent	96.71 %	92.405 %
Crash	0.005 %	0.925 %
Fault	3.285 %	6.67 %

The following subsections present the inferred fault models that allowed us to explain the observed faulty behaviours, along with examples for observed experimental results. Table IV shows the percentage of the observed faulty behaviors concerning each fault model over all obtained faulty behaviors for each target device. *Other* means that the observed faulty behavior cannot be modeled by the presented fault models. However, it is shown that *Other* only counts for less than 1 %.

TABLE IV
PERCENTAGE OF THE CLASSIFICATION OF THE OBSERVED FAULTS UNDER THE INFERRED FAULT MODELS FOR EACH TARGET DEVICE.

	Target device	
Fault model	RISC-V	Arm Cortex-M4
Skip	13.85 %	95.2 %
Skip & repeat	36.23 %	0.075 %
Non-sequential skip & repeat	0 %	0.45 %
Partial update	0.3 %	4.2 %
Skip with forwarding	1.98 %	0 %
Combination	47.03 %	0 %
Other	0.61 %	0.075 %

A. *Skip n bits*

A block of size *n bits* of instructions encoding data is skipped and the execution resumes from the next block. This *n* bits is related either to flash memory access size, cache line size, or internal register size. If the code is aligned, then the skipped block refers to complete instructions. However, the skipped block corresponds to different possibilities when the code is misaligned as explained in [10]. *Skip 32 bits* and *Skip 64 bits* are observed for Arm target device, while only *Skip 32 bits* is observed for RISC-V target.

For RISC-V campaign and referring to Listing 1, this model led to a complete instruction skip, as the code is aligned and each line corresponds to a complete 32-bit instruction. Conversely, this model led to different faulty behaviors depending on the target lines in Listing 2.

For Arm target, an observed example of *Skip 64* bits is shown in Listing 3. In this example, the first two lines in Listing 2 are skipped. As a result, the remaining half `0x040c` of `0xf1a4040c` instruction is executed as a *new* 16-bit instruction: `LSLS r4, r1, 0x10`.

```
1 LSLS r4, r1, 0x10   // 0x040c
2 ADD r2, r4, 0x27    // 0x0227ea43
3 ORR r1, r3, r2      // 0x0102f081
4 EOR r1, r1, 0xf     // 0x010ff106
5 ADD r6, r6, 0xd     // 0x060d0000
```

Listing 3. Observed execution as a result of skipping lines 1 and 2 (64 bits) in Listing 2

B. Skip & repeat n bits

A block of size *n bits* of instructions encoding data is skipped and the previous block with the same size has been repeated. As in the previous model, this *n* bits is related either to flash memory access size, cache line size, or internal register size. If the code is aligned, then the skipped and the repeated blocks refer to complete instructions. Yet, they correspond to different possibilities when the code is misaligned as in [10].

In this paper, *Skip & repeat 64 bits* is observed for Arm target, while *Skip & repeat 32 bits* is observed for RISC-V target. This is due to the flash memory access size as mentioned earlier.

For RISC-V campaign and referring to Listing 1, this model led to skipping a complete instruction and repeating the previous instruction.

For Arm campaign, the observed execution depends on the affected part of the code. An observed example is when skipping the encoding at lines 3 and 4 in Listing 2, and repeating the encoding at lines 1 and 2. Listing 4 shows the resulting execution of this example. In this example, `SUB r0, r4,0x0(0xf1a40000)` has been executed instead of `SUB r4, r4, 0xc(0xf1a4040c)`. Additionally, `ADD r3, r3, 0x3b` has been repeated and `ADD r2, r4, 0x2b` has been skipped. Finally, `SUB r1, r4, 0x2(0xf1a40102)` has been executed instead of `ORR r1, r3, r2(0xea430102)`.

```
1 MOVS R0, R0         // 0x0000f103
2 ADD r3, r3, 0x3b    // 0x033bf1a4
3 SUB r0, r4,0x0      // 0x0000f103
4 ADD r3, r3, 0x3b    // 0x033bf1a4
5 SUB r1, r4, 0x2     // 0x0102f081
6 EOR r1, r1, 0xf     // 0x010ff106
7 ADD r6, r6, 0xd     // 0x060d0000
```

Listing 4. Observed execution as a result of skipping the encoding at lines 3 and 4 (64 bits) and repeating the encoding at lines 1 and 2 (64 bits) in Listing 2

C. Non-sequential skip & repeat 32 bits

This model is only observed for the Arm target. Referring to Listing 2, this model is defined as follow: 32 bits at line *i+2* are skipped, while the 32 bits at line *i* are repeated. Such faulty behavior may occur because of the supported cache lines. A cache line of size 64 bits can be seen as two chunks of 32 bits. Thus, each chunk may get its update separately from the other one at a given clock cycle. Consequently, an occurred fault prevents a chunk from being updated, resulting in repeating 32 bits and skipping another 32 bits.

An observed example of this is when skipping line 3 in Listing 2, and repeating line 1. The resulting execution is demonstrated in Listing 5. It is shown how the misalignment has a significant impact in this case. Nevertheless, the fault model works regardless of the alignment.

```
1 MOVS R0, R0         // 0x0000f103
2 ADD r3, r3, 0x3b    // 0x033bf1a4
3 SUB r0, r4, 0x0     // 0x0000f103
4 ADD r2, r3, 0x27    // 0x0227ea43
5 ORR r1, r3, r2      // 0x0102f081
6 EOR r1, r1, 0xf     // 0x010ff106
7 ADD r6, r6, 0xd     // 0x060d0000
```

Listing 5. Observed execution as a result of skipping the encoding at line 3 and repeating the encoding at line 1 in Listing 2

D. Partial update

While the instructions data are propagated through the microcontroller from flash memory to the pipeline stages, the injected glitch may affect some bits to be updated improbably in internal registers. As a result, the faulty bits get their values either from the *previous value*, the *precharge value*, or even from the *next* value. Update from next is only observed for RISC-V target, and in many cases, it is occurred over 16 bits, especially when the code is misaligned. Section IV-F presents an example for this model as a part of *Combination* on RISC-V. For examples on Arm target, we refer the reader to [11].

E. Skip with forwarding

This model is only observed for RISC-V target device. In this case, an instruction is skipped, however, its expected resulting value is forwarded to be used in another instruction. This can be explained as the instruction has finished its execute pipeline-stage, and the value has been forwarded, however, it has not been written back to the register file, thus, the corresponding register keeps its previous value. In [12], the authors noticed, using *only RTL fault simulation* on a RISC-V core, that bit flips could lead to forwarding faults, leading in certain scenarios, to break control flow integrity of programs.

An observed example for this model is when `ADDI x28, x28, 0x3b` instruction at line 1 in Listing 1 has been skipped, but the correct value of **`x28`** has been forwarded to be used in `OR x6, x28, x7`. As a result, **`x6`** had its golden value, however, **`x28`** kept its initial value.

It has been noticed that a value can be forwarded from an instruction at line *i* to dependent instructions at lines *i+1*, *i+2*, or *i+3*, but not further. This can be explained as the value can be forwarded, at max, from writeback pipeline-stage (5th stage) to decode and register fetch stage (2nd stage).

F. Combination

In this case, the observed faulty behavior is modeled by a combination of more than one fault model from the aforementioned fault models. This *Combination* fault model has been observed only for RISC-V target device. This might be explained due to the higher number of pipeline stages (5 stages) compared to Arm device (3 stages).

An observed example of this behavior is depicted in Listing 6. In this example, *Partial update* from previous value has occurred at line 3, so that **x29** got the result of: x29+0x27, instead of **x7**. However, due to forwarding, the correct value of **x7** has been correctly used for OR at line 4, but finally **x7** kept its initial value. Furthermore, XORI has been skipped.

```
1 ADDI x28, x28, 0x3b   // 0x03be0e13
2 ADDI x29, x29, -0xc   // 0xff4e8e93
3 ADDI x29, x29, 0x27   // 0x027e8e93
4 OR   x6, x28, x7      // 0x007e6333
5 XORI x6, x6, 0xf
6 ADDI x31, x31, 0xd    // 0x00df8f93
```

Listing 6. Observed execution example on RISC-V target for *Combination* fault model (*Partial update, forwarding, Skip*)

It might be noticed that predicting faulty behaviors, based on *Combination* and *Partial update* models, could be difficult. However, we have observed repetitive and reproducible patterns that show that some behaviors are more probable than others. This might be related to *Bit sensitivities* mentioned in [11]. Further investigation is required to confirm this.

V. Conclusion and Future Work

In conclusion, voltage glitch campaigns have been performed on two different devices, embedding RISC-V and Arm Cortex-M4 cores. In order to explain the obtained faulty behaviors, different fault models have been inferred. These models allowed explaining more than **99%** of the obtained faults. Some of these models are applicable to both target devices: *Skip*, *Skip & repeat*, and *Partial update* fault models. These three models are also applicable to clock glitch results, as shown in the literature. Moreover, additional faulty behaviors have occurred due to some features. These features encompass variable-length encoding, cache lines, and forwarding. Therefore, new device features could lead to new possible exploitation. To deal with this, other fault models have been proposed: *Non-sequential skip & repeat*, *Skip with forwarding*, and *Combination*. The presented fault models enable better understanding of the fault effects. Thus, easing the process of vulnerability analyses, and hence, simplifying the thinking of cost-effective countermeasures.

As future work perspectives, countermeasure design at software and/or hardware levels would be very important. Also, evaluating real-life security applications using the proposed fault models would be captivating. Finally, another interesting perspective is a deep investigation of the observed faulty behaviors at hardware level. This will help in determining the vulnerable registers, and thus, easing the design of protections.

Acknowledgment

This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissements d'avenir, and by ARSENE project (PEPR PP7 ARSENE — ANR-22-PECY-0004).

References

[1] I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle, and P. Maistri, "Variable-length instruction set: Feature or bug?" in *25th Euromicro Conference on Digital System Design*. Maspalomas, Spain: IEEE, Aug. 2022, pp. 464–471.

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

[3] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*. IEEE Computer Society, 2013, pp. 77–88.

[4] V. Khuat, J.-L. Danger, and 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)*, 2021, pp. 74–85.

[5] O. Trabelsi, L. Sauvage, and J.-L. Danger, "Characterization of electromagnetic fault injection on a 32-bit microcontroller instruction buffer," in *2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2020, pp. 1–6.

[6] J. Proy, K. Heydemann, A. Berzati, F. Majéric, and A. Cohen, "A first ISA-level characterization of EM pulse effects on superscalar microarchitectures: A secure software perspective," in *Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019*. ACM, 2019, pp. 7:1–7:10.

[7] T. Trouchkine, G. Bouffard, and J. Clédière, "EM fault model characterization on socs: From different architectures to the same fault model," in *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*. IEEE, 2021, pp. 31–38.

[8] V. Werner, L. Maingault, and M. Potet, "An end-to-end approach for multi-fault attack vulnerability assessment," in *Workshop on Fault Detection and Tolerance in Cryptography*. Milan, Italy: IEEE, 2020, pp. 10–17.

[9] M. A. Elmohr, H. Liao, and C. H. Gebotys, "EM fault injection on ARM and RISC-V," in *2020 21st International Symposium on Quality Electronic Design (ISQED)*, 2020, pp. 206–212.

[10] I. Alshaer, G. Burghoorn, B. Colombier, C. Deleuze, V. Beroulle, and P. Maistri, "Cross-layer analysis of clock glitch fault injection while fetching variable-length instructions," *Journal of Cryptographic Engineering*, pp. 1–18, 2024.

[11] I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle, and P. Maistri, "Microarchitectural insights into unexplained behaviors under clock glitch fault injection," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science. Springer Nature Switzerland, 2024, pp. 3–22.

[12] J. Laurent, C. Deleuze, F. Pebay-Peyroula, and V. Beroulle, "Bridging the gap between RTL and software fault injection," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 17, no. 3, pp. 38:1–38:24, 2021.

[13] S. Inc. and B. EECS Department, University of California, "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA," https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf, [Accessed: February 16, 2024].

[14] C. O'Flynn and Z. D. Chen, "Chipwhisperer: An open-source platform for hardware embedded security research," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*, ser. Lecture Notes in Computer Science, E. Prouff, Ed., vol. 8622. Paris, France: Springer, 2014, pp. 243–260.