Variable-Length Instruction Set: Feature or Bug?

Ihab Alshaer*†, Brice Colombier†, Christophe Deleuze*, Vincent Beroulle*, Paolo Maistri†
*Univ. Grenoble Alpes, Grenoble INP[1], LCIS, 26000 Valence, France
†Univ. Grenoble Alpes, CNRS, Grenoble INP[1], TIMA, 38000 Grenoble, France
{first.last}@univ-grenoble-alpes.fr

***Abstract*—With the increasing complexity of digital applications, the use of variable-length instruction sets became essential, in order to achieve higher code density and thus better performance. However, security aspects must always be considered, in particular with the significant improvement of attack techniques and equipment. Fault injection, in particular, is among the most interesting and promising attack techniques thanks to the recent advancements. In this article, we provide proper characterization, at the instruction set architecture (ISA) level, for several faulty behaviors that can be obtained when targeting a variable-length instruction set. We take into account the binary encoding of instructions, and show how the obtained behaviors depend on the alignment of the instructions in the memory. Moreover, we are also able to give a better insight on previous results from the literature, that were still partially unexplained. We also show how the observed behaviors can be exploited in various security contexts.**

***Index Terms*—variable-length instruction set, fault injection.**

I. Introduction

Digital systems complexity, including their running applications, is continuously increasing. This opens the doors to two considerations: on the one hand, the need for high performance and new methods to deal with such advances; on the other hand, new vulnerabilities could appear at different levels of a digital system and can be exploited by attackers.

Digital systems may contain sensitive information that can be effectively protected through cryptographic algorithms, usually implemented in software on an embedded microprocessor. Such implementations, however, might be vulnerable to attacks that aim at extracting this sensitive information, in particular when these attacks do not require years of computations to break an algorithm, and here, physical attacks can take place.

Fault injection is an effective physical attack, belonging to the family of active attacks. In this setting, the attacker has physical access to the digital device or its surrounding environment, and will try to change the normal behavior of the device by injecting one or more faults, then observing the erroneous behavior. The resulting fault or faults may lead to an interesting behavior that could be further exploited as a vulnerability.

To perform the fault injection, a physical interference needs to be applied to the digital device. The interference can be in a form of radiations [1], laser beams [2], [3], electromagnetic (EM) pulses [4], [5], variations in the power supply [6], perturbations to the clock signal [7], [8], or changing the environmental conditions such as the temperature [9], just to cite the most widespread techniques.

Several research studies have been conducted to characterize the faults at the instruction set architecture (ISA) level to propose usable fault models, which are the abstract representations of the underlying physical phenomena: in order to build efficient countermeasures against fault attacks, realistic fault models are required. The choice of ISA level is due to the fact that this can be considered as the focal point for bringing high (software) and low (hardware) levels of abstraction together. The majority of the proposed fault models, in fact, describe the effects on the instruction itself: for this reason, *instruction skip* and *instruction corruption* are the most used models. In particular, the instruction skip (which can also be described as a replacement with one or more `NOP`s) can affect one [5], [7], [8], two [8] or multiple instructions [3], [4], [10]. It is important to highlight, however, that in all of the previous works, the authors always refer to the skip model only for complete instructions, either one or more. In [3] and [4], for instance, the authors refer to the multiple skip as a number of bits or number of bytes, which is related to the size of the instruction cache, but in any case, it was always skipping an integer number of instructions. With respect to the instruction corruption, this may be related to the opcode [2], or the operands [2], [5], [6], either destination or source operands. All these works described the instruction corruption with respect to the targeted instruction itself, without taking into consideration any additional constraints. And most importantly, several observed effects still remained unexplained.

In [8], the authors provided fault effect characterization at ISA level after performing clock glitch fault injection campaigns on ARM Cortex-M4 processor. Some of their observed faulty behaviors could not be explained, such as the corrupted values that are found in some registers, independently of their actual use in the code. Also, the reason of having single or double skip was not clear. Similarly, EM fault injection campaigns on ARM Cortex-A9 processor have been carried out in [5] to provide characterization at ISA level as well: here, the authors explicitly stated that some of the obtained faults remained unexplained. In addition, [11] described EM fault campaigns on two modern processors: ARM BCM2837, which embeds Cortex-A53, and Intel Core i3-6100T CPU. The authors also provided characterization at ISA level to propose general fault models for different architectures: one of their proposed models is random register corruption; moreover, some of their faults were still left unexplained, with unknown

[1]Institute of Engineering Univ. Grenoble Alpes

fault model. Finally, in [6], [12], [13], the authors showed that modifying the program counter, as a result of fault injection attacks, is code dependent in terms of instructions, registers and/or immediate values, without further explanation.

We believe that this work can explain the rationale behind several of the inferred fault models in previous works, and also help in explaining most of the unexplained faulty behaviors. To our knowledge, there is no research that takes into consideration whether the targeted ISA supports variable-length instructions or not, nor uses this knowledge to explain the obtained results. In particular, they never consider whether the instruction bits fetched from the memory are corresponding to complete instructions or not, as the fetch size is always fixed while the ISA may support variable-length instructions. How such information can be exploited in a security application, or taken into account when designing countermeasures has never been discussed either.

In this article, we present two new inferred fault models: *skip 32 bits*, and *skip & repeat 32 bits* that are applied on the *encoding* of the instructions. These two models allow us to explain a wide range of the obtained faulty behaviors at the ISA level, regardless of the targeted instructions. This allows providing proper characterization for the effects of the observed faulty behaviors. Also, we show how the faulty behaviors will differ depending on the alignment of the code in memory. In other words, the difference relies on the fact of whether the fetched 32 bits correspond to the same instruction or not. Finally, we provide various examples to violate a predefined security property in a specific program by exploiting the obtained results.

The rest of the article is organized as follows: Section II provides a background on the variable-length instruction sets. Section III describes the experimental setup. The results and the analysis are presented in Section IV. Section V provides an exploitation example for one of the observed faulty behaviors. The article is concluded along with the perspectives in Section VI.

II. Variable-length Instruction Sets

Reducing code size is one of the earliest applied methods to reduce power consumption and memory space, and with them the overall cost of a digital system, which is highly affected by program code and the fetch stage in the pipeline [14]. Code size reduction, which aims at reaching the highest possible code density and hence better performance, can be achieved by using a variable-length instruction set [14], [15], [16], [17]. Additionally, less power is consumed due to the smaller number of fetches [15].

A variable-length instruction set can be described as a combination of two sets of instructions: short instructions with regard to their encoding, nonetheless providing the same functionality as their corresponding instructions, which have larger encoding. This set can also be called compressed set. The second set consists of instructions that have larger encoding and cannot be compressed while providing the same functionality. High code density can be achieved by the compressed instructions, while the second set allows preserving the high performance. An example of the effect of ISA on cost and performance is the instruction cache: shorter encoding needs smaller caches for the same performance [17], [18]. Therefore, having shorter encoding will have fewer cache misses (with a given size), thus increasing the overall throughput of the processor. On the other hand, dealing with different versions of encoding will increase the complexity of the instruction decoder [18].

The x86 [19] ISA, which is supported by Intel and AMD processors, offers various lengths of encoding between 1 and 15 bytes. Another example of a variable-length instruction set is microMIPS [16], which provides a set of 16-bit instructions that correspond to the commonly used ones, in addition to all of the instructions from the MIPS32/64 ISAs [20], [21]. For some benchmarks, microMIPS has 35% smaller code size and almost similar performance compared to MIPS32 [16]. In 2015, a draft proposal [17] has been published to provide 16-bit encodings for some instructions in RISC-V ISA [22], this new instruction set is known as RISC-V Compressed (RVC), which delivers more than 25% code size reduction [17]. Finally, most ARM processors, such as Cortex-M3 and Cortex-A9, support their dedicated variable-length instruction set as well, known as Thumb2 instruction set [23]. It consists of two sets of instructions: 16-bit and 32-bit. Thumb2 delivers 30% of code size reduction on average [15]. In this article, we chose a Cortex-M3 as target processor, and thus Thumb2 as target instruction set, but we consider that the results can be generalized to other variable-length instruction sets.

III. Experimental Setup

Physical fault injection experiments have been performed in order to investigate the fault injection effects on a variable-length instruction set. This aims at providing better characterization and description, at the ISA level, for the wide range of faulty behaviors that might be obtained when performing fault injection campaigns.

The following subsections present the fault injection technique we have used, the target device, and the target programs. The last subsection describes the injection parameters and the classification categories for the experimental outcomes.

A. Clock glitch fault injection

Applying perturbations to the main clock signal that is fed to the processor is a non-invasive and an effective fault injection technique. Clock glitch is considered as a low-cost fault injection technique compared to other techniques like laser and EM pulses. Also, it can provide an acceptable controllablility with respect to the temporal accuracy, and hence the location of the injection in the target program. However, since the glitch is injected in the global clock, there is no particular knowledge about which architectural element could be affected as a result of the injection.

When performing clock glitch fault injection, a glitch is injected just before or after the rising edge of the clock. This

glitch would appear as a new clock cycle for the microprocessor, disrupting the regular behavior of the clock signal. Thus, a timing violation will possibly occur, leading to various kinds of faulty behaviors.

When performing a clock glitch, three parameters must be tuned as shown in Fig. 1:

- Delay: the time between the rising edge of the trigger signal used for synchronization and the rising edge of the targeted clock cycle.
- Shift: the time between the rising edge of the glitch and the rising edge of the targeted clock cycle.
- Width: the duration of the glitch itself.

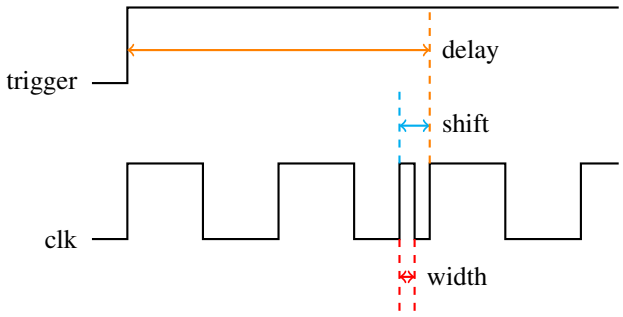


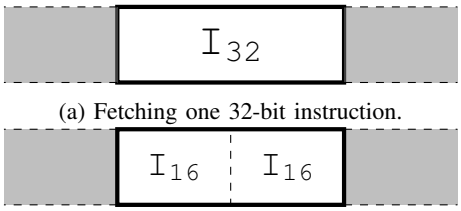
Fig. 1: Clock glitch parameters [8].

B. Target device

The boards that are used for the experiments are the ChipWhisperer [24] boards: ChipWhisperer-Lite, CW308 UFO baseboard and the target board. The Lite board is the control board that is connected to the control laptop through a USB cable. The CW308 allows the portability and the usability of different targets, it provides a host for the microcontroller target board and it is connected to the Lite board while performing an experiment. The target board is a 32-bit microcontroller, which embeds an ARM Cortex-M3 processor. These ChipWhisperer boards include a dedicated environment for side channel analysis, voltage and clock glitch generation. We will leverage the clock glitch capabilities of this setup in the experiments.

Cortex-M3 includes a pipeline with three stages: fetch, decode and execute. Cortex-M3 is based on the ARMv7-M [25] architecture and supports the Thumb2 instruction set, which consists of variable-length instructions as mentioned in the previous section: 16-bit and 32-bit instructions. In this microcontroller, the fetch size from the memory through the AHB (Advanced High performance Bus) is fixed and equal to 32 bits, regardless of the instruction size. Hence, as a result of having variable-length instructions, the fetched 32 bits can belong to one of the cases in Fig. 2 or Fig. 3. Fig. 2 represents the fetching cases when the code is aligned in the memory, while Fig. 3 represents the misaligned cases. In Section IV, we will see how these different possibilities affect the observed execution as a result of the fault injection campaigns.

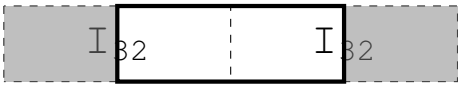
The processor detects whether the instruction that is going to be executed is 16-bit or 32-bit by analyzing the most



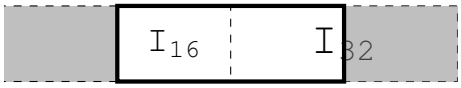
(a) Fetching one 32-bit instruction.

(b) Fetching two 16-bit instructions.

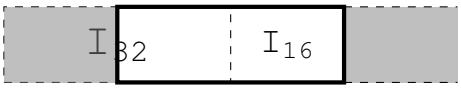
Fig. 2: Fetching aligned instructions.



(a) Fetching the bottom half of a 32-bit instruction and the top half of another 32-bit instruction.



(b) Fetching one 16-bit instruction and the top half of a 32-bit instruction.



(c) Fetching the bottom half of a 32-bit instruction and one 16-bit instruction.

Fig. 3: Fetching misaligned instructions.

significant five bits of the half-word that will arrive first to the processor itself [25]. If these five bits are one of the following values, then the word contains a 32-bit instruction, otherwise, the arrived half-word is a 16-bit instruction:

- `0b11101`,
- `0b11110`,
- `0b11111`.

This knowledge is very important to explain the observed faulty behaviors, as detailed in Section IV. A note to take into consideration for the following sections is that we use the big-endian representation for the binary encoding of the instructions to ease the readability. Thus, for a 32-bit instruction, the most significant 16 bits arrive first in the pipeline, where they are checked for 16- or 32-bit instruction as described above.

C. Target programs

The injection is performed into inline assembly instructions within a C program. To ease the process of the injection, the program is divided into three parts as follows:

- Prologue: initialization instructions to put the processor in a known state before starting the fault injection.
- Target: instructions targeted by the fault injection as well as extra instructions that would allow observing any propagation effect.
- Epilogue: reading the registers' state, in particular the general purpose registers R0 to R12; the values are then transferred to the control laptop through serial communication.

Two series of NOP instructions are used to isolate the three parts from each other. The registers used in the target part are initialized in the prologue.

In the injection campaigns, we used specific instructions in the target part as shown in Listing 1 and Listing 2. Listing 1 shows an example of an aligned code. Hence, every fetched 32 bits are either two full 16-bit instructions or one 32-bit instruction as shown in Fig. 2. The first two instructions (MOV and LSLS) are 16-bit instructions; all the ADD instructions are 32-bit instructions. Such instructions are chosen to simplify the characterization of the fault injection effects. At the end of the normal execution, each register has a different value from the others, which increases the observability of occurring faults. Small and different immediate values are used with the ADD instructions, in order to see if an immediate value is replaced with a register number or vice-versa (*e.g.,* R3 becomes 0x3), as explained in Section IV.

Listing 2 shows an example of a misaligned code. It is exactly the same as Listing 1, except that the first instruction (*i.e.*, MOV instruction) is removed. Therefore, the code is misaligned, and hence, when 32 bits are fetched, they now belong to either two different 32-bit instructions or one 16-bit instruction and the other 16 bits belong to a 32-bit instruction as shown in Fig. 3. In Section IV, we will show how such a small modification of the target code greatly affects the observed behaviors at the ISA level.

The prologue is controlled to be always aligned in the code memory space. Hence, it has no influence on the code alignment, which only depends on the instructions of the target part.

```
1 MOV R8, R4   // R8 = R4
2 LSLS R2, R0, 0x10 // R2 = R0 << 0x10
3 ADD R1, R1, 0x6 // R1 = R1 + 0x6
4 ADD R3, R3, 0xa // R3 = R3 + 0xa
5 ADD R4, R4, 0xb // R4 = R4 + 0xb
6 ADD R5, R6, R3  // R5 = R6 + R3
7 ADD R3, R3, 0xf // R3 = R3 + 0xf
```

Listing 1: Target part in the aligned code.

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R3, R3, 0xa
4 ADD R4, R4, 0xb
5 ADD R5, R6, R3
6 ADD R3, R3, 0xf
```

Listing 2: Target part in the misaligned code.

D. Injection parameters and classification

Each injection campaign consists in repeating the clock glitch fault injection several times (1000-10 000 times) for the same shift and width values. The Delay value is based on the targeted instructions. A single glitch is injected during each program execution.

Three outcomes can occur as a result of the fault injection as follows:

- Crash: the injection causes a crash, a reset, or a failure when reading the final state in the epilogue.
- Silent: the outcome of the injection is identical to the normal state, *i.e.*, without any injection.
- Fault: a successful fault has occurred and can be observed as a result of the clock glitch injection.

In the next section, we focus on a subset of the obtained faults, discarding the crashes and silent cases. This subset represents the largest subset of the obtained faulty behaviors and our focus is on the occurrence of these behaviors, not their probability of occurrence. However, it is worth mentioning that these behaviors are highly and easily reproducible.

IV. Experimental Results and Analysis

Different faulty behaviors were observed after performing the clock glitch fault injection campaigns. In the following subsections, we only focus on the faulty behaviors that are related to two specific inferred fault models that are applied on the *encoding* of the instructions: skip 32 bits or skip & repeat 32 bits. These are strongly related to the fetch stage of the processor pipeline. These faulty behaviors, as mentioned earlier, represent the largest subset of the observed behaviors. We show how the outcomes of the injection campaigns depend on the code alignment in memory, whether it is aligned or misaligned, although the inferred fault models at the encoding level (*i.e.*, skip or skip & repeat 32 bits) are always the same in both cases. The last subsection provides further details about the last behavior, where we were able to execute a new instruction as a result of the clock glitch.

A. Aligned code scenario

Listing 3 represents the binary encoding of the target program in Listing 1. Each line corresponds to one 32-bit instruction, except line 1, which corresponds to the first two 16-bit instructions. Therefore, this code is *aligned* in the memory.

```
1 46a00402 // MOV R8, R4 // LSLS R2, R0, 0x10
2 f1010106 // ADD R1, R1, 0x6
3 f103030a // ADD R3, R3, 0xa
4 f104040b // ADD R4, R4, 0xb
5 eb060503 // ADD R5, R6, R3
6 f103030f // ADD R3, R3, 0xf
```

Listing 3: Binary encoding for the aligned code in Hex.

Assuming *i* is the line number that points to a 32-bit block of the encoded target program, then the "skip 32 bits" and "skip & repeat 32 bits" fault models are described as follows:

- Skip 32 bits: the 32 bits at line *i* are skipped, and the execution resumes from line *i+1*.
- Skip & repeat 32 bits: the 32 bits at line *i+1* are skipped and the 32 bits at line *i* are repeated.

The following subsections show three observed faulty behaviors of skip and skip & repeat after performing the fault

injection campaigns. Although we focus here on a specific instruction for our discussion, what is applied on these examples can also be applied to other locations of the target program without loss of generality. For each subsection, two descriptions are hence provided: at the binary encoding level and at the ISA level.

1) Skip & repeat 32 bits / single instruction skip & single instruction repeat: skipping the `ADD R3, R3, 0xa` instruction at line 3 in Listing 3 and repeating the `ADD R1,R1, 0x6` instruction at line 2 is an example of this model. The observed execution at the ISA level is shown in Listing 4.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R1, R1, 0x6
5 ADD R4, R4, 0xb
6 ADD R5, R6, R3
7 ADD R3, R3, 0xf
```

Listing 4: Observed execution for skip & repeat 32 bits / single instruction skip & single instruction repeat.

2) Skip 32 bits / single instruction skip: skipping any line in Listing 3, except line 1, led to a single instruction skip, as each line of these represents a complete 32-bit instruction. For example, when skipping the `ADD R4, R4, 0xb` instruction at line 4 in Listing 3, the observed execution at the ISA level is shown in Listing 5.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R3, R3, 0xa
5 ADD R5, R6, R3
6 ADD R3, R3, 0xf
```

Listing 5: Observed execution for skip 32 bits / single instruction skip.

3) Skip 32 bits / double instruction skip: skipping the first line in Listing 3, led to a double instruction skip, as this line represents two blocks of 16-bit instructions. Listing 6 shows the observed execution at ISA level for this example.

```
1 ADD R1, R1, 0x6
2 ADD R3, R3, 0xa
3 ADD R4, R4, 0xb
4 ADD R5, R6, R3
5 ADD R3, R3, 0xf
```

Listing 6: Observed execution for skip 32 bits / double instruction skip.

B. Misaligned code scenario

We now consider the case of a *misaligned* code. We achieve this by removing the `MOV` instruction, which is a 16-bit instruction, from the target part in the program as shown in Listing 2. Listing 7 shows the binary encoding of the misaligned code. Just as before, each line contains 32 bits, but unlike previous case each line does not correspond now to a single 32-bit instruction. For sake of clarity, we highlighted each instruction by a different color: the reader can clearly see that each 32-bit instruction is split over two consecutive lines.

Again, in terms of 32-bit faults, similar outcomes were obtained. However, as the line now consists of two 16 bits that belong to two different instructions (as previously shown in Fig. 3), different faulty behaviors were observed at the ISA level. The actual faulty behaviors that can be obtained depend on the target location of the glitch injection; hence, the following subsections provide two observed examples for each model, *i.e.*, two examples of skip & repeat 32 bits, and two examples of skip 32 bits. It is interesting to note that the observed behaviors in this scenario were more complex than the previous one: among several outcomes, for example, we were able to observe double instruction corruption, and new instruction execution.

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00 // bf00: NOP.
```

Listing 7: Binary encoding for the misaligned code in Hex.

1) Skip & repeat 32 bits / double instruction corruption example 1: this example presents the case when skipping and repeating 32 bits refers to the configuration shown in Fig. 3a. An example of this case is when skipping line 4 and repeating line 3 in Listing 7.

As a result, two instructions are corrupted as shown in Listing 8. `0xf104` means that the instruction to be executed is a 32-bit instruction as the most significant five bits are `0b11110`. The repeated red part (`0x030a`) is going to be part of the new executed instruction. In addition, since `0xf104` is repeated, then another new 32-bit instruction is executed. Its first half is from the green and the second half is from the 16 bits that remained from the orange instruction at line 5 in Listing 7 (`0x0503`).

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R3, R3, 0xa
4 ADD R3, R4, 0xa  // f104030a
5 ADD R5, R4, 0x3  // f1040503
6 ADD R3, R3, 0xf
```

Listing 8: Observed execution for skip & repeat 32 bits / double instruction corruption example 1.

To describe the observed behaviors for this example at ISA level and generalizing the obtained faults to other target programs that could have similar structure, we explain the corruption of two 32-bit instructions as follows:

- For the `ADD R4, R4, 0xb` instruction: the destination operand and the second source operand are replaced by operands from the *previous* instruction.
- For the `ADD R5, R6, R3` instruction: the first source operand is replaced by the first source operand from the *previous* instruction. Its opcode (`ADD with register`) is also replaced by the *previous* opcode (`ADD with immediate`). Therefore, the register number R3 is now considered as an immediate value: `0x3`.

2) Skip & repeat 32 bits / double instruction corruption example 2: this example presents the case when skipping 32 bits refers to the configuration shown in Fig. 3a, and repeating 32 bits refers to Fig. 3b. This is the case when skipping line 2 and repeating line 1 in Listing 7.

In this case, the repeated 16 bits: `0x0402`, which is originally 16-bit instruction, is going to be part of a 32-bit instruction since `0xf101` requires a 32-bit instruction to be executed (the most significant five bits are `0b11110`, as discussed in Section III-B). Thus, the `ADD R1, R1, 0x6` instruction is corrupted. The `ADD R3, R3, 0xa` instruction is corrupted as well, as half of it is skipped. Listing 9 shows the observed execution for this example at ISA level.

```
1 LSLS R2, R0, 0x10
2 ADD R4, R1, 0x2  // f1010402
3 ADD R3, R1, 0xa  // f101030a
4 ADD R4, R4, 0xb
5 ADD R5, R6, R3
6 ADD R3, R3, 0xf
```

Listing 9: Observed execution for skip & repeat 32 bits / double instruction corruption example 2.

To describe the observed behaviors for this example at ISA level and generalize the obtained faults to other target programs that could have similar structure, we explain the corruption of two 32-bit instructions as follows:

- For the `ADD R1, R1, 0x6` instruction: two operands were replaced. The two new operands are coming from the *previous* 16-bit instruction encoding (not its operands), as this encoding allows having a valid 32-bit instruction, *i.e.,* valid register number and valid immediate value.
- For the `ADD R3, R3, 0xa` instruction: the first source operand is replaced by the first source operand from the *previous* instruction. Additionally, the opcode is also replaced by the *previous* opcode. However, as in this case the previous opcode is similar, it does not lead to execute a different instruction.

3) Skip 32 bits / single instruction skip and single instruction corruption: this example presents the case when skipping 32 bits refers to Fig. 3a. For instance, it is the case when skipping line 3 in Listing 7. The observed execution at ISA is shown in Listing 10. The `ADD R4, R4, 0xb` instruction is skipped and the `ADD R3, R3, 0xa` instruction is corrupted by replacing two of its operands by operands from the skipped instruction.

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R4, R3, 0xb  // f103040b
4 ADD R5, R6, R3
5 ADD R3, R3, 0xf
```

Listing 10: Observed execution for skip 32 bits / single instruction skip and single instruction corruption.

4) Skip 32 bits / double instruction skip and new instruction execution: this example presents the case when skipping 32 bits refers to Fig. 3b. It occurs, for example, when skipping line 1 in Listing 7.

As a result of skipping line 1, `0x0106` arrives first to the core, and since the most significant five bits of `0x0106` are `0b00000`, a 16-bit instruction is going to be executed, which has the encoding of `0x0106`. This instruction is `LSLS R6, R0, 0x4`. The other instructions in the target program will not be affected and they are going to be executed normally. Listing 11 shows the observed execution of this example at ISA level. The first instruction is colored blue since its encoding came from the original blue instruction. It is shown that two instructions were skipped (one 16-bit instruction and one 32-bit instruction) and a *new* 16-bit instruction was executed instead.

```
1 LSLS R6, R0, 0x4 // 0106
2 ADD R3, R3, 0xa
3 ADD R4, R4, 0xb
4 ADD R5, R6, R3
5 ADD R3, R3, 0xf
```

Listing 11: Observed execution for skip 32 bits / double instruction skip and new instruction execution.

To prove that the new `LSLS` instruction is not related to the original `LSLS` instruction, the following subsection provides a variety of instructions that can be executed as a result of this observed behavior.

C. More on the last observed behavior

Since the encoding of the new logical shift left instruction in the last skip example is coming from the destination register and the second source operand in the `ADD` instruction, then changing these two operands to other values allows "generating" new instructions.

Table I shows examples of new instructions when changing these two operands. All the examples in Table I were validated experimentally by clock glitch fault injection campaigns. In other words, replacing the blue instruction in Listing 7 by an instruction in the first column of Table I, allows observing the execution of the corresponding instruction from the third column of Table I, when performing clock glitch fault injection campaigns. Column 2 shows the encoding of the new instruction, which comes from the least 16 bits of the original 32-bit instruction.

By generating all 65 536 possible 16-bit combinations and disassembling them to check if they are a valid 16-bit Thumb2

TABLE I: Effect of last observed behavior with different destination register and/or immediate value.

Original instruction	Least-significant 16 bits	New instruction
`ADD R4, R1, 0x9`	`0x0409`	`LSLS R1, R1, 0x10`
`ADD R0, R1, 0x46c`	`0x406c`	`EORS R4, R5`
`ADD R12, R1, 0x60c`	`0x6c0c`	`LDR R4, [R1, 0x40]`
`ADD R0, R1, 0x161`	`0x1061`	`ASRS R1, R4, 0x1`
`ADD R0, R1, 0x205`	`0x2005`	`MOV R0, 0x5`
`ADD R3, R1, 0x416`	`0x4316`	`ORRS R6, R2`

instruction or not, we identified more than 58 000 valid 16-bit instructions. Each of these instructions can be executed as a result of this specific faulty behavior, regardless of the opcode of the original 32-bit instruction in the target program.

The consequences are particularly interesting. Trouchkine *et al.* [11] observed R8 and R0 corruption when targeting a series of `AND R8, R8, R8`. They said the corruption is sometimes a complete reset of the register. This `AND` instruction has the encoding: `0xea080808`. Thanks to our analysis, we can fully explain the corruption they observe on the Cortex-A53 processor, which supports the Thumb2 instruction set. The fault injection leads to the creation and execution of the 16-bit instruction `0x0808`, which is the encoding of `LSRS R0, R1, 0x20`. This operation leads to a reset of R0, since the value in R1 is shifted right by 32 bits and its result (obviously 0) is stored in R0.

In Table I, it is shown that many instructions can lead to violate different security properties. For example, executing an `LDR` (LOAD) instruction could lead to reveal some values in the memory, breaking the confidentiality property. As another example, executing the `EORS` (XOR) instruction could allow an attacker to observe a collision in a cryptographic algorithm, which could lead to recover secret data. Finally, moving an immediate value to a register could lead to corrupt a loop counter value if this register is used for the counter itself. In the next section, we focus on a small subset of the instructions that can be executed, allowing an attacker to control the program flow by performing clock glitch fault injection.

V. Exploitation

To exploit the previous results, we leverage the ability of executing a new 16-bit instruction to control the program counter, where those 16 bits are originally belonging to a 32-bit instruction as shown above. We consider this specific example, because controlling the program counter as mentioned in [6], [12], could lead to harmful attacks, like privilege escalation or secure boot violation. Thus, our provided example works as a proof of concept for potential exploitation of the presented results.

In the following, we assume that R8 contains an address that points to a critical part of the code, which can only be executed in a secure mode. Hence, our security property is to not modify the program counter to that critical address. This security property is violated if we manage to execute the `MOV PC, R8` instruction for example. The 16-bit encoding of this instruction is `0x46c7`.

Several examples lead to violate this security property. Table II summarises some of the instructions allowing that. Having one of these instructions in a misaligned code allows controlling the program counter to the value of `R8` when applying a fault injection attack. To validate this theoretical analysis, we performed clock glitch fault injection campaigns on the code in Listing 12.

A specific address is stored in R8 in the prologue. This address refers to line 12 in Listing 12. The instruction at line 6 can be any instruction from Table II and we were able to validate all of them. In other words, we were always able to jump to line 12 in Listing 12 when performing the attack on any of the instructions in Table II.

Other instructions may be a source for such vulnerability, especially if other source registers store critical addresses, and not only `R8` as in our example. However, most of the original instructions that could be a source of such vulnerability, have `R6` as a destination register, as already shown in Table II. Gratchof *et al.* [13], said that there is a higher probability to change the value of the program counter when the destination register of a `MOV` instruction is `R6`. They observed varied jumps in a program when performing fault injection attacks on a Cortex-A9, which supports two execution states: Thumb2 and ARM32. Hence, our observations can reasonably explain and prove their results regarding the R6 register, assuming that their execution state was Thumb2.

```
1  //prologue
2  MOVW R8, 0x056e // storing the critical-
3  MOVT R8, 0x0800 // -address in R8
4  //series of NOPs
5  LSLS R2,R0,0x10
6  //any instruction from Table II
7  ADD R3, R3, 0xa
8  ADD R4, R4, 0xb
9  ADD R5, R6, R3
10 ADD R3, R3, 0xf
11 //series of NOPs
12 LDR R1,[R1,0xf00]
13 MOV R9,R6
14 //epilogue
```

Listing 12: Target program for exploitation example.

TABLE II: Instructions that lead to modify the PC to the value in R8 when performing clock glitch fault injection.

Original instruction	Least-significant 16 bits
`ADD R6, R1, 0x4c7`	`0x46c7`
`SUB R6, R1, 0x4c7`	`0x46c7`
`MOVW R6, 0x4c7`	`0x46c7`
`LDR R4, [r0, 0x6c7]`	`0x46c7`
`ORR R6, R6, 0x63800000`	`0x46c7`

VI. Conclusion and Perspectives

In this article, we show how the observed faulty behaviors at ISA level can dramatically change depending on the code alignment in memory. This comes from the fact that Thumb2 instruction set supports variable-length instructions, which can lead to aligned or misaligned code in memory. We also show that all these behaviors can be explained at the binary encoding level with the same two fault models: skip 32 bits or skip & repeat 32 bits. In addition, we provide some examples on how such behaviors can be exploited in different security contexts. All these claims have been validated through actual fault injection experiments based on clock glitching.

The provided detailed description at ISA level clearly explains many faulty behaviors mentioned in the literature. It would also help in designing the countermeasures, in particular, at the software level, where the countermeasures are usually less expensive to implement.

In terms of perspectives, exploiting the aforementioned behaviors in a real life application would be very interesting. In addition, RTL fault simulation experiments are going to be conducted to validate the inferred fault models and explain the origin of such faulty behaviors at the microarchitectural level. Looking for countermeasures for the different faulty behaviors without significantly affecting the performance would also be very important and necessary. These countermeasures will be investigated both at software and hardware levels.

Finally, This work opens the door to investigate how fault injection would affect different architectures, taking into account the fact that the supported ISA provides variable-length instructions or not. These results could also motivate the (re)engineering of novel compressed instruction sets, which would be designed as to be immune to such vulnerability, even in the presence of faults.

Acknowledgment

This work has been supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) and the French National Research Agency in the framework of the "Investissements d'avenir" program (ANR-15-IDEX-02).

References

[1] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[2] B. Colombier, A. Menu, J.-M. Dutertre, P.-A. Moëllic, J.-B. Rigaud, and J.-L. Danger, "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)*. McLean, United States: IEEE, May 2019, pp. 1–10.

[3] 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.

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

[5] 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.

[6] 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.

[7] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software fault resistance is futile: Effective single-glitch attacks," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2016, pp. 47–58.

[8] I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle, and P. Maistri, "Microarchitecture-aware fault models: Experimental evidence and cross-layer inference methodology," in *2021 16th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2021, pp. 1–6.

[9] S. Skorobogatov, "Local heating attacks on flash memory devices," in *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, 2009, pp. 1–6.

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

[11] 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.

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

[13] J. Gratchoff, N. Timmers, A. Spruyt, and L. Chmielewski, "Proving the wild jungle jump," Technical report, University of Amsterdam, Tech. Rep., 2015.

[14] H. Pan, "High performance, variable-length instruction encodings," Ph.D. dissertation, Massachusetts Institute of Technology, 2002.

[15] Prasad Kulkarni, "16/32-bit arm-thumb architecture and ax extensions." http://www.ittc.ku.edu/ kulkarni/research/thumb_ax.pdf, [Accessed: March 2, 2022].

[16] MIPS Technologies, Inc., "micromipstm instruction set architecture uncompromised performance, minimum system cost ." https://s3-eu-west-1.amazonaws.com/downloads-mips/mips-documentation/login-required/micromips_instruction _set_architecture.pdf, [Accessed: March 2, 2022].

[17] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v compressed instruction set manual, version 1.7," *EECS Department, University of California, Berkeley, UCB/EECS-2015-157*, 2015.

[18] informIT, "Understanding arm architectures." https://www.informit.com/articles/article.aspx?p=1620207&seqNum=3, [Accessed: March 1, 2022].

[19] Tom Shanley — Mindshare, Inc., "x86 instruction set architecture." https://www.mindshare.com/files/ebooks/x86[Accessed: March 2, 2022].

[20] MIPS Technologies, Inc., "Mips32™ architecture for programmers volume ii: The mips32™ instruction set ." https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS _Vol2.pdf, [Accessed: March 2, 2022].

[21] ——, "Mips64™ architecture for programmers volume ii: The mips64™ instruction set." https://scc.ustc.edu.cn/zlsc/lxwycj/200910/W020100308600769158777.pdf, [Accessed: March 2, 2022].

[22] RISC-V, "RISC-V specifications," https://riscv.org/technical/specifications/, [Accessed: February 22, 2022].

[23] ARM Limited, "ARM architecture reference manual Thumb-2 supplement," https://developer.arm.com/documentation/ddi0308/d, [Accessed: February 22, 2022].

[24] 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.

[25] ARM Limited, "Armv7-m architecture reference manual." https://developer.arm.com/documentation/ddi0403/latest, [Accessed: February 22, 2022].