

Cross-Layer Inference Methodology for Microarchitecture-aware Fault Models

Ihab Alshaer, Brice Colombier, Christophe Deleuze, Paolo Maistri, Vincent Beroulle

▶ **To cite this version:**

Ihab Alshaer, Brice Colombier, Christophe Deleuze, Paolo Maistri, Vincent Beroulle. Cross-Layer Inference Methodology for Microarchitecture-aware Fault Models. Microelectronics Reliability, 2022, 139, 10.1016/j.microrel.2022.114841 . hal-03848691

HAL Id: hal-03848691
https://hal.science/hal-03848691

Submitted on 10 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

Cross-Layer Inference Methodology for Microarchitecture-aware Fault Models

Ihab Alshaer[a,b], Brice Colombier[b], Christophe Deleuze[a], Paolo Maistri[b], Vincent Beroulle[a]

[a] *Univ. Grenoble Alpes, Grenoble INP[1], LCIS, Valence 26000, France*
[b] *Univ. Grenoble Alpes, CNRS, Grenoble INP[1], TIMA , Grenoble 38000, France*

Abstract

Fault injection attacks are considered one of the major threats to cyber-physical systems. The increasing complexity of embedded microprocessors involves complicated behaviours in presence of such attacks. Realistic fault models are required to study code vulnerabilities and be able to protect digital systems from these attacks. However, inferring fault models using only limited observations of faulty microprocessors is difficult. In this article, we present experiments that show the difficulty of characterizing and modelling the fault injection effects. From there, we propose a complete approach for fault analysis to build proper fault models at different abstraction levels, which will help in designing suitable countermeasures at reasonable cost. We also present a suite of experiments that works as a proof of concept to validate the proposed methodology.

Keywords:
hardware security, fault injection, microarchitecture, clock glitch fault injection, simulation, fault model.

1. Introduction

With the widespread use of embedded systems in different areas of life, protecting these systems against malicious use becomes crucial. Digital systems contain sensitive information that can be effectively protected through cryptographic algorithms, often implemented in software on an embedded microprocessor. Such implementations, however, might be vulnerable to attacks that aim at extracting this sensitive information.

The protection task should even have a high priority, as the attack techniques and equipment are always improving. Fault injection is one of these attack techniques. In the context of hardware security, it belongs to the class of active physical attacks, possibly non-invasive, where the attacker has physical access to the device or its environment. The attacker will try to change the normal behaviour of the device during a program execution by injecting one or more faults, then observing the erroneous behaviour. It became an attractive research topic since the well-known Boneh *et al.* attack [1], where they were able to break some cryptographic protocols by inducing faults into the computations. The fault injection can be performed in different ways [2]: exposing the device to radiations [3], laser beam [4, 5], intense light [6] or an electromagnetic (EM) pulse [7, 8, 9, 10], inducing variations in the power supply [11], injecting a glitch in the clock signal [12], changing the environmental conditions such as the temperature [13], *etc.* The resulting faults could reveal an interesting behaviour that could be further exploited as a vulnerability.

Securing components, such as microprocessors and microcontrollers, against fault attacks requires a thorough understanding of the faults: on the one hand, this means characterizing, studying, and analyzing the faults that could lead to exploitable code vulnerabilities. On the other hand, it also requires designing countermeasures at different levels, hardware and software, with an acceptable cost.

To build appropriate countermeasures, designers need realistic fault models that provide proper characterization of the fault effects. However, with the increasing complexity of microprocessors, fault effect characterization based on a single level of analysis, such as assembly level or Register-Transfer Level (RTL), is difficult and limits the understanding of the fault. As a consequence, fault model inference becomes a complex task: the models are a high-level approximation, sometimes unrealistic, and the development and evaluation of the countermeasures are not optimized.

For the sake of designing countermeasures, several research studies have been conducted based only on a single level of fault characterization, such as Instruction-Set Architecture (ISA) level in [14, 15] or RTL [16]. However, because of the incomplete fault model, this could lead to either under-engineer or over-engineer the protections. In the former case, a security threat may still be present and hence exploitable; in the latter, this means unnecessarily adding cost and possibly decreasing performance.

Other studies tried to propose analysis by performing fault injection using more than one technique, in order to have a better overview of the observed faulty behaviours. Moro *et al.* [7], for example, carried out EM injection campaigns on a microcontroller and compared the observed

[1] Institute of Engineering Univ. Grenoble Alpes

behaviours with the results given by software simulation based on software fault models. Dureuil *et al.* [9] tried to generalize fault models as a result of performing laser and EM injections on RAMs and Flash memories of smart cards. Then, they simulated faults at the application level in order to provide a so-called "vulnerability rate" for such faults. A similar approach has been followed by Werner *et al.* [5]: the authors carried out laser fault injection along with software fault simulation. However, they focused mostly on performing multi-fault attacks rather than proposing new or more thorough fault models. Additionally, other works, for example [17, 18], described the faulty behaviours that are resulting from physical injections as random corruptions or random bit-flips. However, they did not consider lower levels of abstraction, which could help in providing a correct explanation for the observed behaviours. In the previous works, the authors provide fault characterization at the software level, *i.e.* ISA and/or high level applications, benefiting from observed faulty behaviours produced by physical fault injections. Hence, they did not provide complete details of analyzing the fault at the microarchitectural level in order to show how the fault occurred internally. Therefore, they could not assess the realism of their fault models.

Finally, in [19], Laurent *et al.*, suggested that fault simulations using typical software fault models (such as instruction-skip and test-inversion) are no longer enough to characterize the observed faulty behaviours, in particular when targeting complex microprocessors that have a large number of internal elements, *i.e.* registers and combinational logic. In their work, they provided a comprehensive analysis to assess software fault models by performing RTL fault simulation on a RISC-V microprocessor [20]. However, physical fault injections were not performed to validate the realism of their proposed RTL fault models. Moreover, different architectures should be taken into account in order to generalize the assumptions of their work.

In this article, which is an extended version of our work in [21], we present additional experiments for different target boards. These experiments provide additional clues for illustrating the difficulty of characterizing fault effects resulting from physical injections. In particular, we show that some of the both new and already obtained faulty behaviours are strongly related to the microarchitecture of the target and the target program used in an experiment. Previous works usually focus on one single level at a time, and model the faulty behaviours at just one level. This work highlights the strong need of addressing several abstraction layers at the same time in order to fully understand the fault occurrence mechanisms: this is proved by the fact that minor differences in the code have largely different faulty behaviours when executed on different, though close, micro-architectures.

On the basis of this evidence, we are now more confident that the gap between previous studies can be addressed by the proposed methodology [21]. This can be done by providing a cross-layer analysis of code and microarchitectural vulnerabilities while performing fault injections and simulations at three distinct levels: hardware/physical, RTL, and software levels. We aim at providing a full picture of fault characterization at multiple description levels, by taking into consideration microarchitectural specifications. This will help in assessing the realism of already existing fault models, eliminate unrealistic models, and possibly propose new ones. Such methodology will also help in designing countermeasures at an appropriate cost at both levels: hardware and software. In addition to the extra experiments we provide, we also present preliminary experiments that validate the proposed methodology.

The rest of the article is organized as follows: Section 2 describes the setup of the performed experiments: the results are presented in Section 3, while a detailed discussion of the results is provided in Section 4. These results highlight the large array of possible fault effects to characterize, and the need for a methodology to solve such issues. Section 5 explains the proposed methodology to deal with the issues we observed with our experiments. Finally, the article is concluded along with perspectives in Section 6.

2. Experimental Setup

Physical fault injection experiments have been performed in order to see if the obtained faulty behaviours can be easily characterized and if they are consistent when making a modification to the target codes or other parts of the program. In addition to that, these experiments have been carried out to see if the observed behaviours will also differ from one target board to another.

This section presents the fault injection technique we used, the target boards, and the target programs. The results of the experiments and the related discussion are detailed in Sections 4 and 5, respectively.

2.1. Clock Glitch Fault Injection

Applying perturbations to the clock signal that is fed to the microprocessor is an effective and a non-invasive physical fault injection technique. During a normal execution, for example, in a 3-stage pipeline, at every rising edge of the clock, an instruction is fetched by the microprocessor, while another instruction (previously fetched) is being decoded or executed in another stage of the pipeline. Fig. 1 shows a normal behaviour when having a regular clock signal.

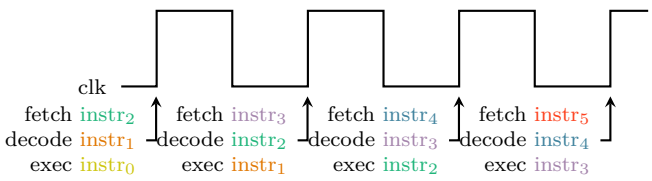


Figure 1: Normal behaviour of a 3-stage processor pipeline with a regular clock signal [21].

When performing clock fault injection, a glitch is injected just before or after the rising edge of the clock. This glitch would appear as a new clock cycle for the microprocessor. Therefore, a new instruction is fetched and the instruction previously decoded is executed. However, as the glitch disrupts the regular behaviour of the clock signal, a timing violation will possibly occur, leading to various kinds of faulty behaviours. Also, since the glitch is injected in the global clock, there is no particular knowledge about which architectural element could be affected.

When performing fault injection by clock glitch, the following parameters must be tuned:

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

Fig. 2 shows the glitch parameters with respect to a clock signal. It is worth mentioning that shift and width values should not be too large or too short. With too small values, the glitch will not be detected by the microprocessor, and hence, no effect will be observed, while too large values will allow the instructions to be executed normally without a fault as the instruction will have enough time to be fully executed.

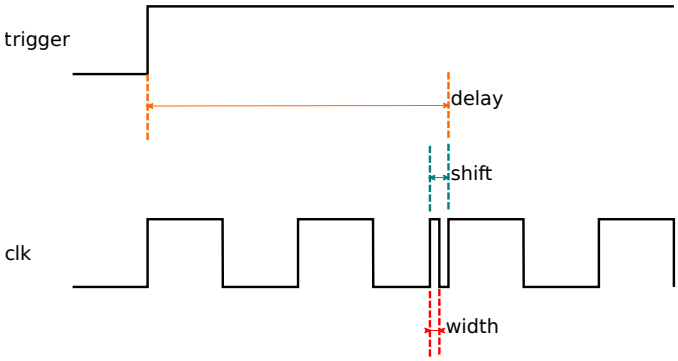


Figure 2: Clock glitch parameters.

2.2. Target Boards

The boards that are used for the experiments are the ChipWhisperer [22] boards: CW1173 ChipWhisperer-Lite and CW308 UFO baseboard with different targets. The targets are 32-bit microcontrollers, each of them embeds one of the following ARM cores: Cortex-M0 [23], Cortex-M3 or Cortex-M4 [24]. These ChipWhisperer boards have dedicated environment for side channel analysis, voltage and clock glitch of the target ARM core. We will leverage the clock glitch capabilities of this setup in the experiments. During an experiment, the ChipWhisperer-Lite board is connected to a control PC through a USB cable.

The Cortex-M0 core supports the Thumb-1 instruction set [25] and a small number of the Thumb-2 instruction set [26] while Cortex-M3 and Cortex-M4 cores support both sets entirely. These ARM cores include a pipeline with three stages: fetch, decode and execute. Up to two 16-bit instructions can be fetched at the same time. Additionally, Cortex-M3 and Cortex-M4 have a hardware integer divide and a prefetch unit with a maximum size of 128 bits [27]. Cortex-M4 has also additional components compared to the others, for example, it has a floating point unit and a digital signal processing unit.

2.3. Target Programs

The injection is performed directly into inline assembly instructions within a C program in order to provide fault effect characterization at ISA level. In order to better analyze the process of the injection, the program is divided into three parts as follows:

- Prologue: instructions for the initialization and the recording of the state before the injection happens.
- Target: instructions targeted by the fault injection as well as extra instructions that would allow observing any propagation effect.
- Epilogue: instructions for reading registers' state [R0-R12] and APSR[2] register (*i.e.*, Negative (N), Zero (Z), Carry (C) and Overflow (V) flags); the values are transferred through serial communication commands to the control PC.

Two series of NOP instruction are used to isolate the three parts. This is done to ease the process of the injection by limiting the search space of the injection parameters especially the delay. This also ensured that the prologue and the epilogue are not affected by the injection. The NOPs were not deleted after compilation as no optimization option is used for the compiler.

In the injection campaigns, two programs are targeted as shown in Listing 1 and Listing 2 respectively. Specific instructions in the target part of these programs are used to allow observing faulty effects on the data- and/or the control-flow of the program: any real-life application can be described in terms of its data flow and/or its control flow. The use of these instructions also allows observing other things. Firstly, it shows if the resulting faulty behaviours are related to these instructions or not, and hence, giving a better understanding of what triggers the faulty behaviour. Secondly, to check if we will be able to reproduce some faulty behaviours that are already mentioned in the literature. It also aims at obtaining the possible faulty behaviours based on the program flow, either the control flow as in Listing 1 or the data flow as in Listing 2. Finally, it helps to understand if software characterization at the ISA level is sufficient to build realistic fault models based on the observations.

[2]Application Program Status Register

Our goal is not to provide a complete characterization of every possible instruction, but rather provide a simple and efficient approach that will cover as much as possible the target architecture, and emphasize diverging behaviours due to fault occurrences. For this reason, we used instructions that explicitly have effects on different architectural elements, such as the APSR flags and the arithmetic logic unit. For the rest of this article, we will refer to the first target program as *Program 1* and to the second target program as *Program 2*.

For *Program 1*, the glitch is injected at the beginning of the target part. The remaining instructions aim at observing possible propagation effects. The registers R4 and R6 used in the experiment were initialized in the prologue to different values. Therefore, in a golden run, the zero flag remains clear, the branch is taken, and the instruction at line three is not executed. We use the term "golden" to refer to the normal behaviour of a program execution (*i.e.*, without any injection).

For *Program 2*, we distinguish two cases of execution: in the first case, just as *Program 1*, the glitch is injected while executing the beginning of the target part. On the other hand, in the second case, a different delay value for the glitch is used while executing the NOP series before the target part, in order to affect the prefetch stage of the targeted instructions.

```
1 CMP R4, R6 //r4-r6 then updates APSR
2 BNE labelx //if (Z!=1): jump to labelx
3 ADD R2, R4, R6 //r2 = r4 + r6
4 labelx:
5 ADD R5, R4, R6 //r5 = r4 + r6
```

Listing 1: Target part in *Program 1*: target control flow.

```
1 ADD R1, R1, 0x6 //r1 = r1 + 0x6
2 ADD R3, R3, 0xA //r3 = r3 + 0xA
3 ADD R4, R4, 0xB //r4 = r4 + 0xB
4 ADD R5, R6, R3 //r5 = r6 + r3
5 ADD R3, R3, 0xF //r3 = r3 + 0xF
```

Listing 2: Target part in *Program 2*: target data flow and arithmetic operations.

2.4. Injection parameters

The injection campaigns consist in repeating the clock glitch fault injection 10 000 times for the same shift, width and delay parameters. A single glitch is injected during each program execution. In the performed experiments, the glitch parameters were tuned to maximize the number of the observed faults for the instructions at the beginning of the target part of each program. These values are given here for reference, but it is important to emphasize that they can change according to the target board and the target program that are used in the experiment, or even environmental conditions such as temperature.

Table 1 shows the shift and the width values that are used for each target board. 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. With respect to the delay parameter, different factors can affect its value: the starting point of the trigger, the number of instructions in the prologue, the number of NOPs between the prologue and the target, and the position of the target instruction in the target part.

Board \ Parameter	**Width**	**Shift**
Cortex-M0	16	-14
Cortex-M3	10	-12
Cortex-M4	10	-12

Table 1: Glitch width and shift values that are used in the fault injection campaigns experiments. Values in % of one clock period.

3. Results

This section presents the results of the performed experiments, and it also describes the obtained faulty behaviours for the different used target boards. Three cases can occur as a result of the fault injection, regardless of the target program, with respect to a golden (reference) behaviour as follows:

- Crash: this class contains the cases when the fault injection causes a crash, a reset, or a failure when getting the final state of the target through the serial channel.
- Silent: this corresponds to the case when the outcome of the injection is identical to the golden state.
- Fault: this happens when the outcome state is different from the golden one.

The rest of this section is organized as follows: the first subsection presents the results for *Program 1*, while the second presents the outcomes for *Program 2*. The detailed analyses of the obtained results are discussed in Section 4.

3.1. Program 1 Results: Control flow Target

The results of the injection campaigns for the different target boards with regards to the three categories are shown in Table 2. Cortex-M4 board has the most successful faults, while Cortex-M3 has the most silent cases and Cortex-M0 has the most crashes. The obtained faulty behaviours for the different boards are described in Fig. 3. The x-axis presents the different observed faulty behaviours, while the y-axis shows their percentages over the successful faults *i.e.* without Crash and Silent cases.

Board \ Case	Silent	Crash	Fault
Cortex-M0	44.08	35.66	20.26
Cortex-M3	97.76	1.64	0.60
Cortex-M4	0.01	1.18	98.81

Table 2: Percentage of Silent, Crash and Fault cases when performing clock glitch fault injection on each target board running *Program 1*.

Complex faulty behaviours appeared as a combination of simpler models even if we only performed single fault injections. For example, the result of a single fault could be an instruction-skip and corruption of R0 at the same time.

During these campaigns, the following faults have been observed:

- Skip: it can be either a single or a double-skip. In other words, either we skip only the CMP instruction at line one in Listing 1, only the BNE instruction at line two, or both. If APSR flags have not been updated, then we assume that the CMP instruction was skipped. If APSR flags have been updated correctly and the ADD instruction on line three is executed, then we assume that the BNE instruction was skipped. If APSR flags have not been updated and the ADD instruction on line three is executed, then we assume that both instructions were skipped.
- R0 corruption: the value of R0 is different from its golden value. Among these corrupted values, we noticed the following: 0 (*i.e.*, the value of R0 becomes 0), right shift by 8, 16 or 24 bits.
- R1 Reset: R1 value becomes 0.
- R3 Reset: R3 value becomes 0.
- R4 corruption: either reset or right shift by 1 bit.
- R0-R5 Reset: all the values of R0 to R5 become 0.
- APSR corruption: one or more of APSR flags have different values from the golden ones.
- Propagation effect on R2: it is caused by executing the ADD instruction on line three. The execution of this instruction can be explained as the consequence of two events. The first explanation is that the BNE instruction at line two in Listing 1 was skipped. The second explanation is that the Zero flag was corrupted. This leads to the branch not being taken as in a normal case, where the Zero flag is 0. Instead, as a result of the injection, the Zero flag was set to 1. These two cases could not be discriminated as both of them might even occur together. In this experiment, this behaviour only appeared in Cortex-M0 and Cortex-M3 but not Cortex-M4.
- Propagation effect on R5: as a result of the corrupted value in R4, R5 has a wrong value at the end, since it is the sum of R4 and R6.

A second experiment has been carried out with the same fault injection parameters (*i.e.*, shift, width and delay) and initialization values but with a duplicated CMP instruction as shown in Listing 3. The second experiment has been performed to see if the faulty behaviours were consistent and to improve the understanding of the induced errors. In particular, its objective was to gain insight about the reason for the propagation effect on R2 as described above.

Regarding the three cases, Table 3 shows their percentages after this experiment. We can see that more faults were observed for Cortex-M0 in the second experiment while no crash cases were obtained. Regarding the Cortex-M4, there was a significant increase in the crash category and decrease in the successful faults. For Cortex-M3, both experiments were comparable in terms of population.

```
1 CMP R4, R6
2 CMP R4, R6
3 BNE labelx
4 ADD R2, R4, R6
5 labelx:
6 ADD R5, R4, R6
```

Listing 3: Target part in *Program 1* in the second experiment with duplicated CMP.

Board \ Case	Silent	Crash	Fault
Cortex-M0	61.29	0.0	38.71
Cortex-M3	96.07	2.69	1.24
Cortex-M4	0.88	39.32	59.80

Table 3: Percentage of Silent, Crash and Fault cases when performing clock glitch fault injection on each target board running *Program 1* after the second experiment.

The results are shown in Fig. 4. In addition to skip, APSR corruption and propagation effect on R5, the following behaviours were observed:

- R0 Reset.
- R4 corruption: different faulty values appeared in R4: 0, left shift of R6 value by 10 or 14 bits, and another faulty value that is equal to R7.
- R5 Corruption: either R5 has its initial value or the value of R7. Having the initial value can be considered as single-skip as well. This could happen due to a fault while fetching this instruction and executing the previous ones.
- R8 corruption: either set (*i.e.*, every bit has 1) or the value of R2.

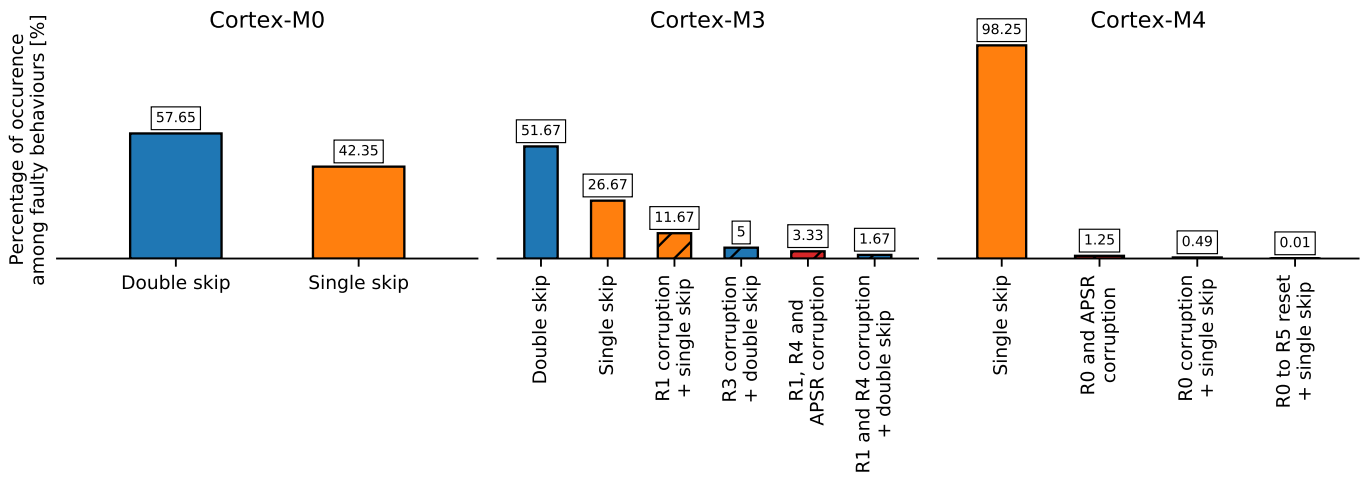


Figure 3: Observed faults for *Program 1* for all target boards.

- Propagation effect on R2: since we target only the beginning of the target instructions, this can not be caused by a skip or other perturbation of the BNE instruction. Therefore, this is necessarily caused by corruption of the Zero flag. This time, this behaviour only appeared in Cortex-M3.

3.2. Program 2 Results: Data flow and Arithmetic Operations Target

The results of the three categories for this experiment are shown in Table 4. Again, almost all the injections resulted in successful faults in the Cortex-M4 board, while they were silent in the Cortex-M3 board. The obtained faulty behaviours are presented in Fig. 5.

Board \ Case	**Silent**	**Crash**	**Fault**
Cortex-M0	75.51	17.28	7.21
Cortex-M3	97.93	1.2	0.87
Cortex-M4	0.0	1.16	98.84

Table 4: Percentage of Silent, Crash and Fault cases when performing clock glitch fault injection on each target board running *Program 2*.

A wide range of faulty behaviours is observed after this experiment as the following:

- Skip: it can only be skipping the first instruction, only the second, only the third, both the first and the third, both the second and the third, or the first 4 instructions (*i.e.* quad-skip).

- Repeat: repeat the first instruction. This behaviour appeared as a combination with skipping the second and the third instructions. In this experiment, it is only observed for the Cortex-M0 target board.

- R0 corruption: different faulty values observed in R0: set, reset, right shift of its original value by 4 or 20 bits, left shift of R2, *etc.*

- R1 corruption: Among the faulty values, there were: reset, a value that is related to the program counter, left shift of R2, the sum of R3 and 0x6 instead of R1 and 0x6, *etc.*

- R2 corruption: set only the most significant bit of the 32 bits. It only appeared for Cortex-M4 board.

- R3 corruption: faulty value related to the program counter, left shift of the original value of R1, the sum of the initial of R3, 0x6 and 0xF instead of R3, 0xA and 0xF, and other faulty values with no obvious relation.

- R4 corruption: between the faulty values that are found: the sum of R0 and 0xB, R1 and oxB, R2 and oxB or R3 and 0xB instead of R4 and 0xB.

- Propagation effect on R5: as a result of a faulty value that is found either in R3 or R6.

- R6 corruption: reset, the most significant bit is only set, the value of R0, left shift of R0 by 4 bits, left shift of R2, *etc.*

- All registers reset: it is only observed for Cortex-M3 target board.

A second experiment has been carried out for *Program 2*, but with adding only a NOP instruction to the prologue part. The main objective of this experiment was to investigate the consequences of a simple modification in the prologue to the target part of the program. Identical injection parameters were used, except adding one to the additional delay value that is used to target the prefetch stage. This is done to take into consideration the instruction that is

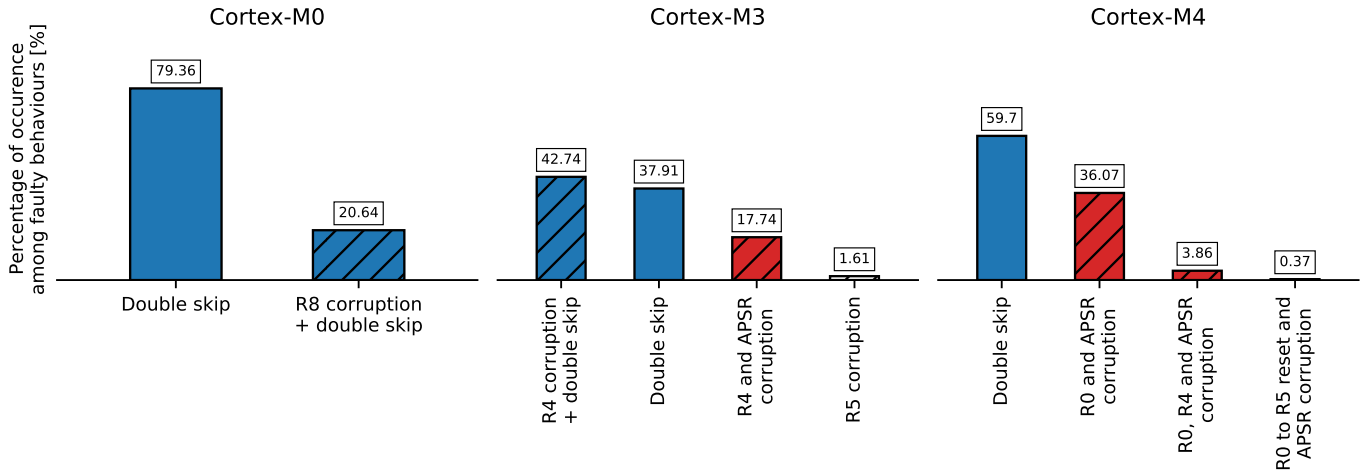


Figure 4: Observed faults for *Program 1* after the second experiment for all target boards.

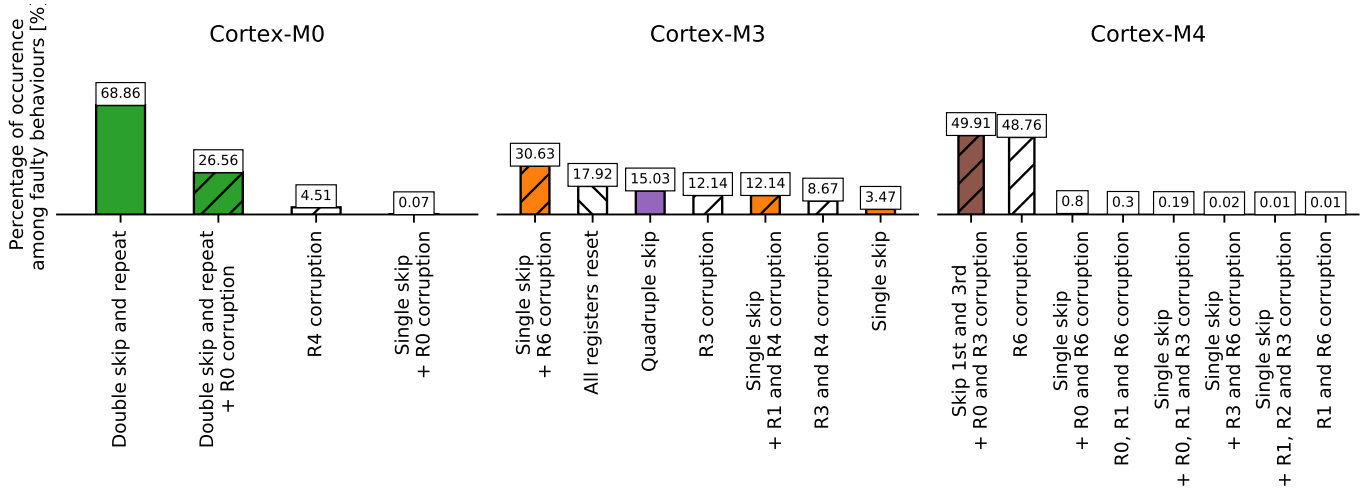


Figure 5: Observed faults for *Program 2* for all target boards.

added to the prologue. Table 5 shows the percentages of the obtained results classes. We can see a significant decrease in the successful faults for Cortex-M4 board, while close proportions for other targets with respect to the first experiment. The observed faulty behaviours are shown in Fig. 6.

Case / Board	**Silent**	**Crash**	**Fault**
Cortex-M0	75.07	12.78	12.15
Cortex-M3	98.55	0.99	0.46
Cortex-M4	49.85	0.24	49.91

Table 5: Percentage of Silent, Crash and Fault cases when performing clock glitch fault injection on each target board running *Program 2* after the second experiment.

In addition to the propagation effect on R5, the following faults have been observed:

- Skip: single-skip only appeared for the second instruction, while double-skip only occurred for the first and the second instruction.
- Repeat: again the first instruction is repeated. This time, it is obtained as a combination with skipping only the second instruction. This behaviour only appeared for the Cortex-M3 board.
- R0 corruption: only the following cases are observed this time: set, reset, only the most significant bit is set.
- R1 corruption: reset, a value related to the program counter, the sum of R0 and 0x6 and other faulty values without an obvious relation.
- R2 corruption: set, reset and other large values. This behaviour only appeared for the Cortex-M0 board.
- R3 Corruption: R3 has a seemingly random value. It only appeared for the Cortex-M3 board.

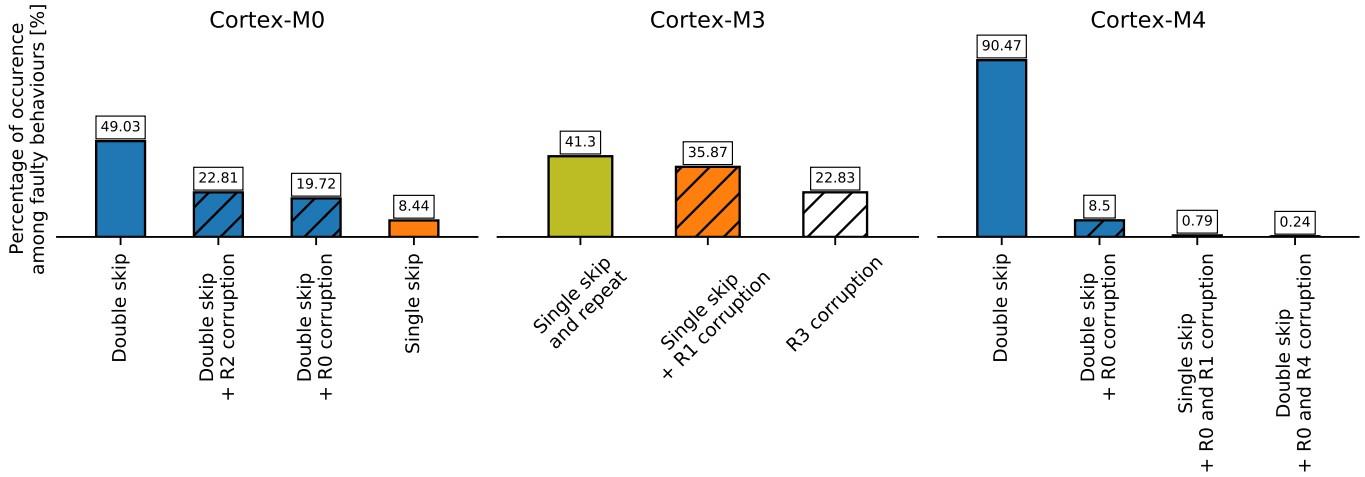


Figure 6: Observed faults for *Program 2* for all target boards after the second experiment.

- R4 corruption: the sum of 0xB and setting the most significant bit. It is only observed for the Cortex-M4 board.

4. Discussion

The aforementioned experimental results led to various conclusions, observations and questions. The following subsections discuss the results in details with respect to different aspects.

4.1. Target Board Dependency

For the same target program, different faulty behaviours can be observed depending on the target board that is used in the experiment. For example, in *Program 1* experiments, R8 Corruption is only observed for the Cortex-M0 board, while R0 to R5 reset is only observed for the Cortex-M4 board. Another example, in *Program 2* experiments, all registers reset behaviour is obtained only for the Cortex-M3 board, while skip and repeat faulty behaviour is observed for the Cortex-M0 and the Cortex-M3 but not for the Cortex-M4. In addition to that, Cortex-M4 target board has the most successful faults among the injections compared to the other boards. This could be explained with the fact that this board has more features and microarchitectural elements. Faulty behaviours as well may appear with different percentages. However, we have found that the occurrence probability of a specific behaviour can be increased or decreased by fine adjustments of the glitch parameters. Tuning the glitch parameters, in particular the delay, for the different boards in order to target different locations of the program could make some faulty behaviours appear or disappear.

4.2. Target Program Dependency

With respect to the target programs, small changes in the target code have large consequences on the observed faults as noticed for *Program 1* experiments: some faulty behaviours disappeared such as R1 and R3 reset. New faults appeared such as R8 corruption. Also, different corrupted values are observed, for instance, R0 had right shift by 8, 16 or 24 bits in the first experiment, but in the second, it had only reset. In addition to that, in [21], similar target program was used for the same target board but with different registers, and some of the obtained faults were different. For example, the propagation effect on the ADD instruction of line three in Listing 1 was obvious, but this time, it is not observed for the Cortex-M4 board. In addition, targeting the same sequence of instructions (*i.e.* the same target part) with two different prologues, even with a simple modification like adding a single NOP, could lead to various faulty behaviours as observed in *Program 2* experiments. For example, quad-skip occurred only in the first experiment, while skipping the first and the second is only observed in the second experiment.

4.3. On the Difficulty of Analyzing the Program Flow Faults

For the second experiment of *Program 1*, one might think that duplicating CMP will work as a countermeasure for APSR corruption since instruction duplication could work as a software countermeasure as described in [7, 28]. However, it did not as the injection affects two instructions in most cases, which might be related to the microarchitectural possibility to fetch two instructions at the same time. Hence, the corruption of APSR might still occur as a result of either corruption in the second CMP or corruption in the first and skipping the second. However, we cannot ensure that a single-skip in one of the CMP instructions has occurred as executing one of them, either properly or improperly, will mask the single-skip effect. Thus, at this step we can only say that either double-skip or APSR corruption have occurred. The corruption of APSR flags can be due to several causes: a change in the registers values

while executing CMP, an error while decoding the register numbers, an error that occurred when updating the APSR flags, a fault in the ALU while executing the subtraction between the registers, or a fault in a control signal related to the APSR flags. All these hypotheses cannot be validated or discarded without a better knowledge of the microarchitecture, which will help in having a suited fault model at the end.

For all injection campaigns on the two target programs, different forms of instruction-skip are obtained, for instance, single, double and quad. This can again refer to the possibility to fetch two instructions at the same time and to the prefetch unit that has a maximum size of 128 bits. More investigation and experiments are needed to uncover the origin of such faults at the microarchitectural level.

4.4. Registers Corruption

In terms of the injection effects on the registers, some registers that are not used in the program end up being corrupted as well: R0, R1, R3 and R8 for *Program 1*; R0 and R2 for *Program 2*. A question arises about what would be the proper fault model to account for this effect. In particular, such errors may have several causes: it might be related to the instruction opcode (*i.e.*, a fault during the instruction fetch) or to the execution stage of the pipeline. And most importantly, there is no explanation at this level for some corrupted values found in the registers, either used or not in the target part of the programs: 0, shift, values related to the program counter, or seemingly random values. We believe that some of these values are related to the microarchitecture, which will affect how a corrupted instruction will be executed. However, some observed faulty values can be explained as a source operand replacement. For example in *Program 2* experiments, some corrupted values in R4 were the result of the sum of R0 and 0xB or the sum of R3 and 0xB instead of R4 and 0xB. The former case can occur due to a fault in the decode stage (R0 instead of R4); while the latter can occur due to a fault of not updating one of the inputs to the arithmetic logic unit in the execute stage (R3 was just used in the previous instruction as shown in Listing 2). This explanation cannot be confirmed without further investigations.

4.5. State-of-the-art Fault Models Reproducibility

A very interesting point is also observed: using clock glitch fault injection, we were able to observe faulty behaviours which were obtained in the literature using other fault injection techniques. For example, quad-skip, which can also be described as 128-bit skip, was obtained in [8] using EM and in [5] using laser. Also, single-skip and source operand substitution were observed in [10] using EM as well, although in their experiments, they used super-scalar microarchitecture: Cortex-A9 [29]. Such a result could help researchers to study the effects of costly fault injections using low cost equipment and techniques such as clock glitch.

4.6. Summary

Finally, the aforementioned faults could be exploited as vulnerabilities in a security application. For example, an APSR corruption can lead to test-inversion where tests are considered very important in the control flow of critical applications.

To sum up, we saw how fault characterization is difficult based on a single level of analysis. These results show the difficulty of building consistent fault models that allow designers to predict the fault injection effects and design efficient and cost-effective countermeasures. Thus, additional research is necessary. In the next section, we propose a methodology that takes into consideration multiple levels of analysis by including software and RTL fault simulations as well as physical fault injections. This will help in explaining the observed points and answering the above-mentioned questions.

5. Proposed Methodology

This section provides a full description of the proposed methodology to infer fault models that will help in designing hardware and software countermeasures at an optimal cost. It deals with three different levels of understanding in order to provide a cross-layer fault analysis.

Fig. 7 depicts the proposed methodology. It is centered around a comparison between the obtained results that are stored in three databases (hardware, RTL and software databases) in order to make decisions about the consistency and applicability of RTL and software fault models. In other words, starting from the observations obtained at the lowest level of abstraction (*i.e.*, hardware level), it will be possible to optimize fault models at the RTL level, for example, by removing RTL faults that do no correspond to observable faulty outputs. Then, by using these RTL models, the models at software level will be optimized in a similar way, by adjusting them to not include behaviours that cannot be observed at RTL or hardware level. This will help in not over-engineering the countermeasures. Also, if a faulty behaviour obtained from the hardware injection does not belong to any faulty output from the RTL simulation, a new RTL fault model must be proposed, and hence, a new software fault model must be inferred. Hence, this will help in not under-engineering the countermeasures. The first three subsections explain each of the three parts in more details, while the last subsection provides a preliminary analysis to validate the proposed methodology. The numbers in Fig. 7 represent the sequential order in following the proposed methodology.

5.1. Hardware Fault Injection

At this stage, corresponding to *step 1* in Fig. 7, the goal is to perform physical fault injections using a variety of

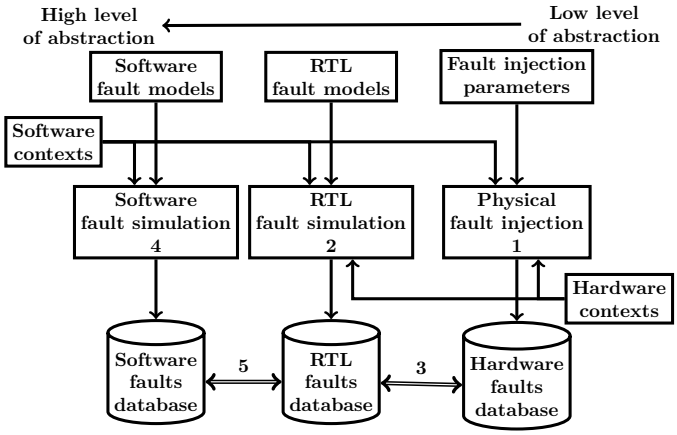


Figure 7: Proposed methodology.

injection techniques. Among these methods: EM fault injection, voltage and clock glitch injections using dedicated printed circuit boards and suitable generators. In each injection campaign, the following procedure is applied:

- Define different software contexts as target programs for the injection process. Faults are going to be injected while executing these programs on one of the hardware targets, for instance, microcontrollers, ASICs and FPGAs. A target part (or parts) within each software has also to be defined where faults should be injected.

- Define the set of injection parameters. For example, in the case of clock glitch attacks, the range of values for the shift and the width of the glitch, as well as the delay, as described and explained in the previous sections. These parameters as well as the target board layout must be taken into account when describing the fault model.

- Get a snapshot of the state of the target: for example, the registers and memory states will be read at the beginning and at the end of the program execution (using a serial communication link with the host PC or a debugger for example). The richer the information that can be accessed, the more precise the model will be: for instance, hidden performance counters could be used to get a more detailed view of the internal state, in particular when advanced microarchitectural features are implemented. Then, the snapshot will be compared with the configuration of a golden run. The faulty behaviours will be stored in a database (*Hardware faults database* in Fig. 7). This step will allow us to observe the relation between the observed faulty behaviours and the instructions in the target part. In other words, the aim is to assess if there is a direct relation (*i.e.*, the effect corresponds to the target instructions), an indirect relation (*i.e.*, the effect is a result of a propagation effect), or no relation at all, which may require further analysis.

Thanks to the analysis of the observed faulty behaviours, a fault model inference process will be followed by generalizing the obtained faulty behaviours.

5.2. RTL Fault Simulation

In order to understand what is exactly happening internally at the microarchitectural level and be able to know the origin of a fault, fault simulation campaigns are going to be performed on the RTL description of the microprocessor; this is *step 2* in Fig. 7. This will help in characterizing further the hardware faulty behaviours by giving more observability and controllability.

With RTL fault simulation, it is possible to inject faults in a very precise manner into the microarchitecture. For instance, inter-stage pipeline registers, multiplexers and different arithmetic units that are involved in executing an instruction in the pipeline stages can be targeted. The injection will consist in forcing the corresponding signals according to fault models such as single or multiple bit-flips, bit-sets and bit-resets.

As in the previous step, the resulting faulty behaviours will be stored in a dedicated database and then be compared with those obtained from the physical injections. To ease the comparison and the fault characterization at the RTL level, a divide-and-conquer approach is used to reduce the complexity: the fault simulation is applied into specific RTL module or specific microarchitectural component at once. This comparison will help in two aspects, as shown visually in Fig. 8. On the one hand, this aims at explaining at the hardware level the faulty behaviours obtained from physical injections, and hence, making the fault effect characterization easier. The explanation is done by revealing the origin of the fault at the RTL level and determining the responsible microarchitectural component, the register, or even the single flip flop behind obtaining the faulty behaviour resulting from injecting the fault(s) (*step 3* in Fig. 7). On the other hand, it also helps in validating and assessing the realism of the used RTL fault models. Hence, it provides a full overview to the hardware designer to build the required countermeasures.

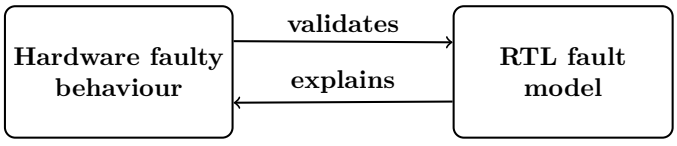


Figure 8: Relation between hardware faulty behaviour and RTL fault model.

5.3. Software Fault Simulation

Software faults will be injected into different target programs. This can be done by performing modification, deletion or addition of instructions in the original program. This represents *step 4* in Fig. 7. The software faults may correspond to a large variety of faulty behaviours modeled at the ISA level. In other words, typical fault models

such as instruction skip, instruction replacement, instruction corruption, register value corruption, test-inversion, or a combination between these models, will be injected into the programs by modifying the instructions. In addition to that, other faulty behaviours must be generated using more complex fault models which take into consideration the modern design of some hardware blocks. This includes, for instance, forwarding and speculative execution. In this case, dedicated techniques shall be employed to model the advanced architectural characteristics and the related faults at the ISA level. This can be achieved by modifying the source code before the compilation or mutating the compiled code itself after applying different combinations of compiler optimization levels.

The expected faulty outputs will again be stored in a corresponding database. Then, a comparison process similar to the one mentioned earlier will take place between the RTL and the software faulty results. In *step 5*, an RTL model validates the consistency of a software model, whereas a software model will be usable to describe the occurrence and explain an RTL model at the application level, which makes the fault effect characterization at this level easier.

5.4. Preliminary Validation

To validate our approach, a preliminary analysis study has been conducted by performing RTL fault simulation experiments on the same target part of *Program 2* shown in Listing 2. The RTL description is for the ARM Cortex-M3, in particular, the DesignStart evaluation version [30].

Using typical RTL fault models, we were able to observe various faulty behaviours. For example, by a single bit-set or a single bit-reset to specific signals that are located in the path of the instructions fetch between the Flash memory and the core as shown in Fig. 9, in particular in the interface, we were able to observe the following faults:

- Quad-skip or 128-bit skip.
- Double-skip of the second and the third instructions.
- Skip the second and repeat the first.
- 128-bit skip and repeat the previous 128-bit. This model is described in [8].

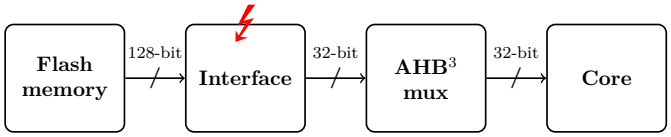


Figure 9: Instructions datapath from the Flash memory to the processor core.

[3] Advanced High-performance Bus.

With these simulation experiments, we were able to validate the realism of some RTL fault models, such as single-bit manipulation for specific signals in a particular microarchitectural component, and the relevant inferred software fault models that were obtained either by our experiments, for instance, double-skip and single-skip and repeat, or in the literature such as 128-bit skip and repeat. And not only that, we were also able to determine the origin of such faults. However, we cannot exclude that faulting other signals in other microarchitectural components could lead to similar behaviours. These experiments represent the start of the implementation of the proposed methodology at the RTL level.

To include the software fault simulation in this preliminary study, applying mutations to the same target part of *Program 2* shown in Listing 2 allows obtaining the same aforementioned faults, which validates the realism of the inferred fault models at the software level. For example, replacing the second instruction by a copy of the first instruction lead to skip the second instruction and repeat the first as shown in Listing 4.

```
1 ADD R1, R1, 0x6 //r1 = r1 + 0x6
2 ADD R1, R1, 0x6 //r1 = r1 + 0x6
3 ADD R4, R4, 0xB //r4 = r4 + 0xB
4 ADD R5, R6, R3 //r5 = r6 + r3
5 ADD R3, R3, 0xF //r3 = r3 + 0xF
```

Listing 4: Mutating Listing 2 to obtain skip the second and repeat the first fault.

5.5. Discussion

Once the links between the three levels are established and formalized, a software developer can design the most suitable countermeasures for a given context. For sure, countermeasures will be studied carefully at both levels: hardware and software. Therefore, the proper ones will be applied by taking into account their cost and their effect on the performance. Thus, if a countermeasure can be implemented at both hardware and software levels with comparable efficiency, only the software option may be considered since software countermeasures are usually less expensive to implement. Therefore, the "cross-layer" aspect can be extended later on to the design of countermeasures.

Getting a complete and detailed analysis of the full design might not be achievable, for complexity issues. Large designs have millions of gates, and addressing every single one is not reasonable. As already suggested, the designer can focus on critical components, or adopt a divide-and-conquer approach to reduce the overall complexity. Other techniques exist: in [5], the authors aim at optimizing the number of experimental injections in the case of multiple faults, but they limit their analysis at one level at a time (e.g., software simulation) to port the results at other levels (e.g., set of injection parameters). Unlike them, our approach uses layer crossing in order to explain the fault origin. In the future, more advanced techniques could be

used to infer relations among models: this, however, is currently out of the scope of this work, and will be addressed in the future.

It is important to highlight that while the full methodology greatly benefits from the full availability of all levels. In our scenarios, we chose embedded microcontrollers that are widely used in the IoT domain, and represent thus a significant test case. It is worth mentioning that an authorized access to the architectures used in this work is provided under the ARM Academic Access (AAA) agreement. This allows us to apply the RTL fault simulation to the three architectures: Cortex-M0, Cortex-M3 and Cortex-M4. In section 5.4, a suite of RTL fault simulation experiments has been performed on a Cortex-M3 version, which works as a test case for the described methodology.

On the other hand, we can observe that its application can be limited to a subset of levels (*e.g.*, ISA to a higher software level) once the models (and their respective relation) have been defined. For example, if the manufacturer makes the model description available, this information could be exploited to guide the developers in implementing more robust algorithms without having access to the hardware descriptions. Additionally, building the fault models at physical level needs to be done only once for a specific implementation, and data can be reused afterwards for different software contexts. Our methodology does not take into account any architecture-specific feature, and we consider it to be architecture-independent.

6. Conclusion and Perspectives

In this article, we presented the existing problems in analyzing and understanding fault attacks in complex microarchitectures. We highlighted this by providing experimental evidence of intrisically microarchitectural faults, using clock glitch as the fault injection technique. The experimental results showed that the faulty behaviours can be target-dependent, prologue-dependent and architecture-dependent. After that, we proposed a new methodology to provide a cross-layer analysis for characterizing faulty behaviours, along with preliminary experiments to validate it. Such methodology can be used to build realistic fault models at different levels such as RTL and software. It can also provide explanation for the origin of some faults. Hence, this gives the possibility to design suited countermeasures at the most appropriate cost at hardware and software levels.

In terms of perspectives, performing larger campaigns of RTL fault simulation into different microarchitectural components and different RTL descriptions will be taken into account. Also, automating the analysis of the faulty behaviours and the comparison among the three different databases obtained at the different layers is necessary to move forward in this research direction.

Acknowledgment

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

References

[1] D. Boneh, R. A. Demillo, R. J. Lipton, On the importance of checking cryptographic protocols for faults, Springer-Verlag, 1997, pp. 37–51.

[2] A. Barenghi, L. Breveglieri, I. Koren, D. Naccache, Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures, Proceedings of the IEEE 100 (11) (2012) 3056–3076.

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

[4] B. Colombier, A. Menu, J.-M. Dutertre, P.-A. Moëllic, J.-B. Rigaud, J.-L. Danger, Laser-induced Single-bit Faults in Flash Memory: Instructions Corruption on a 32-bit Microcontroller, in: 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), IEEE, McLean, United States, 2019, pp. 1–10.

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

[6] S. P. Skorobogatov, R. J. Anderson, Optical fault induction attacks, in: B. S. Kaliski, ç. K. Koç, C. Paar (Eds.), Cryptographic Hardware and Embedded Systems - CHES 2002, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 2–12.

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

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

[9] L. Dureuil, M. Potet, P. de Choudens, C. Dumas, J. Clédière, From code review to fault injection attacks: Filling the gap using fault model inference, in: N. Homma, M. Medwed (Eds.), International Conference on Smart Card Research and Advanced Applications, Vol. 9514 of Lecture Notes in Computer Science, Springer, Bochum, Germany, 2015, pp. 107–124.

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

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

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

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

[14] N. Theißing, D. Merli, M. Smola, F. Stumpf, G. Sigl, Comprehensive analysis of software countermeasures against fault attacks, in: E. Macii (Ed.), Design, Automation and Test in Europe, Grenoble, France, 2013, pp. 404–409.

[15] A. Höller, A. Krieg, T. Rauter, J. Iber, C. Kreiner, Qemu-based fault injection for a system-level analysis of software countermeasures against fault attacks, in: Euromicro Conference on Digital System Design, IEEE Computer Society, Madeira, Portugal, 2015, pp. 530–533.

[16] S. Bergaoui, P. Vanhauwaert, R. Leveugle:, A new critical variable analysis in processor-based systems, IEEE Transactions on Nuclear Science 57 (4) (2010) 1992–1999.

[17] C. Spensky, A. Machiry, N. Burow, H. Okhravi, R. Housley, Z. Gu, H. Jamjoom, C. Kruegel, G. Vigna, Glitching demystified: analyzing control-flow-based glitching attacks and defenses, in: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2021, pp. 400–412.

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

[19] J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, A. Papadimitriou, Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor, Microprocessors and Microsystems 71 (2019).

[20] RISC-V Foundation, The RISC-V instruction set manual, https://riscv.org/technical/specifications/, [Accessed: May 4, 2021].

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

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

[23] J. Yiu, The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ Processors, Newnes, 2015.

[24] J. Yiu, The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, Newnes, 2013.

[25] ARM Limited, Armv6-m architecture reference manual, https://developer.arm.com/documentation/ddi0419/c?lang=en, [Accessed: November 22, 2021].

[26] ARM Limited, ARM architecture reference manual Thumb-2 supplement, https://developer.arm.com/documentation/ddi0308/d, [Accessed: May 4, 2021].

[27] ARM Limited, Arm cortex-m programming guide to memory barrier instructions, https://documentation-service.arm.com/static/5efefb97dbdee951c1cd5aaf?token=, [Accessed: November 28, 2021].

[28] N. Theißing, D. Merli, M. Smola, F. Stumpf, G. Sigl, Comprehensive analysis of software countermeasures against fault attacks, in: 2013 Design, Automation Test in Europe Conference Exhibition (DATE), 2013, pp. 404–409. doi:10.7873/DATE.2013.092.

[29] ARM Limited, Cortex-a9 technical reference manual, https://developer.arm.com/ip-products/processors/cortex-a/cortex-a9, [Accessed: November 29, 2021].

[30] ARM Limited, Arm cortex-m3 designstart eval rtl and testbench user guide r0p0, https://developer.arm.com/documentation/100894/0000/introduction/ about-cortex-m3-designstart-eval, [Accessed: November 29, 2021].