Cross-Layer Analysis of Clock Glitch Fault Injection while Fetching Variable-length Instructions

Ihab Alshaer[1,2*], Gijs Burghoorn[2], Brice Colombier[2,3], Christophe Deleuze[1], Vincent Beroulle[1], Paolo Maistri[2]

[1]Univ. Grenoble Alpes, Grenoble INP, LCIS, Valence, 26000, France.
[2]Univ. Grenoble Alpes, CNRS, Grenoble INP, TIMA, Grenoble, 38000, France.
[3]Université Jean Monnet Saint-Etienne, CNRS, Institut d Optique Graduate School, Laboratoire Hubert Curien UMR 5516, Saint-Etienne, F-42023, France.

*Corresponding author(s). E-mail(s): ihab.alshaer@univ-grenoble-alpes.fr;

Abstract

With the increasing complexity of embedded systems, the use of variable-length instruction sets has become essential, so that higher code density and better performance can be achieved. Security aspects are closely linked, considering the continuous improvement of attack techniques and equipment. Fault injection is among the most interesting and rising physical attack techniques. However, hardware designers and software developers lack accurate fault models to evaluate the vulnerabilities of their designs or codes in the presence of such attacks. In this article, we provide a proper characterization, at instruction set architecture (ISA) level, of several faulty behaviors that are experimentally observed when a processor running a variable-length instruction set is targeted. We include the binary encoding of instructions, and show how the obtained behaviors depend on the alignment in memory. Moreover, we give a deeper insight on previous results from the literature, that were still left unexplained. Additionally, we move downward at system level and consider the register-transfer level (RTL) to perform RTL fault simulation; This enables a better understanding of the faults propagation, validate the inferred fault models at ISA level, and reveal the origin of such faults at microarchitectural level. Finally, applying the given fault models leads us to provide vulnerability analysis on three different implementations of AES.

Keywords: variable-length instruction set, fault injection attacks, fault modeling, RTL fault simulation, vulnerability analysis.

1 Introduction

Embedded systems complexity, along with their running applications, is continuously increasing. This opens the door to two considerations: The need for high performance and new methods to deal with such advances and on the other hand, the emergence of new vulnerabilities exploitable by attackers at different levels. As sensitive data are frequently processed by embedded systems, some form of protection is necessary to prevent information leakage or modification. The actual processing and protection might be vulnerable to attacks that aim at extracting this sensitive information. Physical attacks in particular are a serious threat to embedded systems.

In the context of hardware security, fault injection is an efficient physical attack, belonging to the family of active attacks [1]. In this setting, the attacker has physical access to the digital device or its immediate environment, and tries to change the normal behavior of the device by injecting one or more faults, which may lead to an erroneous behavior that could be further exploited as a vulnerability.

To inject a fault, a physical interference is applied on the digital device: radiations [2], laser light [3, 4], electromagnetic pulses [5, 6], variations of power supply [7], perturbations of clock signal [8, 9], or changes in the environmental conditions such as the temperature [10] or else.

A detailed knowledge of the impact of the faults is thus necessary to protect embedded systems, especially the most complex microcontrollers and advanced microprocessors. This entails identifying, researching, and evaluating the flaws that may result in exploitable vulnerabilities at various levels of abstraction. Yet, it also calls for developing reasonable-cost countermeasures at these levels.

1.1 Cross-layer analysis

With the increasing complexity of embedded systems, characterizing the effects of faults based on a single level of analysis could lead to incomplete fault models, which are the abstract representations of the underlying physical phenomena. Thus, there is a need to follow and analyze the propagation of the faults at different abstraction levels. For example, one can investigate at Register-Transfer Level (RTL), Instruction Set Architecture (ISA) level, or binary instruction-encoding level. Such analysis helps to better understand the effects of a fault. From there, efficient and well-tailored countermeasures can be designed as a response.

In [11], we have presented a cross-layer inference methodology to provide optimized sets of fault models at hardware and software levels of an embedded system. In our approach, each fault model is validated and explained by the fault effect description at another level of abstraction, in order to strengthen the consistency of the given models. To achieve this, we conduct an analysis that involves comparing the observed faulty behaviors from physical fault injection experiments with the results obtained through simulation experiments at both hardware and software levels, as illustrated in Figure 1. In order to simulate faults at these levels, we utilize inferred fault models. Through an iterative application of this comparative process, we can propose a collection of realistic fault models. In this work, we have been following the same methodology to provide optimal set of fault models at three different levels: RTL, binary encoding, and ISA levels. This has been done by targeting a variable-length instruction set. The rationale behind this firstly came after performing preliminary experiments, of both clock glitch and RTL fault simulation, in [11]. We could observe diverse faulty behaviors as a result of adding a single `NOP` instruction to the target program.

The presented fault models can effectively help in analyzing possible vulnerabilities of software codes and/or hardware designs; Thus resulting in the creation of more efficient countermeasures against fault attacks.

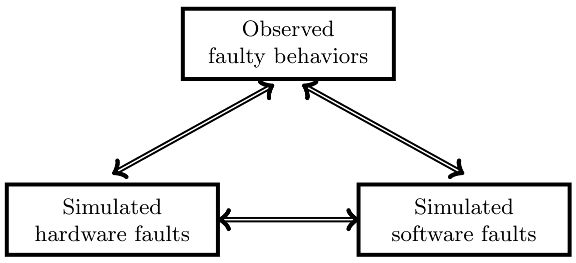


Fig. 1: Comparison of simulations and physical injection results.

1.2 Fault effect characterization at ISA level

Several research studies have characterized faults at ISA level, due to the fact that it can be considered as the focal point for bringing high (software) and low (hardware) levels of abstraction together. The majority of the submitted fault models have described the effects on the instruction itself: for this reason, “instruction skip” and “instruction corruption” are natural fault models.

In particular, the instruction skip fault model, also described as a replacement with one or more `NOPs`, can affect one [6, 8, 9], two [9], or multiple instructions [4, 5, 12]. It is to be highlighted that in all of the previous works, the authors have always referred to the instruction skip fault model only for *complete* instructions, either one or more.

In [4] or [5], the authors have observed that the skip is in line with a given number of bytes being related to the size of the instruction cache. However, it always means skipping an integer number of instructions.

Regarding the instruction corruption fault model, the reaction may translate into a corruption of the opcode [3] or the operands (destination or source ones) [3, 6, 7]. These studies have described the instruction corruption fault model considering the instruction as the basic element. As we highlight in this article, several monitored effects have remained unexplained.

In [9], the authors have provided fault effect characterization at ISA level after performing clock glitch fault injection campaigns on an Arm Cortex-M4 processor. Some behaviors could not be explained, such as the corrupted values found in some registers, independently of their actual use in the code. Additionally, the reason for having a single or double instruction skip fault model has been unclear. Similarly, electromagnetic fault injection campaigns on an Arm Cortex-A9 processor have been carried out in [6] to characterize the faults at ISA level as well. The authors have explicitly stated that some of the obtained faults remained unexplained. In addition, [13] have described the results of electromagnetic fault injection on two modern processors: an Arm BCM2837, which embeds an Arm Cortex-A53, and an Intel Core i3-6100T CPU. They have also provided a characterization at ISA level to suggest general fault models for different architectures, one being "random register corruption". Some of the faults they captured have been left unexplained, with an unknown fault model. Finally, in [7, 14, 15], the authors have stated that, as a result of fault injection attacks, alterations of the program counter are code-dependent in terms of instructions, registers and/or immediate values, providing no further explanation.

The existing fault models in the literature could not explain all behaviors in other ways than resorting to *random corruption*, either in the instructions or data. This actual work can now unravel the rationale behind several of the inferred fault models, thereby clarifying most of the previously unexplained faulty behaviors. The previous studies have not considered whether the targeted ISA could support variable-length instructions, nor have they used this knowledge to explain the results under fault injection. Specifically, they have not regarded whether the instruction bits fetched from the memory would correspond to complete instructions, knowing that the fetch size is always fixed whereas the ISA may support variable-length instructions. How such information could be exploited in a security application, or taken into account when designing countermeasures, has not been part of previous research.

1.3 Contributions

In this article, which is an extended version of our work in [16], we target a **second** 32-bit microcontroller, which embeds a different processor and has different flash memory access size. As a result, we confirm the two new inferred fault models: "skip" and "skip & repeat", for a specific number of bits, applied on the *binary encoding* of the instructions. These two models enable to explain a wider range of the obtained faulty behaviors at ISA level, regardless of the target instructions and target device. A proper description of the effects of the observed faulty behaviors is provided.

In addition, we extend the analysis of the clock glitch injection to cover more levels of abstraction, focusing particularly on **hardware level**. Here, we give a detailed description of the propagation of the glitch injection from low to high levels of abstraction. Starting from the **post-synthesis** timing simulation of the clock glitch on an FPGA, we progress through the **RTL** fault simulation and description of the fault propagation to various **microarchitectural** components. Finally, we analyze the effect of the fault at the level of **binary encoding** of instructions, before eventually detailing the monitored faulty behaviors at **ISA** level. Such analysis helps in designing efficient countermeasures at different levels: software and hardware, taking into consideration their cost and effects on performance.

We then give a **real-life** example of the use of these presented fault models to perform vulnerability analysis on three different implementations of the Advanced Encryption Standard (AES) algorithm.

1.4 Outline

This article is organized as follows: Section 2 provides the necessary background on variable-length instruction sets. Section 3 describes the experimental setup, then experimental results are reported and discussed in Section 4. Section 5 relates the simulation experiments performed at hardware level. These experiments are then compared and analyzed in Section 6, which details the cross-layer analysis. In Section 7, we carry out a vulnerability analysis of three different AES implementations, by applying the presented fault models. The article is concluded along with future research perspectives in Section 8.

2 Variable-length Instruction Sets

Reducing code size is a well-known method to reduce power consumption and memory usage. It lowers the overall cost of an embedded system highly affected by program coding and the fetch stage in the pipeline [17]. Code size reduction, targeting the highest possible code density, can be achieved by using a variable-length instruction set [17–20]. Additionally, less power is consumed due to the smaller number of fetches [18].

A variable-length instruction set can be defined as a combination of two instruction sets: a first set of short instructions (with respect to their encoding), providing the same functionality as if possessing larger encoding; and a second set composed of instructions with larger encoding that cannot be compacted while giving the same functionality. For this reason, the first set can also be referred to as the *compressed set*. High code density is achieved by the compressed instructions, while the second set allows for the preservation of high performance and expressive power. An example of the effect of ISA on cost and performance is the instruction cache: shorter encodings need smaller caches for the same performance [20, 21]. Therefore, having a shorter encoding induces fewer cache misses, with a given size, thus increasing the overall throughput of the processor. On the other hand, dealing with different length of encoding increases the complexity of the instruction decoder [21].

Several different variable-length instruction sets exist. The x86 [22] instruction set, supported by Intel and AMD processors, offers various lengths of encoding from 1 to 15 bytes. Another example of a variable-length instruction set is microMIPS [19], providing a set of 16-bit instructions that correspond to the most commonly used ones, in addition to all the instructions from the MIPS32/64 instruction sets [23, 24]. MicroMIPS shows 35 % smaller code size although almost similar performance as MIPS32 [19]. In 2015, a draft proposal [20] was published to procure 16-bit encodings for some instructions in the RISC-V instruction set. This new instruction set has been known as RISC-V Compressed (RVC) and reduces the code size by more than 25 % [20]. Finally, most Arm processors, including the Arm Cortex-M4 and Cortex-A9 mentioned above, support a dedicated variable-length instruction set as well, known as Thumb2 instruction set [25]. It consists of two sets of 16-bit and 32-bit instructions. Thumb2 delivers 30 % of code size reduction on average [18].

In this article, we have chosen an Arm Cortex-M3 and an Arm Cortex-M4 as target processors, for their wide adoption in embedded systems. Thumb2 is the target instruction set, but we assume that it is straightforward to generalize our findings to other variable-length instruction sets, as the fetching mechanisms are similar in most devices regardless of the alignment of the code in memories. For example, Intel and AMD architectures usually support fixed-size of caching blocks called *cache lines* [26, 27]. Consequently, as these cache lines have fixed size, Intel and AMD architectures allow splitting the instructions' data over the cache lines. This splitting may occur when an instruction spans across two or more cache lines. In such cases, the processor fetches the necessary cache lines to obtain the complete instruction.

3 Experimental Setup

In order to investigate the effects of fault injection on a variable-length instruction set, we have carried out several physical fault injection experiments. This aims at providing better characterization and description, at ISA level, for the wide range of faulty behaviors obtained when performing fault injection campaigns.

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

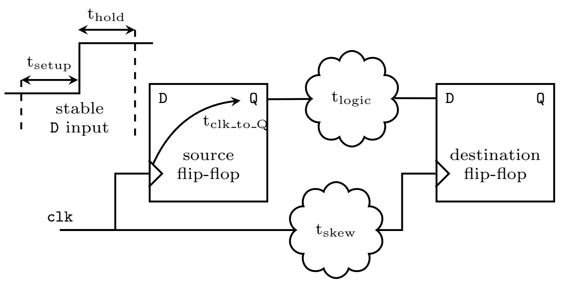


Fig. 2: Timing metrics in a simple digital design.

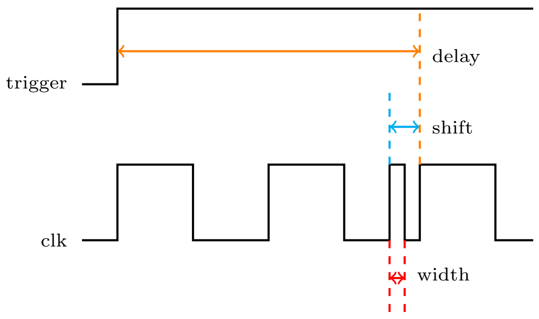


Fig. 3: Clock glitch parameters (from [9]).

3.1 Clock glitch fault injection

Applying perturbations to the main clock signal that is fed to the processor is a non-invasive and effective fault injection technique. Clock glitch is considered a low-cost fault injection technique compared to other techniques such as laser or electromagnetic pulses. Also, it can provide an acceptable controllability with respect to temporal accuracy, and hence the modified instruction(s) in the target program. However, since the glitch is injected on the global clock, one cannot know which microarchitectural element is affected as a result of the injection.

When performing clock glitch fault injection, a perturbation is injected just before or after the rising edge of the clock. This glitch is seen as a new, shorter clock cycle by the microprocessor, disrupting the regular pattern of the clock signal. Thus, a timing violation may occur, leading to various kinds of faulty behaviors.

In order to ensure correct operation of a digital design, timing must satisfy the setup and hold equations [28, 29], as presented in Equations (1) and (2) and illustrated in Figure 2 above where:

- t_{clk} is the clock period,
- t_setup is the duration for which data on the D input must be stable before the rising edge of the clock signal [30],
- t_hold is the duration for which data on the D input must be stable after the rising edge of the clock signal [30],
- t_logic is the propagation delay in the combinational logic between the source and destination flip-flops,
- t_skew is the time difference between the arrival of the clock signal at source D flip-flop and destination D flip-flop,
- t_clk_to_Q is the delay from the rising edge of the clock input to the Q output inside source D flip-flop [31].

$$t_{\text{clk\_to\_Q}} + t_{\text{logic}} + t_{\text{setup}} \leq t_{\text{clk}} + t_{\text{skew}} \tag{1}$$

$$t_{\text{clk\_to\_Q}} + t_{\text{logic}} \geq t_{\text{hold}} + t_{\text{skew}} \tag{2}$$

Performing clock glitch fault injection could result in a violation of the inequalities found in one or both of Equations (1) and (2), leading to observe a faulty behavior.

In this work, the ChipWhisperer [32] environment has been used to perform the clock glitch fault injection campaigns. In this setting, three parameters must be tuned, as shown in Figure 3:

- delay: the time between the rising edge of the trigger signal (used for synchronization) and the rising edge of the target clock cycle.
- shift: the time between the rising edge of the glitch and the rising edge of the target clock cycle.
- width: the duration of the glitch itself.

It is worth mentioning that while this article employs clock glitch using ChipWhisperer environment for fault injection, the findings presented can be extrapolated to other fault injection techniques relying on timing violations. This encompasses techniques such as voltage glitch [33–35] and electromagnetic fault injection [36, 37]. Moreover, recent studies [38–40] have demonstrated the vulnerability of modern systems supporting Dynamic Voltage and Frequency Scaling (DVFS) to glitch attacks. These attacks can be executed remotely through software, eliminating the need for a dedicated glitch fault injection setup.

3.2 Target devices

The target devices are two 32-bit microcontrollers: STM32F1, which embeds an Arm Cortex-M3 processor, and STM32L4, which embeds an Arm Cortex-M4 processor. The Arm Cortex-M3 and Cortex-M4 cores both include a 3-stage pipeline: fetch, decode, and execute. Both are based on the ARMv7-M [41] architecture and support the Thumb2 instruction set, consisting of variable-length instructions as mentioned in the previous section: 16-bit and 32-bit instructions.

In the Arm Cortex-M3 device, the fetch size from the memory to the AHB (Advanced High-performance Bus) is fixed and equal to 32 bits, regardless of the size of the instruction. Hence, as a result of having variable-length instructions, the fetched 32 bits can belong to one of the cases in Figure 4 or Figure 5. Figure 4 represents the fetching cases when code is aligned in memory, while Figure 5 represents the misaligned cases.

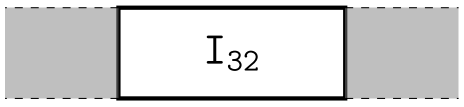
The Arm Cortex-M4 device supports cache lines of 64 bits. Therefore, in this case the flash memory access size is 64-bit wide. Consequently, the fetched 64 bits from the memory to the cache line can belong to two combined blocks of cases in Figure 4 or Figure 5. How these different possibilities affect the observed execution, as a reaction of fault injection campaigns, will be addressed in Section 4.

The processor detects whether the instruction that is about to be executed is a 16-bit or 32-bit one by analyzing the five most significant bits of the half-word that arrives first [41]. If these five bits have one of the following three values, then the word contains a 32-bit instruction:

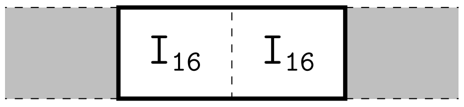
- `0b11101`,
- `0b11110`,
- `0b11111`.

All the other values define a 16-bit instruction. This understanding is central to unravel the monitored faulty behaviors, as detailed in Section 4.

In the following sections, big-endian representation for the binary encoding of instructions is used for readability. Thus, for a 32-bit instruction, the most significant 16 bits arrive first in the pipeline.

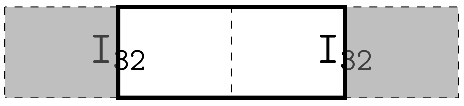


(a) Fetching one 32-bit instruction.

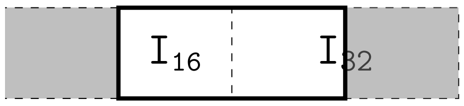


(b) Fetching two 16-bit instructions.

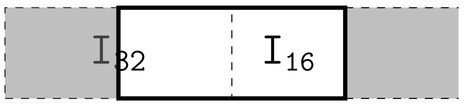
Fig. 4: Fetching aligned instructions.



(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. 5: Fetching misaligned instructions.

3.3 Target programs

Injection is performed into inline assembly instructions within a C program. To ease the process, the program is divided into three parts separated by `NOP` instructions:

- Prologue: instructions initialize the processor in a known state ahead the fault injection.
- Target: instructions are targeted by the fault injection as well as extra instructions which would emphasize any propagation effect for further monitoring.
- Epilogue: instructions read the value of the registers, specifically the general purpose registers R0 to R12; values are then transferred to the control computer through serial communication.

In the experiments, arithmetic instructions have been used in the Target part as shown in Listings 1 and 2. Such arithmetic instructions make it easier to characterize the fault effects.

Listing 1 shows an example of an *aligned* code. In this case, when 32 bits are fetched, they are either two full 16-bit instructions or one 32-bit instruction as shown in Figure 4. The first two instructions, `MOV` and `LSLS`, are 16-bit instructions. All the `ADD` instructions are 32-bit instructions. At the end of a normal execution, each register has a different value from the others; Faults can be more easily identified when they do occur. Various small immediate values are used with the `ADD` instructions to assess whether one would be replaced with a register number or vice versa, for example whether R3 becomes `0x3`, as detailed in Section 4.

```
MOV  R8, R4          // R8 = R4
LSLS R2, R0, 0x10    // R2 = R0 << 0x10
ADD  R1, R1, 0x6     // R1 = R1 + 0x6
ADD  R3, R3, 0xa     // R3 = R3 + 0xa
ADD  R4, R4, 0xb     // R4 = R4 + 0xb
ADD  R5, R6, R3      // R5 = R6 + R3
ADD  R3, R3, 0xf     // R3 = R3 + 0xf
```

Listing 1: Target part in the aligned code.

A *misaligned* code is illustrated in Listing 2. It is similar to Listing 1 except that the first `MOV` instruction has been removed. Only one 16-bit instruction is now fetched. Therefore, the code is misaligned. When 32 bits are fetched, they now belong to either two different 32-bit instructions, or one 16-bit instruction and half of a 32-bit instruction, as shown in Figure 5. Section 4 will develop how such a small modification of the Target code can greatly affect the observed faulty behaviors at ISA level.

```
LSLS R2, R0, 0x10    // R2 = R0 << 0x10
ADD  R1, R1, 0x6     // R1 = R1 + 0x6
ADD  R3, R3, 0xa     // R3 = R3 + 0xa
ADD  R4, R4, 0xb     // R4 = R4 + 0xb
ADD  R5, R6, R3      // R5 = R6 + R3
ADD  R3, R3, 0xf     // R3 = R3 + 0xf
```

Listing 2: Target part in the misaligned code.

The Prologue is always aligned in the code memory space and does not influence the overall code alignment, which only depends on the instructions of the Target part.

3.4 Injection parameters and classification

Each injection campaign involves repeating the clock glitch fault injection 10 000 times for each combination of glitch parameters to maximize the number of captured faults. Thus, for each of the presented examples in Section 4, 10 000 executions are conducted. Table 1 illustrates the shift and width values that allowed observing the presented faulty behaviors, for each target device. The values are expressed in percentage of one clock period. The negative value of the shift means that the glitch is injected before the rising edge of the targeted clock cycle. These shift and width values replicate comparable faulty behaviors when targeting microcontrollers from the same manufacturer (*i.e.*, other STM32F1 and STM32L4). This confirmation further simplifies and substantiates the reproducibility of the experiments presented. The glitch is timed accordingly to target specific instructions. Thus, the delay value depends on the target instructions, the number of instructions in the Prologue, and the number of `NOP` instructions that precede the Target part. A single glitch is injected during each program execution. Three outcomes can occur in reaction to the fault injection:

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

In the next section, we focus on an observable subset of captured faults, discarding crashes and silent cases. This subset is the **largest** subset of the obtained faults, and our focus is on the **occurrence** of these faults, not their probability. Nonetheless, Table 1 shows the percentage of occurrence for each of the presented faulty behaviors over 10 000 executions.

4 Experimental Results and Analysis

Different faulty behaviors have been observed after performing clock glitch fault injection campaigns. In the following subsections, we only focus on faulty behaviors related to two specific inferred

Table 1: Shift and width values used in Section 4 experiments, and percentage of occurrence of each faulty behavior over 10 000 executions.

	Cortex-M3			Cortex-M4		
Section	shift	width	occurrence	shift	width	occurrence
4.1.1	-49	4	9.06 %	-14	10	92.68 %
4.1.2	-	-	-	-13	10	100 %
4.1.3	-13	3	99.8 %	-	-	-
4.1.4	-	-	-	-6	2	75.98 %
4.2.1	-49	5	1.97 %	-	-	-
4.2.2	-12	3	99.51 %	-12	6	100 %
4.2.3	-	-	-	-13	10	100 %
4.2.4	-13	3	99.98 %	-	-	-
4.2.5	-	-	-	-6	2	65.34 %

fault models: The ones referring to the *encoding* of the instructions "skip" or "skip & repeat" a specific number of bits. It can either be 32 or 64 bits. These fault models are related to the fetch stage of the processor pipeline.

We will demonstrate 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 encoding level, *i.e.*, "skip" or "skip & repeat", are always the same in both cases. The last subsection provides further details about a specific behavior, where a *new* instruction is executed as a result of the clock glitch. All the occurrences of 32-bit faulty behaviors in this section were observed when targeting the Arm Cortex-M3 device. Some of them were also observed when targeting the Arm Cortex-M4 device, as shown in Table 1. On the other hand, 64-bit faulty behaviors were only obtained when targeting the Arm Cortex-M4 device. Their fetch size is the reason, as previously detailed in Section 3.2.

4.1 Aligned code scenario

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

Assuming that i is the line number that points to a 32-bit block of the target program binary encoding, then "skip" and "skip & repeat " fault models can be defined as such:

- 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.
- Skip 64 bits: the 32 bits at line i and the 32 bits at line $i + 1$ are skipped, and the execution resumes from line $i + 2$.
- Skip & repeat 64 bits: the 32 bits at line $i + 1$ and the 32 bits at $i + 2$ are skipped, while the 32 bits at line i, and the 32 bits at line $i - 1$ are repeated.

```
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 hexadecimal format.

The following subsections display samples of the observed "skip" and "skip & repeat" faulty behaviors after performing the fault injection campaigns.

Although we target a given instruction inside the target part for our discussion in the following subsections, the conclusions drawn from these examples can also be applied to other locations of the target program without any loss of generality, as the corresponding faulty behaviors are also observed.

For each subsection, faults are described at two different abstraction levels: binary encoding and ISA levels.

4.1.1 Skip 32 bits / single instruction skip

Except line 1 related to the binary encoding of two 16-bit instructions, skipping any line in Listing 3 has led to a single instruction skip. The reason being that each line corresponds to a full 32-bit instruction. Skipping the `ADD R4, R4, 0xb` instruction at line 4 in Listing 3 is an example; the observed execution at ISA level is reported in Listing 4.

Naturally, skipping line 1 in Listing 3 has led to a double instruction skip, since this line corresponds to two 16-bit instructions.

```
MOV R8, R4
LSLS R2, R0, 0x10
ADD R1, R1, 0x6
ADD R3, R3, 0xa
ADD R5, R6, R3
ADD R3, R3, 0xf
```

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

4.1.2 Skip 64 bits / double instruction skip

Skipping line 2 and line 3 in Listing 3 has brought a double instruction skip since these two lines contain two 32-bit instructions. Listing 5 shows the observed execution at ISA level for this sample.

```
MOV R8, R4
LSLS R2, R0, 0x10
ADD R4, R4, 0xb
ADD R5, R6, R3
ADD R3, R3, 0xf
```

Listing 5: Observed execution for Skip 64 bits / double instruction skip.

4.1.3 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 illustration of this fault model. The monitored execution at ISA level is exposed in Listing 6.

4.1.4 Skip & repeat 64 bits / double instruction skip and double instruction repeat

Skipping line 4 and line 5, and repeating line 2 and line 3 in Listing 3 is an example of this fault model. The observed execution at ISA level is shown in Listing 7.

```
MOV R8, R4
LSLS R2, R0, 0x10
ADD R1, R1, 0x6
ADD R1, R1, 0x6
ADD R4, R4, 0xb
ADD R5, R6, R3
ADD R3, R3, 0xf
```

Listing 6: Observed execution for Skip & repeat 32 bits / single instruction skip & single instruction repeat.

```
MOV R8, R4
LSLS R2, R0, 0x10
ADD R1, R1, 0x6
ADD R3, R3, 0xa
ADD R1, R1, 0x6
ADD R3, R3, 0xa
ADD R3, R3, 0xf
```

Listing 7: Observed execution for Skip & repeat 64 bits / double instruction skip & double instruction repeat.

4.2 Misaligned code scenario

The focus is now on *misaligned* code. It is achieved by removing the leading `MOV` instruction, a 16-bit instruction, from the target part in the program as displayed in Listing 2. Listing 8 shows the binary encoding of the misaligned code.

Each line still contains 32 bits, but unlike the previous case, it does not correspond to a single 32-bit instruction. For the sake of clarity, each instruction has been highlighted with a different color: the reader will notice that each 32-bit instruction is split in two consecutive lines.

```
0402f101
0106f103
030af104
040beb06
0503f103
030fbf00 // bf00: NOP.
```

Listing 8: Binary encoding for the misaligned code in hexadecimal format.

There again, similar fault models are used on 32-bit or 64-bit data. However, as a line now consists of two 16-bit blocks belonging to two separate instructions, as previously shown in Figure 5, different faulty behaviors have been examined at ISA level. The actual recorded faulty behaviors depend on the target location of the glitch injection. The following subsections provide various observations of each model, *i.e.*, different examples of "skip" and "skip & repeat" models. It is interesting to highlight that the monitored behaviors in this scenario are significantly more complex than in the aligned code scenario. Among several outcomes, for instance, we have witnessed double instruction corruption or even new instruction execution.

4.2.1 Skip 32 bits / single instruction skip and single instruction corruption

This case refers to Figure 5a, and happens when skipping line 3 in Listing 8 for example. The observed execution at ISA level is in Listing 9.

```
LSLS R2, R0, 0x10
ADD R1, R1, 0x6
ADD R4, R3, 0xb  // f103040b
ADD R5, R6, R3
ADD R3, R3, 0xf
```

Listing 9: Observed execution for Skip 32 bits / single instruction skip and single instruction corruption.

The `ADD R4, R4, 0xb` instruction is skipped and the `ADD R3, R3, 0xa` instruction is corrupted, replacing two of its operands by the corresponding ones from the skipped instruction [16].

4.2.2 Skip 32 bits / double instruction skip and new instruction execution

Figure 5b illustrates this case; Line 1 in Listing 8 is skipped for example. As a result, `0x0106` arrives first to the core and, since the five most significant bits of `0x0106` are `0b00000`, a 16-bit instruction is executed, with the following encoding: `0x0106`. This instruction is `LSLS R6, R0, 0x4`. The other instructions in the target program are not affected and are executed normally.

Listing 10 displays the monitored execution of this example at ISA level. The first instruction is painted blue since its encoding comes from the original blue instruction. It is shown that two instructions have been skipped: a 16-bit and a 32-bit one; a *new* 16-bit instruction has been executed instead.

To prove that the new `LSLS` instruction is not related to the original `LSLS` instruction, section 4.3 supplies a variety of instructions that can be executed as a result of this observed behavior.

```
LSLS R6, R0, 0x4 // 0106
ADD R3, R3, 0xa
ADD R4, R4, 0xb
ADD R5, R6, R3
ADD R3, R3, 0xf
```

Listing 10: Observed execution for Skip 32 bits / double instruction skip and new instruction execution.

4.2.3 Skip 64 bits / single instruction corruption, double instruction skip and new instruction execution

This case relates to two consecutive blocks of Figure 5a, and happens when skipping lines 2 and 3 in Listing 8 for instance. Listing 11 shows its observed execution at ISA level.

```
LSLS R2, R0, 0x10
ADD R0, R1, 0x0  // f1010000
LSLS R3, R1, 0x10 // 040b
ADD R5, R6, R3
ADD R3, R3, 0xf
```

Listing 11: Observed execution for Skip 64 bits / single instruction corruption, double instruction skip and new instruction execution.

The `ADD R1, R1, 0x6` is corrupted after skipping its bottom half (`0x0106`). Since `0xf101` means a 32-bit instruction is about to be executed, it should be padded with another 16-bit block. However, in this case, it is not padded with 16 bits from an upcoming or skipped instruction, as in Section 4.2.1. Instead, it is padded with zeros: `0x0000`. This might be due to the unavailability of another 16-bit block from the bus or in the fetch unit, when the decision of executing a 32-bit instruction is taken. The `ADD R3, R3, 0xa` instruction is skipped as all of its bits had been discarded. 0x040b, the remaining part of the original ADD R4, R4, 0xb instruction, is executed as a *new* 16-bit instruction, since the five most significant bits, 0b00000, identify a valid 16-bit instruction: LSLS R3, R1, 0x10. The skipped 32 bits (0x030af104) might also be replaced with zeros. Here it is worth mentioning to say that the encoding of 0x0000 belongs to a 16-bit dummy instruction that has no effect: MOVS R0, R0. Thus, executing 0x00000000 would have no effect.

This behavior is also observed in terms of 32 bits, not only 64 bits, when the code is misaligned in memory.

4.2.4 Skip & repeat 32 bits / double instruction corruption

Figure 5a as a reference; when two half instructions are affected. By way of illustration, in Listing 8 it happens when line 4 is skipped and line 3 is repeated.

```
LSLS R2, R0, 0x10
ADD R1, R1, 0x6
ADD R3, R3, 0xa
ADD R3, R4, 0xa   // f104030a
ADD R5, R4, 0x3   // f1040503
ADD R3, R3, 0xf
```

Listing 12: Observed execution for Skip & repeat 32 bits / double instruction corruption.

As a consequence, two instructions are corrupted, it shows in Listing 12. 0xf104 means that the instruction to be implemented is a 32-bit one, as the five most significant bits are 0b11110. The repeated red section (0x030a) is part of the new executed instruction.

In addition, since 0xf104 is repeated, another *new* 32-bit instruction is performed. Its first half is from the ADD R4, R4, 0xb instruction (0xf104) and its second one is from the 16 bits that remained from the ADD R5, R6, R3 instruction at line 5 in Listing 8 (0x0503).

To report the observed behaviors at ISA level and generalise the obtained faults to other target programs with similar structure, we explain the corruption of two 32-bit instructions as follows:

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

4.2.5 Skip & repeat 64 bits / double instruction corruption, single instruction skip and single instruction repeat

This case concerns two consecutive blocks of Figure 5a, when two 32-bit blocks are skipped and two 32-bit blocks are repeated; it happens when skipping lines 4 and 5 and repeating lines 2 and 3 in Listing 8 for example. Listing 13 displays its recorded execution at ISA level.

```
LSLS R2, R0, 0x10
ADD R1, R1, 0x6
ADD R3, R3, 0xa
ADD R1, R4, 0x6   // f1040106
ADD R3, R3, 0xa
ADD R3, R4, 0xf   // f104030f
```

Listing 13: Observed execution for Skip & repeat 64 bits / double instruction corruption, single instruction skip and single instruction repeat.

The ADD R4, R4, 0xb instruction is corrupted, as the bottom half of it (0x040b) is replaced with the repeated part of the blue instruction (0x0106). The ADD R5, R6, R3 instruction is skipped, and instead, the ADD R3, R3, 0xa instruction is repeated, since all of its 32 bits have been repeated. Finally, the ADD R3, R3, 0xf instruction is corrupted as the top half of it (0xf103) is replaced with the repeated part of the ADD R4, R4, 0xb instruction (0xf104).

4.3 More on the ability to execute a new instruction

Since the encoding of the new Logical Shift Left instruction in 4.2.2 is coming from the destination register and the second source operand in the ADD R1, R1, 0x6 instruction, then changing these two operands to other values essentially enables to "craft" new instructions.

Table 2: Possible 16-bit instructions coming from different destination registers and/or immediate value in the original 32-bit instruction.

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`

Table 2 shows examples of new instructions when changing these two operands. They have all been experimentally validated by clock glitch fault injection campaigns on both the Arm Cortex-M3 and Cortex-M4 devices. In other words, replacing the `0xf1010106` instruction from Listing 8 with an instruction from the first column of Table 2 enables to observe the execution of the corresponding instruction in the third column of Table 2. The second column illustrates the encoding of the new instruction, coming from the least significant 16 bits of the original 32-bit instruction.

By generating all of the 65 536 possible 16-bit combinations and disassembling them to check whether they were valid Thumb2 instructions, we have identified more than 58 000 valid 16-bit instructions. Each of these can be executed as a result of this specific fault model, regardless of the opcode of the original 32-bit instruction in the target program.

The consequences are particularly enlightening when it comes to previously observed fault models left unexplained. Trouchkine *et al.* examined a corruption of `R8` and `R0` when targeting a series of `AND R8, R8, R8` instructions [13]. They stated that corruption is sometimes a complete reset of the register. This `AND` instruction has the following encoding: `0xea080808`. Thanks to our analysis, we can fully explain the corruption they observed on the Arm Cortex-A53 processor, supporting Thumb2 instruction set. The fault injection leads to the creation and execution of the 16-bit instruction `0x0808`. It is the encoding of `LSRS R0, R1, 0x20`. This operation brings a reset of R0, since the value in R1 is shifted to the right by 32 bits and its result, obviously 0, is stored in R0.

Many instructions from Table 2 may lead to violate various security properties. For instance, executing an `LDR` (Load) instruction could reveal some values in the memory, breaking the confidentiality property. As another example, executing the `EORS` (XOR) instruction may allow an attacker to observe a collision in a cryptographic algorithm, which could lead to recover secret data. Or moving an immediate value to a register could result in corrupting a loop counter value if this register is used as the counter itself. In our previous work [16], we demonstrated how the program counter could be modified to a value stored in one of the registers by exploiting this specific behavior.

5 Hardware Fault Simulation

Clock glitch fault injection cannot controllably target a specific microarchitectural component. A cross-layer analysis approach is indispensable to understand the propagation of faults and identify reliable fault models. By following the cross-layer analysis methodology, to better understand and pinpoint the origin of the faulty behaviors described above, RTL fault simulation experiments have been performed on the same target programs in Listing 1 and Listing 2. This should confirm the observations regarding the executed instructions. The RTL description used in the simulation is related to the Arm Cortex-M3 processor. Access to this description is provided under the Arm Academic Access Agreement (AAA). Questa simulator (version 2021.3_2) has been used to perform the RTL fault simulation.

The following subsections provide the chosen methodology of RTL fault simulation, the RTL fault models permitting to observe the same faulty behaviors at higher levels, and explanations of these fault models using post-synthesis clock glitch simulation.

5.1 RTL fault simulation methodology

With RTL fault simulation, the goal is to find the signals or internal registers[1] that, when faulty, lead to the erroneous behaviors described in the

[1]The word “register” here does not refer to a purpose register, but to one or multiple D-flip flops that store an internal value.

previous sections, and hence, to reveal their origin and better understand their propagation at lower levels of abstraction. However, since there are thousands of registers in the RTL description of a processor such as the Arm Cortex-M3, a specific methodology is necessary to accelerate the analysis and find the targeted registers in reasonable time. To this end, path delay analysis within specific architectural components or modules has been set up. Clock glitch being likely to cause timing errors, then critical or almost critical paths are more inclined to be faulty [34, 42, 43]. Consequently, the registers involved in such paths have more chance to capture faulty values as a result of path timing violations.

Based on the above, our methodology consists in faulting the involved registers in the paths that come first, according to the maximum paths delays, when generating a timing analysis report for specific architectural components or RTL modules. The destination and source registers, of these paths, are the involved registers. The fault models detailed below are then applied on them. Vivado 2019.2 simulator has been used to generate the timing analysis reports.

5.2 RTL fault models

The aforementioned methodology has been applied to the RTL modules that are relevant to the *fetch* stage of the processor, in order to confirm our assumption regarding the origin of the observed faulty behaviors.

To implement RTL fault simulation, two fault models have been introduced. They follow the intuition behind timing violations, being that the value of a register might not be correctly updated.

The first RTL fault model consists in *preventing* the update of a register value at a given clock cycle. Therefore, the register keeps its previous value. This RTL fault model has validated the "skip & repeat 32 bits" faulty behavior at binary encoding level. All the registers located in the fault propagation path, shown in Figure 6, have generated the same faulty behavior; meaning the faulty register can either be in the interface between the fetch unit and the AHB, in the AHB component, or in the interface between the AHB and the flash memory. The propagation starts from the interface between the fetch unit and the AHB: This interface and the fetch unit are parts of the core itself.

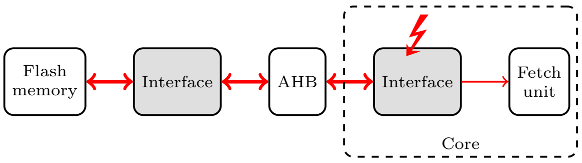


Fig. 6: Fault propagation path for Skip 32 bits or Skip & repeat 32 bits fault models.

Here, the fault propagation follows the opposite direction of the instruction data path from the flash memory to the processor core, *i.e.,* the *fetch* path. Then, the fault propagation continues from the flash memory to the fetch unit. Thus, the origin faulty register could be in the opposite path of the *fetch*, or in the same path as the *fetch*.

The second fault model involves *anticipating* the update of the value of a register at a given clock cycle; at clock cycle i, the value that the register would actually store at clock cycle $i + 1$ is loaded. This RTL fault model leads to the "Skip 32 bits" behavior at binary encoding level. Fewer registers, when targeted, generate the "Skip 32 bits" behavior, and all are located in the interface between the fetch unit and the AHB as shown in Figure 6.

It is noteworthy that there can be tens of registers present within a single microarchitectural component. However, the count of susceptible registers (*i.e.,* the ones we applied the fault simulation on) varies based on the specific microarchitectural component being targeted, typically falling between 2 and 6. The reason not to explicitly specify the registers is tied to the confidentiality of this information, governed by our AAA agreement with Arm.

5.3 Post-synthesis timing simulation

In order to explain how the aforementioned RTL fault models have validated the same faulty behaviors observed at higher levels of abstraction, another layer has been taken into account: performing post-synthesis clock glitch simulation on an Artix-7 FPGA using Vivado 2019.2. There again, the implementation of this simulation has targeted modules related to the "fetch" stage of the processor. The objective of this test has been to investigate on the effects of the clock glitch over specific registers within these modules from the architectural perspective.

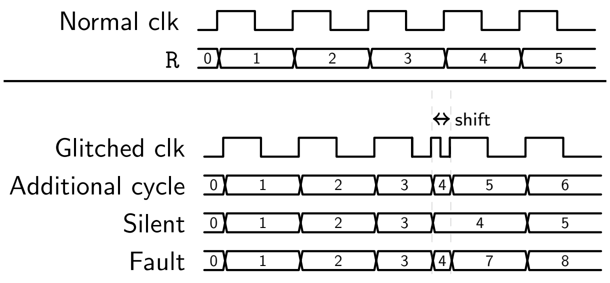


Fig. 7: Possible effects of post-synthesis clock glitch simulation on register R.

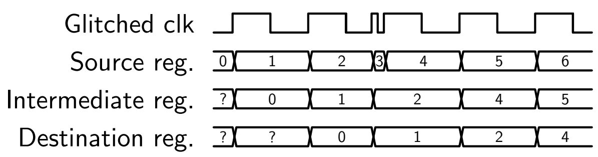


Fig. 8: “Skip” behavior description with post-synthesis timing simulation: Skip 3.

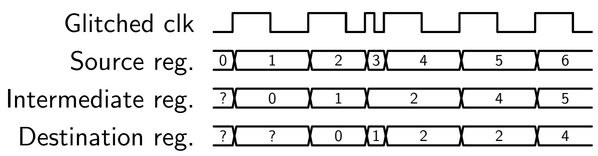


Fig. 9: “Skip & repeat” behavior description with post-synthesis timing simulation: Skip 3 and repeat 2.

These simulation experiments consist of employing a VHDL file that represents the test-bench. It is used to initialize the input signals and to generate the glitch over the main clock signal. The output of the simulation is analyzed using the resulting waveform. Each simulation execution takes a few seconds, as the post-synthesis simulation is done only on a few RTL modules. The simulation is repeated hundreds of times while changing the glitch parameters and the targeted clock cycle.

As a result, three cases have emerged, as presented in Figure 7:

- *Additional cycle*: the glitch acts as an extra clock cycle. In other words, register R is normally updated because of the glitch.
- *Silent*: the glitch has no effect on the values of R. However, it has been observed that R is updated at the rising edge of the glitch and not at the rising edge of the following clock cycle. Hence, the final value is not affected.
- *Fault*: faulty values have been monitored at the rising edge of the clock cycles following the glitch, either the first rising edge or the ones after, as a result of a timing violation. Various faulty values are captured based on fine-tuning of the glitch parameters. Among them, we could observe that the same value occurs two clock cycles earlier, some values occur two cycles later, some are incoherent values. And more.

The glitch parameters are the reason behind these three different effects. It has to be remembered that the effect of the glitch, with given parameters, might not be the same for all the registers within the same module; the reason being that not all paths between registers have the same delay. Based on that, it's possible to observe faulty behaviors due to the various effects on the registers. Therefore, by considering the glitch effect on the output of source and destination registers, Figure 8 and Figure 9 report how the “Skip” and “Skip & repeat” behaviors can come up. The effect of the glitch is similar on the source and the intermediate register in both figures: the source register is subject to *Additional cycle* effect, while the intermediate undergoes *Silent* effect. However, in the second clock cycle that follows the glitch, the intermediate register captures the faulty value 4, the one available at that time. Thus, the glitch effect on the destination register determines whether the resulting erroneous behavior would be a Skip or Skip & repeat. In Figure 8, the destination register is subject to *Silent* effect, and hence, no value is repeated. Therefore, the obtained effect is Skip. On the other hand, in Figure 9, the destination register takes *Additional cycle* effect, which leads to capture the value 2 twice. Thus, in this case, a Skip & repeat behavior is procured. All these detailed observations validate the use of the aforementioned RTL fault models.

Figure 10 shows the result of applying RTL fault models on registers. In order to model Skip & repeat at RTL level, *preventing the update* fault model is applied to either the intermediate or the destination register or even any further destination register; this results in getting two consecutive 2s instead of 2 then 3. On the other hand, aiming for Skip requires *anticipating the* *update* fault model applied at the source register; thus 4 appears instead of 3 after 2.

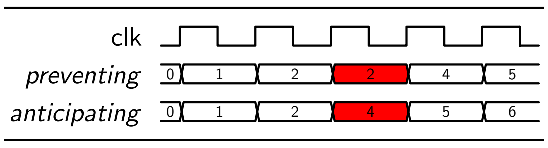


Fig. 10: Result of applying RTL fault models.

5.4 Summary

These simulation experiments have confirmed our observations on executed instructions in the previous clock glitch tests. Therefore, they validate the inferred 32-bit fault models at binary encoding level, *i.e.,* "Skip 32 bits" and "Skip & repeat 32 bits". They also corroborate our assumption on the origin of such faulty behaviors: the *fetch* stage in the pipeline. In particular, we successfully pinpointed not only the vulnerable registers, but also the entire fault propagation path between the Decode stage and the Flash memory. Additionally, they have unfolded the discriminating rationale behind the Skip and Skip & repeat fault models. We are confident that performing identical simulations on RTL description with 64-bit cache line size would unravel the corresponding 64-bit fault models as well.

Furthermore, it has been shown that the same effect at hardware level could lead to different effects at software level, as the effect at software level varies based on the alignment of instructions. This is very relevant because, before that, these behaviours were considered random.

It is worth mentioning that clock glitch can cause **other** faulty behaviors than "skip" and "skip & repeat" while fetching instructions, as detailed in [44]. In which, a glitch affects a part of a register, not all of its flip-flops as described in previous sections. However, observing such faulty behaviors is less probable than observing "skip" and "skip & repeat". Additionally, Section 5 helps in explaining the origin of the observed faulty behaviors in [44], at the hardware level, in a precise way.

It should be mentioned that in order for the presented hardware approach to work, it requires simulations to be performed at different levels, which means that the corresponding descriptions should be available. For instance, our test case was based on an Arm architecture, where we had actual devices available for our experiments and RTL descriptions (thanks to AAA) of the same architecture. This has a few consequences: first of all, this methodology cannot be fully applied on designs where descriptions are not available (for instance, Intel or AMD architectures), as it would not be possible to delve into levels lower than ISA. Secondly, it means also that the more information we have, the more precise and effective the methodology would be. Using our test case again as a reference, we were able to cross several levels (ISA, RTL, and post-synthesis), but our analysis based on critical paths is an approximation of the final device, as the layout of the actual microcontrollers that we used was not available. Nonetheless, we can consider that post-synthesis analysis can give results close enough to post-layout analysis: the loss of accuracy can be mitigated by slightly increasing the search space at the higher level.

6 Cross-layer analysis: bottom-up summary

The previous sections have illustrated how the effects of clock glitch injection can be modeled from low to high levels of abstraction. In this cross-layer analysis process, an inferred fault model at a specific level is validated and explained by faulty behaviors and effects observed at other levels of abstraction.

Firstly, by performing post-synthesis clock glitch simulation, three different effects have been monitored: *Additional cycle*, *Silent*, and *Fault*. They validate the use of two fault models at higher RTL level: *preventing the update* or *anticipating the update* of a register using the next value instead of the current one. These two models, when they become erroneous behaviors with respect to binary encoding level, confirm the use of Skip and Skip and repeat fault models. Similarly, the two fault models at binary encoding level can be translated into two faulty behaviors at ISA level, and hence, provide the validation of large set of fault models at such higher abstraction level.

This would include "single instruction corruption", "single instruction skip", "single instruction repeat", "new instruction execution", etc.

A notable contribution of this work is that fault models at ISA level, looking like random behaviors at first sight, can be clearly explained by studying the fault at a lower level of abstraction. Previous works [7, 13, 45, 46] tried to explain "instruction skip" and/or "instruction corruption" as random bit flips at binary encoding or RTL levels; we have now proved that these faulty behaviors are not simply random bit flips.

It is to be remembered that more ISA fault models can arise by combining different blocks of the fetching cases shown in Figure 4 and Figure 5. There, block combination takes into account different sizes of flash memory access or instruction cache, such as 64-bit as already detailed in 4.1.2, 4.1.4, 4.2.3 and 4.2.5. In addition, some more cache sizes may be found in other microcontrollers, such as 128-bit widths.

Table 3: Fault models description at three different abstraction levels: RTL, binary encoding, and ISA levels.

RTL	Binary encoding	ISA
• preventing the update.	• skip & repeat 32 bits.	• single instruction skip & single instruction repeat, • double instruction corruption.
• anticipating the update.	• skip 32 bits.	• single instruction skip, • double instruction skip, • single instruction skip and single instruction corruption, • double instruction skip and new instruction execution.

7 Vulnerability analysis on AES

In their work [47], the authors showed, through static analysis, that flipping some bits of variable-length instructions encoding could realign the code, resulting in dangerous erroneous behaviors. In addition, it was claimed in another paper [48] that variable-length instructions might bring more return-oriented programming attacks (specific attacks detailed in [49]).

This section presents a vulnerability analysis of AES encryption algorithm using the preceding fault models, specifically when the code is misaligned in memory. The study focuses around three different AES implementations: *BroAES*[2], *TinyAES128*[3] and *MbedTLS-AES*[4]. Such tests illustrate real-life applicability of misaligned faults. First, we concentrate on an exploitable branch logic error within *BroAES*, experimentally validated by clock glitch fault injection. Then, we cover two general observations with examples from *TinyAES128*, then *MbedTLS-AES*.

AES is a round-based symmetric encryption algorithm [50]. It iteratively transforms an input plaintext into a ciphertext. The last round slightly differs from the previous ones, omitting *MixColumns* transformation and adds an extra *AddRoundKey*. Consequently, software implementations of the algorithm usually loop over the iterations, while making an exception for the last round. The first section targets this branch logic.

7.1 Branch logic error on *BroAES*

A clock glitch attack on *BroAES* is depicted here, exposing the key in a known plaintext scenario with the use of the described fault models.

In *BroAES*, all rounds are encapsulated in a `for` loop. Depending on the round, some transformations are excluded. Listing 14 shows the instructions initializing the memory, used to keep track of iterations and branching.

Before the instructions at line 8, checking the loop condition and branch to the transformations, there are three `STR` instructions storing values onto the stack. Line 3 causes `[SP+4]` to contain the value of `num_rounds`. Later instructions use it to verify the loop condition and determine whether to perform *SubBytes* and *ShiftRows* transformations. Line 4 causes `[SP+8]` to contain the value of `num_rounds - 1`. Later, the loop body uses it to assess whether to perform *MixColumns* transformation. Line 6 makes `[SP+0]` contain the loop iteration value.

[2]https://github.com/brobwind/bro_aes
[3]https://github.com/kokke/tiny-AES-c
[4]https://github.com/Mbed-TLS/mbedtls

```
1  |b087|e9d0   // SUB   SP, 0x1c
                // LDRD  R7, R8,[R0,0x8]
2   7802|f8d2   // LDR.W R0, [R2, 0xb0]
3   00b0|9001|  // STR   R0, [SP, 0x4]
                // [SP+4] = num_rounds
4  |3801|9002|  // SUBS  R0, 0x1
                // STR   R0, [SP, 0x8]
                // [SP+8] = num_rounds-1
5  |2000|f102   // MOVS  R0, 0x0
                // ADD.W R4, R2, 0xb4
6   04b4|9000|  // STR   R0, [SP, 0x0]
                // [SP+0] = iteration
     count
7  |e9dd c000|  // LDRD  R12, R0, [SP]
8  |4560|da06|  // CMP   R0, R12
                // BGE.N <...>
```

Listing 14: Initialization of memory used for iteration.

Targeting the instruction `LDR.W R0, [R2, 0xb0]` at line 2, with the fault model described in Section 4.2.1 in order to skip 32 bits at line 3, leads to what is reported in Listing 15:

The value of `[SP+4]` not being set, it is therefore an arbitrary value. Since no earlier instructions touch this part of the stack, one can reasonably assume that in the first round `[SP+4]` is set to 0. The value of `[SP+8]` is set to `R0` which is initially a pointer to a stack value. The value of `[SP+0]` is normally handled and set to 0. Then, after executing the `LDRD` instruction at line 6, `R12` is set to `[SP+0]`, which is 0 and `R0` is set to `[SP+4]`, which is also 0. This causes the `CMP` instruction at line 7 to compare 0 with 0. As a result, the loop only runs once. The flags set by the `CMP` instruction are reused within the loop body to select transformations.

At a higher level, the effect is a such: First, the loop body performs *AddRoundKey* transformation, then it skips *SubBytes* and *ShiftRows* transformations because the flags set by `CMP` are reused; lastly, it performs *MixColumns* transformation. However, since the loop is not executed anymore, the resulting value of this transformation is never used again.

In the end, only the *AddRoundKey* transforms the output ciphertext. Therefore, in a known plaintext scenario, Equation (3) holds and enables

```
1  |b087|e9d0   // SUB   SP, 0x1c
                // LDRD  R7, R8,[R0,0x8]
2   7802|f8d2   // LDR.W R3, [R2, 0x801]
3   3801|9002|  // STR   R0, [SP, 0x8]
4  |2000|f102   // MOVS  R0, 0x0
                // ADD.W R4, R2, 0xb4
5   04b4|9000|  // STR   R0, [SP, 0x0]
6  |e9dd c000|  // LDRD  R12, R0, [SP]
7  |4560|da06|  // CMP   R0, R12
                // BGE.N <...>
```

Listing 15: Observed execution for Skip 32 bits / single instruction skip and single instruction corruption when targeting line 3 in Listing 14.

to recover the secret key.

$$\begin{aligned} ciphertext &= plaintext \oplus key \\ key &= ciphertext \oplus plaintext \end{aligned} \qquad (3)$$

Clock glitch campaigns on Arm Cortex-M3 and Cortex-M4 devices have confirmed the presence of this behavior beyond theoretical analysis. It should be mentioned that skip 64 bits fault model would also lead to the same described exploitation.

7.2 Early return on *TinyAES128*

This section illustrates a possible future attack vector with the described fault models, allowing for an Early Return from a function. An excerpt from *TinyAES128* containing the necessary instruction pattern is detailed here.

The new fault models enable the exploitation of consecutive branch instructions. In *TinyAES128*, the encryption function utilizes separate functions to perform various encryption transformations. The encryption function then calls these functions sequentially. Listing 16 highlights a part of the resulting instructions. When these calls use misaligned 32-bit `B` or `BL` instructions, the fault models allow attackers to turn branches into Early Returns, giving way to return from the encryption function.

After an Early Return, the memory for the resulting ciphertext contains an internal state used during encryption process. A similar attack to the one described in Section 7.1 can then be mounted.

```
1  ....|f7ff   // BL <SubBytes>
2  ffd1|f7ff   // BL <ShiftRows>
3  ff9d|....
```

Listing 16: Misaligned consecutive branch instructions within TinyAES128.

Thumb2 branch instructions use a *relative* instruction offset. The 32-bit branch instruction encoding stores the 12 least significant bits of this offset in the second-half (16 bits) of the encoding [41]. Because of these two points, applying the model described in Section 4.2.1 on the first of two consecutive misaligned 32-bit branch instructions effectively ignores the first branch and executes the second branch offset by −32 bits as shown in Listing 17.

Usually, 32 bits before a function is the `POP`/`return` instruction of the calling function. Since no `PUSH` instruction was given beforehand, the result is an Early Return from the calling function, which is the whole encryption function in our case.

Conditions apply on functions to witness this behavior: For example, the position of the function declarations, the similarity of function parameter types, and the usage of the link register. It is also important to consider that not all registers might be properly restored when returning. This is especially the case for the `BL` instruction.

```
1  ....|f7ff   // BL <ShiftRows-0x4>
2  ff9d|....
```

Listing 17: Observed execution for Skip 32 bits / single instruction skip and single instruction corruption when targeting line 2 in Listing 16.

7.3 Hint and control instructions in *MbedTLS-AES*

This section describes the usage of hint and control instructions to mimic instruction skips. As a result, it would simplify performing differential fault analysis attacks on this implementation of AES.

Thumb2 instruction set contains hint and miscellaneous control instructions [41]. They detail a requested internal behavior to the processor concerning memory usage (*e.g.,* `PLD`), system events (*e.g.,* `WFI`), speculative execution (*e.g.,* `CSDB`) and pipelining (*e.g.,* `ISB`). In most places, these instructions have no impact on execution and behave like a `NOP` within the execution flow. This makes them of special interest when exploiting the portrayed instruction corruption fault models.

Within the explored targets, instruction corruption only ever yields a `PLD` hint instruction. Within *TinyAES128*, the `PLD` instruction has undefined arguments and is therefore unusable. In *MbedTLS-AES*, both the `O1` and `O2` compiler optimization levels yield corrupted instructions to precise `PLD` instructions. Listing 18 and Listing 19 show how `LDRB.W` instruction has been corrupted to a `PLD` one.

```
1  ....|fa53   // UXTAB LR, R3, LR
2  fe8e|f89e   // LDRB.W R2, [LR, #40]
3  2028|69f6|  // LDR R6, [R6, #28]
4 |4072|....   // EORS R2, R6
```

Listing 18: Selection of MbedTLS-AES encryption instructions.

```
1  ....|fa53   // UXTAB LR, R3, LR
2  fe8e|f89e   // PLD    [LR, #3726]
3  fe8e|f89e   // LDRB.W R4, [LR, #114]
4  4072|....
```

Listing 19: Observed execution for Skip & repeat 32 bits / double instruction corruption when skipping line 3 and repeating line 2 in Listing 18 .

7.4 Further remarks

To summarize, our primary focus in this section was not on introducing novel attacks against AES. Instead, our objective was to showcase the practical utility of the proposed fault models. These models effectively identified *new* vulnerabilities in such AES implementations. Consequently, this empowers developers to opt for alternative implementation choices or explore adequate countermeasures. Additionally, it is noteworthy that developers commonly lean towards software countermeasures in response to fault attacks, like duplicating instructions. However, in the case we present here, such countermeasures would inadvertently broaden the attack surface, potentially simplifying the exploitation.

An important point to be noticed is that any exploitation using the described fault models is

Table 4: Number of possible fault insertions with 32-bit misaligned instruction corruptions within the encryption function and the number of created undefined instructions for the explored target codes at several optimization levels.

Target code	Optimization level	Inserted faults	Undefined instructions
BroAES	`-O0`	29	10 (34.5%)
	`-O1`	59	16 (27.1%)
	`-O2`	87	31 (35.6%)
	`-Os`	85	19 (22.4%)
TinyAES128	`-O0`	33	11 (33.3%)
	`-O1`	44	21 (47.7%)
	`-O2`	106	17 (16.0%)
	`-Os`	52	26 (50.0%)
MBedTLS-AES	`-O0`	69	35 (50.7%)
	`-O1`	293	84 (28.7%)
	`-O2`	283	60 (21.2%)
	`-Os`	193	52 (26.9%)

extremely dependent on minor choices made by the compiler and linker. The optimization levels, positions of functions and chosen instructions encoding play a major role in deciding whether these fault models would produce exploitable behaviors. Furthermore, many applications of the misaligned instruction corruption fault models lead to *undefined* instructions which would cause a crash or a fault handler to trigger. An indication of the frequency of these undefined instructions for the different targets of AES is reported in Table 4.

Other applications can create *unpredictable* instructions which may behave differently across various architectures and activate fault handlers too.

Even after considering all these comments, every target and optimization level still creates numerous possible injection points. Therefore, it is common to find multiple injection points that execute without crashing and where changes to the output or control flow of the program can be witnessed. Whether these injection points are fully exploitable depends on the target program and target architecture.

8 Conclusion and Perspectives

In this article, we have exposed how the observed faulty behaviors at ISA level could dramatically change depending on the code alignment in memory; this is due to Thumb2 instruction set supporting variable-length instructions, which can lead to aligned or misaligned code in memory. We have also shown that all these behaviors could be fully explained at binary encoding level with the two fault models: Skip and Skip & repeat. The provided detailed description at ISA level can now clear up many faulty behaviors mentioned in the literature. Also, the obtained results could be generalized to other cases of memory access size than 32 or 64 bits present in this work. In addition, RTL fault simulation experiments have been depicted and reveal the origin of such erroneous behaviors, validating the inferred fault models. Finally, examples have been produced on how these behaviors may be exploited in various security contexts. Actual clock glitch fault injection experiments confirm all these statements.

The portrayed cross-layer analysis will help to design countermeasures to such faults at lower levels of abstraction, *i.e.,* RTL and/or microarchitecture. It could also be helpful at software level, where countermeasures are usually less expensive to implement.

In terms of perspectives, looking for countermeasures against different faulty behaviors while keeping the best performance possible will be very important and necessary. These countermeasures could be investigated both at software and hardware levels. At software level, this may include: adding dummy instructions to avoid aligned or misaligned vulnerable instruction, or replacing operand register with another, if specific register could lead to vulnerable encoding. At hardware level, duplicate the vulnerable registers, adding parallel delay lines, as glitch detectors, to the input of the vulnerable flip-flops, or even splitting the critical paths might be possible solutions. Moreover, the effect of such countermeasures in terms of cost, performance and effectiveness should be studied carefully. As an additional perspective, this work opens the door to research on the effect of fault injection on various architectures, keeping in mind that the supported ISA can provide variable-length instructions. Finally, (re)engineering novel compressed instruction sets could come up, designed to be immune to such vulnerabilities, even in the presence of faults.

Acknowledgments

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

[2] Baumann, R.C.: Radiation-induced soft errors in advanced semiconductor technologies. IEEE Transactions on Device and Materials Reliability **5**(3), 305–316 (2005)

[3] 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. IEEE, McLean, United States (2019)

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

[5] 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)

[6] Proy, J., Heydemann, K., Berzati, A., Majéric, F., Cohen, A.: 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, pp. 7–1710. ACM

[7] Timmers, N., Spruyt, A., Witteman, M.: Controlling PC on ARM using fault injection. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 25–35 (2016)

[8] 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)

[9] Alshaer, I., Colombier, B., Deleuze, C., Beroulle, V., Maistri, P.: 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), pp. 1–6 (2021)

[10] Skorobogatov, S.: Local heating attacks on flash memory devices. In: 2009 IEEE International Workshop on Hardware-Oriented Security and Trust, pp. 1–6 (2009)

[11] Alshaer, I., Colombier, B., Deleuze, C., Maistri, P., Beroulle, V.: Cross-layer inference methodology for microarchitecture-aware fault models. Microelectronics Reliability **139**, 114841 (2022)

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

[13] Trouchkine, T., Bouffard, G., Clédière, J.: 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), pp. 31–38 (2021). IEEE

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

[15] Gratchoff, J., Timmers, N., Spruyt, A.,

Chmielewski, L.: Proving the wild jungle jump. Technical report, Technical report, University of Amsterdam (2015)

[16] Alshaer, I., Colombier, B., Deleuze, C., Beroulle, V., Maistri, P.: Variable-length instruction set: Feature or bug? In: 25th Euromicro Conference on Digital System Design, DSD 2022, Maspalomas, Spain, August 31 - Sept. 2, 2022, pp. 464–471. IEEE

[17] Pan, H.: High performance, variable-length instruction encodings. PhD thesis, Massachusetts Institute of Technology (2002)

[18] 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]

[19] MIPS Technologies, Inc.: microMIPSTM Instruction Set Architecture Uncompromised Performance, Minimum System Cost . [Accessed: March 2, 2022]. https://s3-eu-west-1.amazonaws.com/downloads-mips/mips-documentation/login-required/micromips_instruction_set_architecture.pdf

[20] Waterman, A., Lee, Y., Patterson, D.A., Asanović, K.: The RISC-V compressed instruction set manual, version 1.7. EECS Department, University of California, Berkeley, UCB/EECS-2015-157 (2015)

[21] informIT: Understanding ARM Architectures. [Accessed: March 1, 2022]. https://www.informit.com/articles/article.aspx?p=1620207&seqNum=3

[22] Tom Shanley — Mindshare, Inc.: x86 Instruction Set Architecture. [Accessed: March 2, 2022]. https://www.mindshare.com/files/ebooks/x86%20Instruction%20Set%20Architecture.pdf

[23] MIPS Technologies, Inc.: MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set . [Accessed: March 2, 2022]. https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol2.pdf

[24] MIPS Technologies, Inc.: MIPS64™ Architecture For Programmers Volume II: The MIPS64™ Instruction Set. [Accessed: March 2, 2022]. https://scc.ustc.edu.cn/zlsc/lxwycj/200910/W020100308600769158777.pdf

[25] ARM Limited: ARM Architecture Reference Manual Thumb-2 Supplement. [Accessed: February 22, 2022]. https://developer.arm.com/documentation/ddi0308/d

[26] Intel Corporation: Intel®64 and IA-32 Architectures Software Developer Manuals, Volume 3A: System Programming Guide, Part 1. Intel Corporation, Santa Clara, CA (2016)

[27] Advanced Micro Devices, Inc.: AMD64 Architecture Programmer's Manual Volumes 1–5. Advanced Micro Devices, Inc., Santa Clara, CA (2023). https://www.amd.com/system/files/TechDocs/40332.pdf

[28] Harris, S.L., Harris, D.M.: 3 - sequential logic design. In: Harris, S.L., Harris, D.M. (eds.) Digital Design and Computer Architecture, pp. 108–171. Morgan Kaufmann, Boston (2016)

[29] Ankit Mahajan: Relation between clock skew and frequency of operation. https://www.linkedin.com/pulse/relation-between-skew-frequency-operation-ankit-mahajan/. [Accessed: July 3, 2022]

[30] Texas Instruments: Basics of SPI: Timing Requirements and Switching Characteristics. [Accessed: July 3, 2022]. https://training.ti.com/sites/default/files/docs/adcs-spi-communications-timing-presentation.pdf

[31] Markovic, D., Nikolic, B., Brodersen, R.: Analysis and design of low-energy flip-flops. In: Proceedings of the 2001 International Symposium on Low Power Electronics and Design, pp. 52–55 (2001)

[32] O'Flynn, C., Chen, Z.D.: Chipwhisperer: An open-source platform for hardware embedded

security research. In: Prouff, E. (ed.) International Workshop on Constructive Side-Channel Analysis and Secure Design. Lecture Notes in Computer Science, vol. 8622, pp. 243–260. Springer, Paris, France (2014)

[33] Zussa, L., Dutertre, J.-M., Clédière, J., Robisson, B., Tria, A.: Investigation of timing constraints violation as a fault injection means. In: 27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France, p. (2012)

[34] Zussa, L., Dutertre, J.-M., Clédière, J., Tria, A.: Power supply glitch induced faults on fpga: An in-depth analysis of the injection mechanism. In: 2013 IEEE 19th International On-Line Testing Symposium (IOLTS), pp. 110–115 (2013)

[35] Selmane, N., Bhasin, S., Guilley, S., Danger, J.: Security evaluation of application-specific integrated circuits and field programmable gate arrays against setup time violation attacks. IET Inf. Secur. **5**(4), 181–190 (2011)

[36] Bayon, P., Bossuet, L., Aubert, A., Fischer, V., Poucheret, F., Robisson, B., Maurine, P.: Contactless electromagnetic active attack on ring oscillator based true random number generator. In: Schindler, W., Huss, S.A. (eds.) Constructive Side-Channel Analysis and Secure Design, pp. 151–166 (2012)

[37] Ghodrati, M., Yuce, B., Gujar, S., Deshpande, C., Nazhandali, L., Schaumont, P.: Inducing local timing fault through EM injection. In: Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018, pp. 142–11426. ACM

[38] Tang, A., Sethumadhavan, S., Stolfo, S.J.: CLKSCREW: exposing the perils of security-oblivious energy management. In: Kirda, E., Ristenpart, T. (eds.) 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017, pp. 1057–1074. USENIX Association

[39] Murdock, K., Oswald, D.F., Garcia, F.D., Bulck, J.V., Piessens, F., Gruss, D.: Plundervolt: How a little bit of undervolting can create a lot of trouble. IEEE Secur. Priv. **18**(5), 28–37 (2020)

[40] Qiu, P., Wang, D., Lyu, Y., Tian, R., Wang, C., Qu, G.: Voltjockey: A new dynamic voltage scaling-based fault injection attack on intel SGX. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **40**(6), 1130–1143 (2021)

[41] ARM Limited: ARMv7-M Architecture Reference Manual. [Accessed: February 22, 2022]. https://developer.arm.com/documentation/ddi0403/latest

[42] Agoyan, M., Dutertre, J.-M., Naccache, D., Robisson, B., Tria, A.: When clocks fail: On critical paths and clock faults. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) Smart Card Research and Advanced Application, pp. 182–193. Springer, Berlin, Heidelberg (2010)

[43] Li, Y., Ohta, K., Sakiyama, K.: New fault-based side-channel attack using fault sensitivity. IEEE Transactions on Information Forensics and Security **7**(1), 88–97 (2012)

[44] Alshaer, I., Colombier, B., Deleuze, C., Beroulle, V., Maistri, P.: Microarchitectural Insights into Unexplained Behaviors under Clock Glitch Fault Injection. In: Springer (ed.) 22nd Smart Card Research and Advanced Application Conference (CARDIS 2023). Lecture Notes in Computer Science (LNCS), pp. 1–20. Springer, Amsterdam, Netherlands (2023). https://hal.science/hal-04273995

[45] Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In: Fischer, W., Schmidt, J. (eds.) Workshop on Fault Diagnosis and Tolerance in Cryptography3, pp. 77–88. IEEE Computer Society, Los Alamitos, CA, USA (2013)

[46] Spensky, C., Machiry, A., Burow, N., Okhravi, H., Housley, R., Gu, Z., Jamjoom,

H., Kruegel, C., Vigna, G.: Glitching demystified: analyzing control-flow-based glitching attacks and defenses. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 400–412 (2021). IEEE

[47] Benso, A., Di Carlo, S., Di Natale, G., Prinetto, P.: Static analysis of seu effects on software applications. In: Proceedings. International Test Conference, pp. 500–508 (2002)

[48] Escouteloup, M., Lashermes, R., Lanet, J.-L., Fournier, J.J.-A.: Recommendations for a radically secure ISA. In: CARRV 2020 - Workshop on Computer Architecture Research with RISC-V, pp. 1–22. ACM, Valence (virtual), Spain (2020). https://hal.archives-ouvertes.fr/hal-03128242

[49] Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Transactions on Information and System Security (TISSEC) **15**(1), 1–34 (2012)

[50] Daemen, J., Rijmen, V.: Rijndael for AES. In: The Third Advanced Encryption Standard Candidate Conference, pp. 343–348. National Institute of Standards and Technology,, New York, USA (2000)