Microarchitectural Insights into Unexplained Behaviors under Clock Glitch Fault Injection

Ihab Alshaer[1,2][0000−0002−1374−3757], Brice Colombier[3], Christophe Deleuze[1], Vincent Beroulle[1], and Paolo Maistri[2]

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

Abstract. With the widespread use of embedded system devices, hardware designers and software developers started paying more attention to security issues in order to protect these devices from potential threats. Physical attacks represent an important threat to these devices, and fault injection is one of the major physical attacks. However, misunderstanding the effects of the fault injection would lead to proposing either over-protections or under-protections for these devices, thus affecting the performance/cost ratio and/or the security of the device. In this article, we provide a better representation of occurring fault, as a result of clock glitch, through novel models, in order to better understand the effects of fault injection. Also, we examine their dependencies with respect to the target device and the target program. Finally, we make use of the presented fault models to break the control-flow integrity of a program by altering the value of the program counter, in order to provide an actual application example.

Keywords: Fault injection attacks · Clock glitch · Fault model.

1 Introduction

Given how frequently embedded systems are used in various spheres of life, securing them from malicious activities is fundamental. Sensitive data in embedded systems can be efficiently protected using cryptographic algorithms, which are frequently implemented in software on embedded microprocessors. However, such solutions can be vulnerable to attacks that seek to gain access to this private data. In particular, they might be vulnerable to physical attacks.

Fault injection is a major and powerful active physical attack. Since the well-known Boneh *et al.* attack [7], where the authors were able to break an implementation of CRT-RSA by inducing faults into the computations, it has been an attractive research topic.

It is possible to perform the fault injection in a variety of ways: by exposing a digital device to radiations [6], laser beams [11], or an electromagnetic pulse [13], by causing perturbations in the power supply [20] or in the clock signal [2], by altering the environment's temperature [18], *etc.*

1.1 Fault Injection Effects

Several works [10,13,15,22,19,8] claimed that the effect of a fault injection or the success of a fault injection attack is somehow random. In some cases [10,19,8], the corruption is expressed as random bit flips or random byte faults. On the other hand, other works [13,15,22], described it as random data corruptions that are applied at the instruction set architecture (ISA) level, either on the instruction data or on the contents of the registers.

Based on such variety, analyzing the vulnerabilities that fault injection can exploit is extremely challenging, and thus, developing countermeasures is significantly more complex. This will definitely result in either over-protections, which will affect the performance and the cost of the device, or conversely, in under-protections too, which will affect the security of the device.

Recent studies [12,21,3,11,2] tried to explain the effects of the fault injection by looking at the lower levels of abstraction of digital systems. In particular, they focus on the register transfer level (RTL), microarchitectural level and/or binary encoding level of the instructions. In some cases [12,21], however, the authors only conducted simulations at ISA and RTL levels: they did not confirm the realism of their analysis with physical fault injections. In contrast, [3,11] performed physical fault injections, but they only focused on two kinds of faulty behaviors: complete-instructions skip and complete-instructions replay faults. In [2], authors offered a thorough analysis and justification of the experimental findings that show how the alignment of the instructions in the flash memory can affect the obtained faulty behaviors. Based on that, they proposed two fault models at the binary encoding level: *Skip* a specific number of bits and *Skip and Repeat* a specific number of bits, whose value is strictly related to the flash memory access size. However, in their work, they explicitly said that these two fault models explain many of the obtained faulty behaviors, but *not all* of them.

1.2 Contributions

In this article, we propose a new inferred fault model, the *partial update fault model*, that is applied to the binary encoding of the instructions. This new fault model aims at explaining different faulty behaviors that are obtained when performing clock glitch fault injection campaigns on a 32-bit microcontroller. Therefore, it improves the vulnerability analysis process against fault injection, and hence, allows developers to design cost-effective countermeasures. We also show how a subcase of the new fault model is instruction-independent with high probability, and its manifestation is highly device-dependent. We show how we can execute new instructions even with a different length of encoding as a result of a clock glitch, by exploiting the variable-length capabilities of the target microcontroller. Finally, we make use of the presented fault model to modify the program counter to a chosen address, whose value is stored in a general-purpose register.

1.3 Outline

This article is organized as follows: Section 2 briefly describes the methodology we followed to explain the obtained results. Section 3 presents the inferred binary encoding fault models. Section 4 describes the experimental setup, then experimental results are reported and discussed in Section 5. Section 6 presents different practical scenarios to modify the program counter, based on the presented fault models. The article is concluded along with future perspectives in Section 7.

2 Fault model inference

The method we followed in this work to describe and characterize the fault injection results is comparable to the methods used in [9,3,12,13]. The core of the analysis consists in comparing the outcomes of executions, both the simulations and the actual fault injections, at various levels of digital system abstraction. In this work, we focus our analysis on two abstraction levels: ISA level and binary encoding of the instructions. We also enrich these descriptions by providing insights at the microarchitectural level.

On one side, physical fault injections are performed, with appropriate injection parameters, on a target device that is executing a simple target program, which is formed of a sequence of assembly instructions (step ① in Figure 1). Then, from the physical fault injection results, we infer fault models at the binary encoding level of the instructions (step ② in Figure 1). Applying the inferred binary fault models to the simulated execution of the same target program, that was used in step ①, is the next step (step ③ in Figure 1). The outcomes of the physical fault injection and the software execution are then compared in order to provide better characterization of the impact of the fault injection and validate the inferred fault models (step ④ in Figure 1). The comparison is carried out on the output values of the processor's general-purpose registers. Each of these registers has a known value at the beginning, and any change can be detected after performing step ④ in Figure 1.

3 Partial update fault model

This section presents the inferred binary encoding fault models that are applied to the target programs. These fault models seek to explain the faulty behaviors that have been observed after physical fault injection campaigns have been carried out on the target device that is running these target programs.

It has been observed through these physical fault injection experiments that not all of the observed faulty behaviors can be explained by the binary encoding fault models described in [2]. There are in fact other faulty behaviors, which can be explained with the new fault models described in this section.

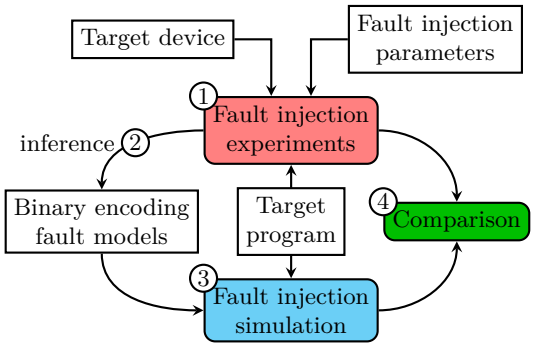


Fig. 1: Fault model inference methodology.

3.1 Partial update from the precharge value

This fault model corresponds to a fault that happens while the fetched data or instruction is propagated between internal registers from the flash memory to the core, as shown in Figure 2.

The hypothesis behind this fault model is based on the fact that not all bits of the data are propagated at the same speed from an internal register to another through a bus or combinational logic. Consequently, not all flip-flops within the destination register will get the update at the same time at a rising edge of a new clock cycle.

In nominal conditions, the clock period is defined such that all signals can be correctly sampled (i.e., the critical path has a positive slack). In case of a clock glitch, however, this behavior is disrupted by the fact the clock edge occurs quite sooner than expected. Thus, with the suitable injection parameters, it may happen that some flip-flops will receive the correct update, while some will receive the precharge value of the bus. Assuming that the precharge value of a bus or a wire between two registers is zero, then the correct update of a flip-flop means receiving the correct logic one or zero, while not receiving the correct update means capturing the precharge value of the bus, *i.e.* zero.

This model is observed as a reset on some bits while the instructions are transferred through the fetch data path in Figure 2, as shown experimentally in subsection 5.1.

It should be mentioned that in [13], the authors claimed that some of the observed faults, as a result of electromagnetic fault injection, might be related

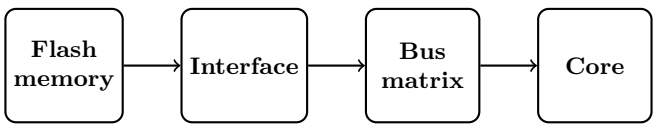


Fig. 2: Fetch data path in a microcontroller.

to the precharge value of the target microcontroller's bus. However, they didn't have a clear model that could explain their observed faults.

3.2 Partial update from the previous value

This fault model is somehow similar to the previous one. It occurs on the same path shown in Figure 2. However, instead of receiving the precharge value from the bus, some flip-flops within a destination register will keep their old values, either because the values have not been changed or because the corresponding wire still keeps the old values, and hence, the updated values are similar to the old ones. Conversely, and at the same time, other flip-flops in the destination register will receive the correct updated value.

This model is formally described as a bitwise OR between the old value and the new value of an internal register. This merge might be a full merge or a partial merge, as shown experimentally in subsection 5.2. Thus, in each flip-flop, the resulting value can be either the previous value or the correct value *i.e.* the value that the flip-flop should receive under normal execution, without any fault injection.

When looking at the instructions execution in this case, we observe the following behavior. The instruction(s) fetched at clock cycle i is executed normally. However, the instruction(s) fetched at clock cycle $i+1$ is not the one being executed. Instead, the observed instruction(s) is a full or partial merge between the fetched data at clock cycle i and the fetched data at clock cycle $i+1$. More details about this behavior are provided in subsection 5.2.

3.3 Discussion

Exploiting the transition value of a wire or a bus from a precharge (or a previous) value to a new value is a well-established modelling approach in power analysis attacks [1,16,17], which employ the so-called Hamming weight (or distance) leakage model. Likewise, our approach shows a similar pattern: depending on the type of register transition occurring (from previous or precharge value), the corresponding partial update fault model applies.

In section 5, we show that both cases can occur for the same device. This is not in contrast with our modeling, as depending on the actual element that is affected by the fault injection (in our case, the clock glitch) and the fine-tuning of the injection parameters, we may see different outcomes. Further details ahead.

4 Experimental setup

The target device and the fault injection method we employed are presented in this section. Section 5 includes the target programs, the corresponding experimental outcomes, and the discussion that follows.

4.1 Clock glitch fault injection

An effective method of fault injection is to introduce perturbations on the clock signal. Compared with other fault injection methods like laser or electromagnetic pulses, clock glitch is known to be effective and the least expensive method. Also, it can offer a respectable level of controllability thanks to its temporal accuracy and, consequently, the location of the injection within the target program. In clock glitch fault injection, the glitch interferes with the normal operation of the clock signal, possibly causing a timing violation that results in a variety of erroneous behaviors. Additionally, since the glitch is introduced into the global clock, it is uncertain which microarchitectural component might be affected by the fault injection.

The following settings, illustrated in Figure 3, are tuned when performing clock glitch fault injection:

- Delay: the time between the rising edge of a trigger signal 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.

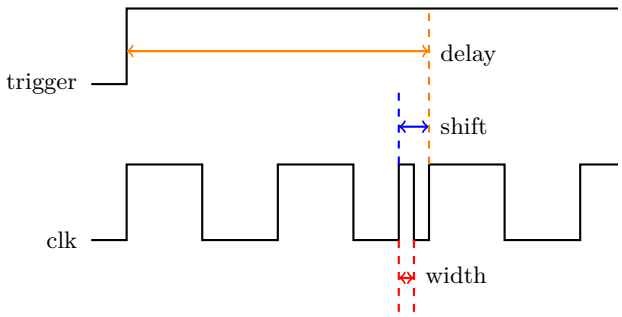


Fig. 3: Clock glitch parameters

In this work, the clock glitch fault injection campaigns have been carried out using the ChipWhisperer environment [14].

4.2 Target device

The target device is a 32-bit microcontroller that embeds an Arm Cortex-M4 processor. The Arm Cortex-M4 has a 3-stage pipeline: fetch, decode and execute. It has 13 general-purpose 32-bit registers, `R0` to `R12`. Arm Cortex-M4 is based on ARMv7-M architecture [5] and supports the Thumb-2 instruction set [4].

Thumb-2 is a variable-length instruction set that offers two encoding lengths: 16 and 32 bits. The instruction has a 32-bit encoding if the most significant five

bits of a 32-bit word have one of the following values [5]: `0b11101`, `0b11110` or `0b11111`.

The flash memory access size in this microcontroller is 64 bits. Therefore, up to two 32-bit or four 16-bit instructions can be fetched simultaneously. Additionally, as the supported instruction set is a variable-length instruction set, misaligned instructions can be fetched in several configurations as described in [2]. For example, the first half of a 32-bit instruction may be fetched at a given clock cycle, while the second half is fetched at the next clock cycle.

In the experiments, the processor is put in a known state before each fault injection. This is done by initialization instructions, that are located before the target instructions. After each execution, the values of the general purpose registers are transferred to a control computer via a serial communication in order to analyze the results. The target programs are presented in the next section.

5 Experimental results

The result of any fault injection experiment is assigned to one of these classes:

- Crash: we obtain a crash, reset, or failure when attempting to read the target final state via the serial communication,
- Silent: the final state of the target is the so-called golden state, *i.e.* as if no fault was injected,
- Fault: the final state of the target is different from the golden state.

The results of the different clock glitch fault injection campaigns are discussed separately with respect to each fault model in the following subsections.

5.1 Partial update from the precharge value

This section demonstrates how the *partial update from the precharge value* fault model explains many of the faulty behaviors observed during the fault injection campaigns. Also, it demonstrates the relation between this fault model and both the target instruction and the target device. To put it another way, it determines whether some bits in the fetched data are more sensitive to this fault model than other bits and, if so, whether the target instruction or the target device is to blame. The following subsections provide detailed results when targeting different instructions, and also when targeting a new device, identical to the already used one.

High-Hamming weight instruction Since the *partial update from the precharge value* fault model causes some bits of the target instruction to be reset, it makes sense to choose an instruction with a large Hamming Weight in order to maximize the occurrence of the considered fault model. Under this assumption, we chose the instruction `SUBS R6, 0xff`, whose encoding in Thumb-2 is `0x3eff`. Our rationale is twofold: the instruction has a comparatively large Hamming

Weight (13) given its size. Secondly, since the *partial update from the precharge value* fault model causes some bits of the instruction to be reset, applying it on `0x3eff` results in an instruction that can be discriminated with high probability.

The objective behind these experiments is to show that several faulty behaviors can be explained using the *partial update from the precharge value* fault model. In addition, we want to see if some bits are more vulnerable than others to be reset within a target instruction. Finally, were it the case, we need to know if this is because of the target instruction or of the target device. Since the fetch size in the target device is 64 bits, a 16-bit instruction may reside in any of four different positions within these 64 bits. Therefore, four injection campaigns have been performed, where the position of `0x3eff` is different from one campaign to another. The main reason of changing the position of the target instruction is to find out if the fault model depends on the target instruction, or it depends on its position within the fetched 64 bits, and hence, depends on the physical implementation of the target device. The remaining three positions are filled with three instructions with the encoding `0x0000`, in order to minimize possible side effects from other instructions and make the analysis easier. This encoding corresponds to the `MOVS R0, R0` instruction, which is equivalent to a `NOP`.

Table 1 gives the target part code of each fault injection campaign. It also shows the glitch parameters that are used. These parameters are chosen in order to maximize the number of faults that can be classified under the *partial update from the precharge value* fault model. Position refers to the location of `0x3eff` within the fetched 64 bits. The values of shift and width are provided as a percentage of a single clock cycle: the glitch is introduced before the rising edge of the target clock cycle if shift is negative. ChipWhisperer provides an additional parameter, called fine-width, which is used to offer fine-tuning of the width parameter. It has been noticed that fine-width provides better reproducibility of the results when it is used. Repetitions is the number of executions for each combination of parameters. For each fault injection campaign, the total number of experiments corresponds therefore to more than 20 000 executions, as summarized in the last row of the table. The same value of delay is used in all the campaigns, and it depends on the number of initialization instructions that precede the target part.

The results of the four injection campaigns on `0x3eff` with respect to the three classes (*i.e.,* Crash, Silent and Fault) are presented in Table 2. All the resulting faulty behaviors can be classified under two fault models: *Skip* (all the general purpose registers keep their initial values), or *partial update from the precharge value* . Table 2 also provides the number of observed behaviors linked to each fault model among the faulty executions.

Figure 4 shows the encoding of the **executed** instructions for each injection campaign, along with the number of times each of them is observed. All of these faulty behaviors are classified under the *partial update from the precharge value* fault model. This is because all of them can be seen as a reset on some bits of the original instruction `0x3eff`. It should be noticed that resetting all

Table 1: Experimental parameters

Position	1st	2nd	3rd	4th

Target part code	`0x3eff 0x0000 0x0000 0x0000`
	`0x0000 0x3eff 0x0000 0x0000`
	`0x0000 0x0000 0x3eff 0x0000`
	`0x0000 0x0000 0x0000 0x3eff`			

Shift	-13			
Width	{6, 10}	{6, 10}	{3, 4}	{3, 4}
Fine width	[-255, 255]			
Repetitions	20			
Total	20440			

Table 2: Fault obtained when targetting the `0x3eff` instruction at four different positions

Class	Position 1st	2nd	3rd	4th
Crash	0	0	23	1
Silent	33	1574	2273	158
Fault	20 407	18 866	18 144	20 281
Skip	11 523	8295	11 107	7901
Partial update from the precharge value	8884	10 571	7037	12 380

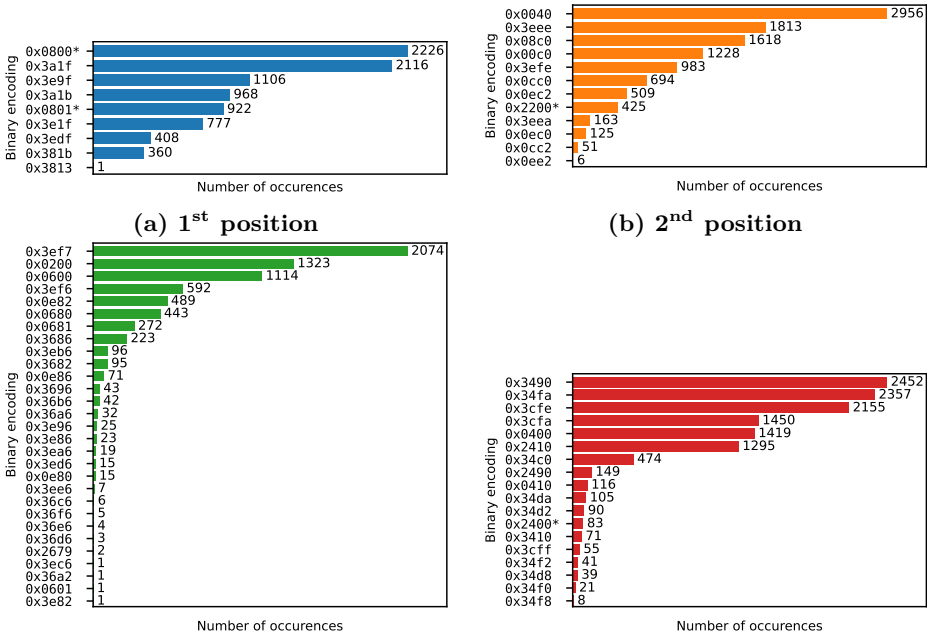
the bits of `0x3eff` will result in executing `0x0000`, which is an instruction with no effect as mentioned earlier, and thus classified under the skip fault model.

Additionally, Table 2 and Figure 4 show that the number of faulty behaviors and the observed executed instructions depend on the position of the instruction in the fetched 64 bits. Thus, the results depend on the position rather than the instruction. Furthermore, the results of each position show that some instructions are more probable to be executed than others as a result of the fault injection.

To better understand the effect of the fault at each position, and hence, on each bit in the position, we define a metric called *bit sensitivity*. It measures the probability for a bit to be reset as a result of the clock glitch fault injection over the faulty behaviors that are classified under the *partial update from the precharge value* fault model at a specific position. The *bit sensitivity* $S_p(f, b)$ of bit b to a given fault model f at position p is defined in Equation (1).

$$S_p(f, b) = 1 - \frac{P(b = 1 \mid p)}{P(\text{fault model} = f \mid p)} \tag{1}$$

Figure 5 presents the bit sensitivity values for the results obtained during the fault injection campaigns on `0x3eff` at all positions. Obviously, bits that are zero



(a) 1st position **(b) 2nd position**

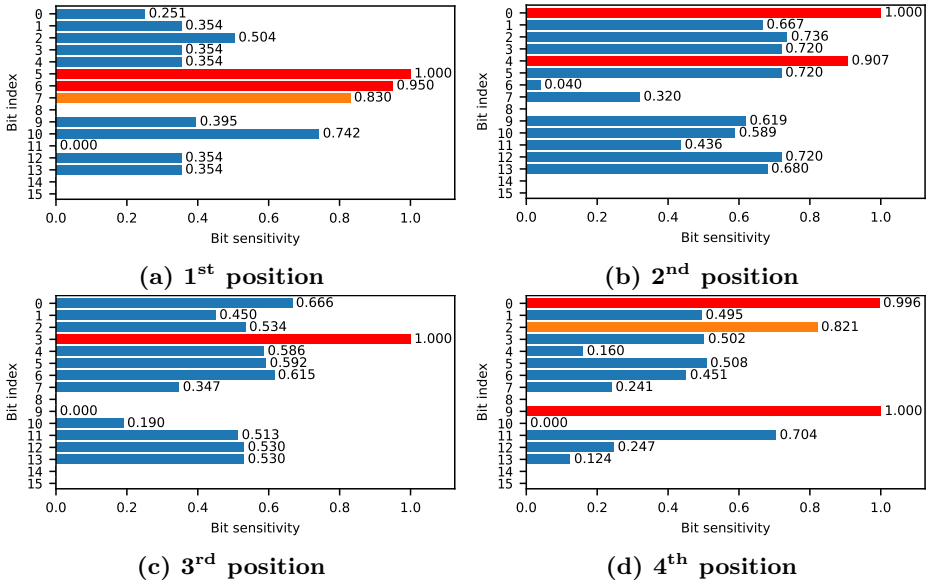
(c) 3rd position **(d) 4th position**

Fig. 4: Encoding of the observed executed instructions when targeting `0x3eff` at four different positions within the target programs.

in `0x3eff` (bits 8, 14, and 15) have no corresponding bit sensitivity value. We can see that the bit sensitivity is different from one position to another and from one bit to another at the same position. Thus, under the *partial update from the precharge value* fault model, some instructions are more probable than others.

Subsection 5.1 presents the results of targeting a different instruction, to confirm that the *partial update from the precharge value* fault model depends on the physical implementation of the device, and not on the target instruction.

It is important to note that whenever there is a doubt about the execution of an instruction, results are confirmed using alternative initial register values. Nonetheless, in rare circumstances, more than one instruction may produce the same outcome, for instance, when the value of a register is zero. For example, this might happen because of moving zero to the register, or by shifting its value by 32 bits. In Figure 4, when the encoding is starred, it means that there is an alternative instruction that could lead to the same outcome, and we selected one based on other observed encoding at the same position. It is important to stress that this is happening only in a few cases (4 times), and does not affect the measurements or the general conclusion.



(a) 1st position **(b) 2nd position**

(c) 3rd position **(d) 4th position**

Fig. 5: Bit sensitivity values obtained when targeting `0x3eff`.

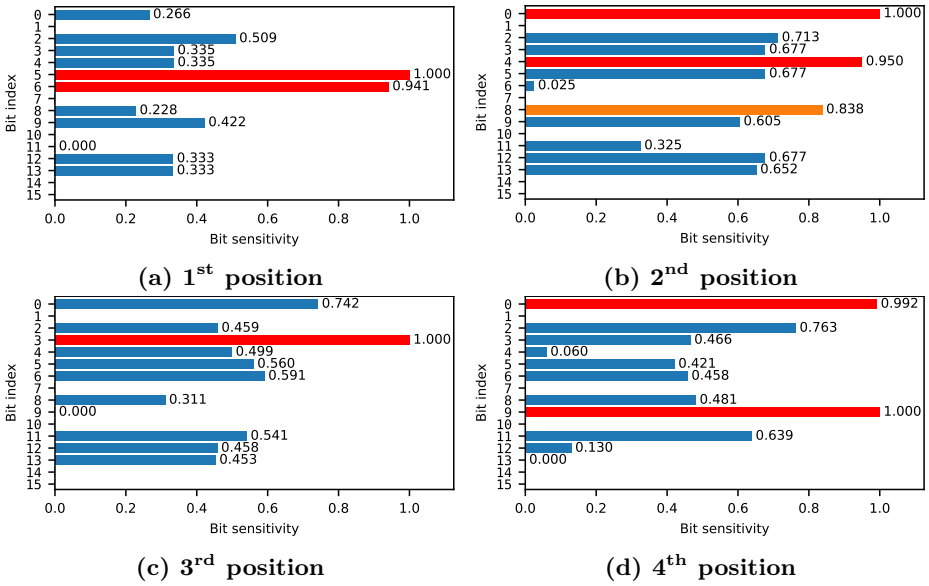
Confirming sensitive bits We carried out extra experiments with the value `0x3b7d`, which is the encoding of the `SUBS R3, 0x7d` instruction. Again, we chose this instruction since it has a relatively high Hamming weight, and allows recognizing the encoding of the executed instructions as a result of the *partial update from the precharge value* fault model with high probability. However, we specifically took care to have ones in the most sensitive positions from Figure 5 to see if these measurements are reproducible when targeting a different instruction.

The experimental parameters for the fault injection campaigns on `0x3b7d` are identical to that of `0x3eff`, given in Table 1. The only difference is that the target program has `0x3b7d` instead of `0x3eff`. The classification results are presented in Table 3, while the bit sensitivity values are plotted in Figure 6.

Table 3: Fault obtained when targeting the `0x37bd` instruction at four different positions

Class	Position			
	1st	2nd	3rd	4th
Crash	0	0	0	0
Silent	39	2304	2589	197
Fault	20 401	18 136	17 851	20 243
Skip	11 694	8386	10 528	7606
Partial update from the precharge value	8707	9750	7323	12 637

It is clear that the classification results and the bit sensitivity values of `0x3b7d` are very close to the corresponding results of `0x3eff`. This leads us to the conclusion that the *partial update from the precharge value* fault model is instruction-independent with high probability. If it depended on the instruction, then changing the position should not have an observable distinct effect on the executed instructions and on the bit sensitivity of different positions. On the other hand, the next subsection shows that bit sensitivity greatly depends on the target device.



(a) 1st position **(b) 2nd position**

(c) 3rd position **(d) 4th position**

Fig. 6: Bit sensitivity values obtained when targeting `0x3b7d`.

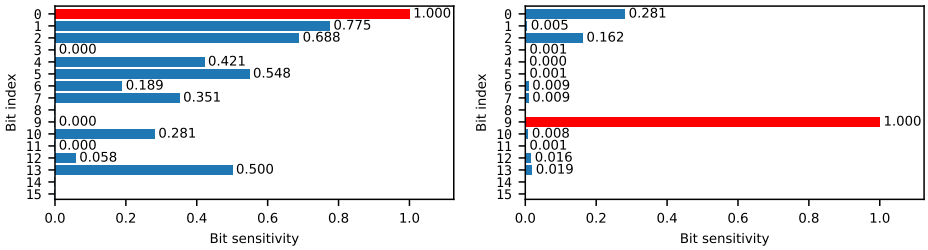
`0x3eff` experiments on a new microcontroller In this section, we present the results of targeting the `0x3eff` instruction again while using a brand new device, which we did not use to perform any experiment previously. In any other means, it is identical to the one we used in the previous experiments. This is done to better understand the dependency of the *partial update from the precharge value* fault model on the target device. The experimental parameters of this campaign are identical to those in Table 1.

For our purposes, it is enough to present the results on the 2nd and 4th positions to see that they are very different between the two devices. The results are presented in Table 4 and Figure 7.

A very interesting observation is that the number of faults and the bit sensitivity were much higher when performing the fault injection campaigns on the old device. This is clear for the bit sensitivity of the 4th position in Figure 7b, as

Table 4: Fault obtained when targeting the `0x3eff` instruction using the *new* device

Class	Position 2^{nd}	Position 4^{th}
Crash	0	2
Silent	18 058	16 995
Fault	2382	3443
Skip	345	0
Partial update from the precharge value	2037	3443



(a) 2^{nd} position **(b) 4^{th} position**

Fig. 7: Bit sensitivity values obtained when targeting `0x3eff` on the *new* device at the 2nd and the 4th positions.

we can see the distribution of the bit sensitivities is similar to that in Figures 5d and 6d, however, on the old device, the sensitivity is much higher. This could be explained as an aging effect, since the old device has been used for fault injection experiments for a few months. We speculate that the bit sensitivity could increase over time (a common consequence of performance degradation due to aging), but further research is needed to confirm this observation.

Conclusion on bit sensitivity To summarize, the bit sensitivity figures illustrate that, as a result of the *partial update from the precharge value* fault model, the probability distribution of the corrupted instruction is not random, and it depends on several features that are mostly device-dependent. The probability of executing a given instruction differs from the probability of executing another. This discrepancy is determined by both the instruction's position inside the target program and the target device itself. This is of prime importance if the instruction results in a security vulnerability, as will be highlighted in section 6.

5.2 Partial update from the previous value

In this section, we focus on the occurrence of faulty behaviors that can be classified under the *partial update from the previous value* fault model. In this case,

there is no precharge value and the transition occurs from the value that was previously stored in the register. In the time while the register is updating its value, a transient situation may occur when some bits already have the new value, whereas others are still to be updated. This behavior can be seen a merge between the previous and the new instruction. The following subsections give examples of observed faulty behaviors that are classified as full or partial merge.

Full merge The merge is considered full if and only if the observed executed instruction(s) can be expressed as a bitwise OR between the data fetched at clock cycle i and the data fetched at clock cycle $i + 1$. In the following, an example is provided, which illustrates how *two new* 32-bit instructions are executed as a result of a full merge between eight different 16-bit instructions.

Listing 1.1 shows the target program and the encoding of each instruction of this example. The observed execution as a result of the clock glitch fault injection is given in Listing 1.2. The bitwise OR of the first two hexadecimal digits at line 1 (`0xa9`) and the corresponding digits at line 5 (`0x42`) gives `0xeb`. Since the most significant five bits are `0b11101`, this word is decoded as a 32-bit instruction, as explained in subsection 4.2. The same holds for the merging of instructions at lines 3 and 7.

Listing 1.2 is obtained by a full merge applied on Listing 1.1 as follows:

- Merging the 32 bits at lines 1 and 2 with the 32 bits at lines 5 and 6 respectively: `0xa9000000 | 0x42000305 = 0xeb000305`.
- Merging the 32 bits at lines 3 and 4 with the 32 bits at lines 7 and 8 respectively: `0xa9000000 | 0x42020405 = 0xeb020405`.

```
1 ADD  R1, SP, 0x0    // 0xa900
2 MOVS R0, R0         // 0x0000
3 ADD  R1, SP, 0x0    // 0xa900
4 MOVS R0, R0         // 0x0000
5 TST  R0, R0         // 0x4200
6 LSLS R5, R0, 0xc    // 0x0305
7 TST  R2, R0         // 0x4202
8 LSLS R5, R0, 0x10   // 0x0405
```

Listing 1.1: Target program to execute two new 32-bit instructions as a result of full merge.

```
1 ADD  R1, SP, 0x0   // 0xa900
2 MOVS R0, R0        // 0x0000
3 ADD  R1, SP, 0x0   // 0xa900
4 MOVS R0, R0        // 0x0000
5 ADD  R3, R0, R5    // 0xeb000305
6 ADD  R4, R2, R5    // 0xeb020405
```

Listing 1.2: Observed execution as a result of full merge on Listing 1.1.

Partial merge In this case, only part of the fetched data at clock cycle i and the data fetched at clock cycle $i+1$ is merged. The target code that is used for this example is shown in Listing 1.3.

```
1 ADD  R1, R1, 0x4   // 0xf1010104
2 ANDS R2, R0        // 0x4002
3 MOVS R0, R0        // 0x0000
4 ADD  R2, R2, 0xa   // 0xf102020a
5 MOVS R4, R0        // 0x0004
6 MOVS R0, R0        // 0x0000
```

Listing 1.3: Target program for partial merge experiment.

One of the observed executions that can be classified under Partial merge is as the following: We observed that not all the 32 bits at lines 1 (`0xf1010104`) are systematically merged with the corresponding 32 bits at line 4 (`0xf102020a`): only the destination and source registers are merged. In addition to this behavior, another partial merge occurred in the following instructions, only over the least significant digit, between the 16 bits at line 2 (`0x4002`) and the 16 bits at line 5 (`0x0004`). As a consequence, only the destination register at line 5 is affected. The observed execution of this example is shown in Listing 1.4. It should be mentioned that we cannot discriminate on the opcode values (0xf1), as it is the same in both `ADD` instructions. It is worth mentioning that a Full merge was also observed for the target program in Listing 1.3

```
1 ADD  R1, R1, 0x4   // 0xf1010104
2 ANDS R2, R0        // 0x4002
3 MOVS R0, R0        // 0x0000
4 ADD  R3, R3, 0xa   // 0xf103030a
5 MOVS R6, R0        // 0x0006
6 MOVS R0, R0        // 0x0000
```

Listing 1.4: Observed execution as a result of partial merge after targeting Listing 1.3.

6 Program counter modification

In this section, we exploit the proposed fault models to change the value of the program counter to an address stored in one of the general purpose registers. Being able to modify the program counter allows to break the control flow integrity of a program. This is leveraged in various attacks, such as privilege escalation or secure-boot violation [20].

In the following subsections, we measure the probability of modifying the program counter under different scenarios for the target program in Listing 1.5. The results of the different scenarios, along with the fault models that led to the success of the attack, and the glitch parameters that allowed observing the

results are summarized in Table 5. The success rate is computed over 10 000 executions for each clock glitch fault injection scenario. The glitch parameters are tuned to maximize the success rate.

```
1  R8 = address of line 11
2  // series of 0x0000
3  ADD R6, R1, 0x4c7   // 0xf20146c7
4  ADD R3, R3, 0xa
5  ADD R4, R4, 0xb
6  ADD R5, R6, R3
7  ADD R3, R3, 0xf
8  // series of 0x0000
9  ADD R5, R5, 0x5
10 // series of 0x0000
11 ADD R1, R1, 0x3
12 ADD R9, R0, R6
```

Listing 1.5: Target program for PC modification experiments.

6.1 Misaligned code

In [2], the authors were able to modify the program counter to an address stored in `R8` as a result of the skip fault model in a misaligned code. This is done by executing the least significant 16 bits of a misaligned 32-bit instruction. The first half of the 32-bit instruction is fetched at clock cycle i and its second half is fetched at clock cycle $i+1$. Therefore, skipping the fetched data at clock cycle i results in decoding the remaining half that is fetched at clock cycle $i+1$, and executing when it is a valid encoding for a 16-bit instruction. The same thing can happen for the `ADD R6, R1, 0x4c7` instruction shown in Listing 1.5. Its least significant 16 bits (`0x46c7`) are the encoding of `MOV PC, R8`, which stores the value of `R8` into the program counter. Thus, executing `MOV PC, R8` leads to a jump from line 3 to line 11, since `R8` stores the address of line 11.

We reproduced their attack on Listing 1.5. Many useful and dummy instructions are used in Listing 1.5 to make sure of detecting the execution of `MOV PC, R8`. The success rate in this scenario was 100 %. We noticed that 9996 of the executions can be classified under the skip fault model. However, *four* executions can be classified under the *partial update from the precharge value* fault model. This is because resetting some bits of the most significant 16 bits of `ADD R6, R1, 0x4c7`, will lead to execute two 16-bit instructions, as the most significant five bits do not identify a valid encoding for a 32-bit instruction (as detailed in 4.2). For these four executions, we confirmed this is happening by observing the values of the registers that the `MOVS R1, R0` instruction, of encoding `0x0001`, had been executed. Thus, the instructions `MOVS R1, R0` and `MOV PC, R8` are executed in sequence.

6.2 Aligned code

The aforementioned attack relies on the misalignment of the code in memory, as explained in [2]. We add a single `MOVS R0, R0 (0x0000)` to the target program, just before `ADD R6, R1, 0x4c7` instruction, to realign it. The code is now aligned, all bits of `0xf20146c7` are fetched in a single clock cycle. In this case, the fault model that we can rely on to create new instructions (and thus modify the program counter) is the *partial update from the precharge value* fault model. The aim is to reset bits over the most significant 16 bits while not touching the least significant 16 bits, in order to keep the encoding of `MOV PC, R8`, *i.e.,* `0x46c7`. The success rate of the clock glitch fault injection campaign in this case was 0.71 %. However, no side effect is observed along with executing `MOV PC, R8`, but this is normal as resetting some bits of the most significant 16 bits of `0xf20146c7` could lead to execute many 16-bit instructions with no observable effect like `MOVS R0, R0 (0x0000)`, or `TST R0, R0 (0x4200)` for example.

This result is an improvement over the state of the art, since one could imagine that making the code aligned will protect from the misaligned faulty behaviors that are described in [2]. Thus, aligning the code cannot be considered a sufficient countermeasure against clock glitch attacks, that might focus on misaligned codes. However, aligning the sensitive instructions can effectively decrease the success rate, as demonstrated experimentally.

6.3 Countermeasure: register substitution

In this scenario the code is misaligned, but we changed the destination register in `ADD R6, R1, 0x4c7` from `R6` to `R2`. Other occurrences of `R6` are replaced with `R2` in the rest of the program. Now, the least significant 16-bit word for `ADD R2, R1, 0x4c7` is `0x42c7`. The success rate in this scenario was *zero*: no fault led to modify the program counter to the value in `R8`, even when we used the same experimental parameters that previously led to a success rate of 100 %. The `R2` register was chosen because `2` in the encoding can not be turned into a `6` by resetting bits. Thus, we avoid obtaining the encoding of `MOV PC, R8`.

This scenario shows that a clear understanding of the fault effect led to the design of a very simple and cost-effective countermeasure. This proposal clearly has no overhead and is easily implemented by the compiler, except in rare cases where registers might be under a lot of pressure.

6.4 Trojan

In this case, dummy code with no effect on the target program is added just before `ADD R2, R1, 0x4c7`, where the code is protected against executing `MOV PC, R8`. This dummy code is shown in Listing 1.6. It implements a Trojan that can be activated by clock glitch fault injection in order to controllably execute the `MOV PC, R8` instruction. It is clear that the *partial update from the previous value* fault model in the full merge setting will lead to execute `MOV PC,`

Table 5: Experimental results obtained, and fault injection parameters used when attempting to modify the program counter.

	Fault injection scenario			
	Misaligned	Aligned	Protected	Trojan
Success rate	100 %	0.71 %	0.0 %	95.11 %
Fault models	Skip [2] (99.96 %) *partial update from the precharge value* (sect. 5.1) (0.04 %)	*partial update from the precharge value* (sect. 5.1)	-	*partial update from the previous value* (sect. 5.2)
Shift	-12	-13	-	-9
Width	3	10	-	4

`R8`, since we have that `0x4281 | 0x0446 = 0x46c7 (MOV PC, R8)`. The experimental success rate of this scenario was 95.11 %.

This scenario is possible if we assume that the attacker is the software developer himself. Alternatively, the compiler used to compile the code may be untrusted and thus represent the attacker. As a countermeasure, a code review, based on the presented fault models, should be able to detect such Trojans.

```
1 CMP  R1, R0          // 0x4281
2 MOVS R0, R0          // 0x0000
3 MOVS R0, R0          // 0x0000
4 MOVS R0, R0          // 0x0000
5 LSLS R6, R0, 0x11    // 0x0446
6 MOVS R0, R0          // 0x0000
7 MOVS R0, R0          // 0x0000
8 MOVS R0, R0          // 0x0000
```

Listing 1.6: Dummy code implementing a Trojan.

7 Conclusion and future works

A new binary encoding fault model has been presented and defined: the partial update fault model, which comes in two variations: the *partial update from the precharge value* and the *partial update from the previous value* fault models. These fault models allow explaining a wide range of the faulty behaviors that are obtained when performing clock glitch fault injection campaigns. Therefore, they can be used to perform vulnerability analysis of software codes against these fault attacks, and help in better designing efficient and low-cost countermeasures. We have also given an exploitation example: modifying the program counter can be achieved and explained through these fault models. Following that, we proposed a simple yet effective countermeasure against such vulnerability. We also examined the dependency of *partial update from the precharge value* fault model with respect to the target device and program.

In terms of future works, proper formalization of protections against the presented fault models will be very necessary. At software level, an automated framework made of vulnerability assessment followed by automatic code protection would greatly improve the security of the targeted application. At a lower level, several approaches might be envisioned at different abstraction levels, from ISA down to transistor-level. Also, targeting other architectures will be important to see if the proposed models can be generalized to various architectures.

Finally, although clock glitch has been used in this article to perform the fault injection, the presented results may be generalized to other fault injection techniques that rely on timing violations. This includes, for example, voltage glitch and electromagnetic fault injection.

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. Alioto, M., Poli, M., Rocchi, S.: Differential power analysis attacks to precharged buses: A general analysis for symmetric-key cryptographic algorithms. IEEE Transactions on Dependable and Secure Computing **7**(3), 226–239 (2010)
2. 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. pp. 464–471. IEEE, Maspalomas, Spain (Aug 2022)
3. Alshaer, I., Colombier, B., Deleuze, C., Maistri, P., Beroulle, V.: Cross-layer inference methodology for microarchitecture-aware fault models. Microelectronics Reliability **139**, 114841 (2022)
4. ARM Limited: ARM architecture reference manual Thumb-2 supplement. https://developer.arm.com/documentation/ddi0308/d, [Accessed: February 24, 2023]
5. ARM Limited: Armv7-m architecture reference manual. https://developer.arm.com/documentation/ddi0403/latest, [Accessed: February 24, 2023]
6. Baumann, R.: Radiation-induced soft errors in advanced semiconductor technologies. IEEE Transactions on Device and Materials Reliability **5**(3), 305–316 (2005)
7. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of eliminating errors in cryptographic computations. J. Cryptology **14**, 101–119 (2001)
8. Buhren, R., Jacob, H.N., Krachenfels, T., Seifert, J.: One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) ACM SIGSAC Conference on Computer and Communications Security. pp. 2875–2889. ACM, Virtual Event, Republic of Korea (Nov 2021)
9. Dureuil, L., Potet, M., de Choudens, P., Dumas, C., Clédière, J.: From code review to fault injection attacks: Filling the gap using fault model inference. In: International Conference on Smart Card Research and Advanced Applications. Lecture

Notes in Computer Science, vol. 9514, pp. 107–124. Springer, Bochum, Germany (Nov 2015)
10. Khelil, F., Hamdi, M., Guilley, S., Danger, J., Selmane, N.: Fault analysis attack on an FPGA AES implementation. In: Aggarwal, A., Badra, M., Massacci, F. (eds.) International Conference on New Technologies, Mobility and Security. pp. 1–5. IEEE, Tangier, Morocco (Nov 2008)
11. Khuat, V., Danger, J., Dutertre, J.: Laser fault injection in a 32-bit microcontroller: from the flash interface to the execution pipeline. In: Workshop on Fault Detection and Tolerance in Cryptography. pp. 74–85. IEEE, Milan, Italy (Sep 2021)
12. Laurent, J., Deleuze, C., Pebay-Peyroula, F., Beroulle, V.: Bridging the gap between RTL and software fault injection. ACM J. Emerg. Technol. Comput. Syst. **17**(3), 38:1–38:24 (2021)
13. 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.) 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013. pp. 77–88. IEEE Computer Society (2013)
14. 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 (Apr 2014)
15. 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: International Conference on Availability, Reliability and Security. pp. 7:1–7:10. ACM, Canterbury, UK (Aug 2019)
16. Randolph, M., Diehl, W.: Power side-channel attack analysis: A review of 20 years of study for the layman. Cryptography **4**(2), 15 (2020)
17. Shelton, M.A., Samwel, N., Batina, L., Regazzoni, F., Wagner, M., Yarom, Y.: Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In: Annual Network and Distributed System Security Symposium. The Internet Society, Virtual event (Feb 2021)
18. Skorobogatov, S.P.: Local heating attacks on flash memory devices. In: Tehranipoor, M., Plusquellic, J. (eds.) IEEE International Workshop on Hardware-Oriented Security and Trust. pp. 1–6. IEEE Computer Society, San Francisco, CA, USA (Jul 2009)
19. 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: IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 400–412. IEEE, Taipei, Taiwan (Jun 2021)
20. Timmers, N., Spruyt, A., Witteman, M.: Controlling PC on ARM using fault injection. In: Workshop on Fault Diagnosis and Tolerance in Cryptography. pp. 25–35. IEEE Computer Society, Santa Barbara, CA, USA (Aug 2016)
21. Tollec, S., Asavoae, M., Couroussé, D., Heydemann, K., Jan, M.: Exploration of fault effects on formal RISC-V microarchitecture models. In: Workshop on Fault Detection and Tolerance in Cryptography. pp. 73–83. IEEE, Virtual Event / Italy (Sep 2022)
22. Trouchkine, T., Bouffard, G., Clédière, J.: EM fault model characterization on SoCs: From different architectures to the same fault model. In: Workshop on Fault Detection and Tolerance in Cryptography. pp. 31–38. IEEE, Milan, Italy (Sep 2021)