Version of Record: https://www.sciencedirect.com/science/article/pii/S0141933118304745
Manuscript_de01a42009f0ff086612534da2249b9b

Cross-Layer Analysis of Software Fault Models and Countermeasures Against Hardware Fault Attacks in a RISC-V Processor

Johan Laurent[a], Vincent Beroulle[a], Christophe Deleuze[a], Florian Pebay-Peyroula[b], Athanasios Papadimitriou[a]

[a.] Univ. Grenoble Alpes, Grenoble INP*, LCIS
26000 Valence, France
firstname.lastname@lcis.grenoble-inp.fr

[b.] Univ. Grenoble Alpes, CEA, LETI
38000 Grenoble, France
firstname.lastname@cea.fr

Declarations of interest: none

***Abstract* – Fault injection is a powerful technique for attacking digital systems. Software developers have to take into account hardware fault effects when system security is a concern. Software fault models have been developed in an attempt to predict these faults. However, these models are often designed independently of any hardware consideration and thus raise the problem of realism. The generality of these models often cannot account for the specificities of each architecture. As a consequence, software countermeasures based on such software fault models do not guarantee an effective protection against fault attacks. Processor microarchitecture should be precisely analysed to better understand faulty behaviours. A cross-layer approach can then be developed, using conjointly hardware and software characteristics to design stronger software countermeasures with reasonable overheads. To illustrate this assumption, this paper shows actual faulty behaviours observed in a RISC-V processor RTL simulation, and shows that they can bypass countermeasures designed to protect against faults predicted by typical software fault models.**

***Keywords* – Fault attack, software fault model, RISC-V, software countermeasures**

1. Introduction

Nowadays, fault injection is a recognized threat for system security. This technique has been used extensively to deduce information from a system or to bypass security measures [1]. These successful attacks impose the need to re-assess system security. To this end, fault models have been designed at different abstraction levels. In this paper, we use a cross-layer approach considering both Register-Transfer Level (RTL) and software level fault models.

In its interpretation of the Common Criteria Methodology for smartcards [2], the Joint Interpretation Library defined several effects that a fault can produce in a processor. These effects include: modification of a value in memory, instruction skipping, instruction replacement, test inversion, jumps and calculation errors. Because of their generality, these software fault models have been used in works treating fault attacks in software [3][4][5]. They offer a relatively easy and flexible way to study the impact of faults in a program. However, while they seem plausible, they are not devoid of any criticism. As Touloupis et al. raised in [6], they suffer from two intrinsic limitations. First, they do not take into account microarchitectural states that are not visible to the programmer. Second, they only consider fault injection between instructions but not during their execution. These observations arise from the fact that software fault models are designed independently of hardware considerations. They are thus limited in their ability to model reality.

To quantify the inaccuracies of software fault models compared to injections at lower abstraction levels, several works have been carried out. Cho et al. showed in [7], with an extensive fault injection campaign, that realistic faults, injected at a low level of abstraction, cannot all be modelled by a simple software model. Wang et al. also carried out an experiment in [8] to locate the origin of faults in a processor pipeline. Finally, Espinosa et al. studied in [9] how a single fault can propagate into multiple architectural states that affect program execution.

In order to fill the gap between hardware and software fault models, some works have been conducted. In [10], Moro et al. carried out a fault injection campaign on an ARM target. From the results, they deduced which model was the most relevant for their processor. Dureuil et al. proposed in [11] and [12] a methodology to infer a software fault model from injection experiments. They built a model by iteratively proposing hypotheses for the origin of faults, and testing these hypotheses. Finally, in [13], Kelly et al. performed fault attacks on different instructions to observe their effects on registers and memory. In these three papers, one of the main

* Institute of Engineering Univ. Grenoble Alpes

This work was funded thanks to the French national program 'programme d'Investissements d'Avenir, IRT Nanoelec' ANR-10-AIRT-05.

assumptions was that the processor was a black-box: neither its RTL description nor its layout were available; the faults were thus physically injected in the actual device. As a consequence, the modelling of fault effects resulted from observing for example the architectural registers or the memories during the execution of the faulted program. This high level point of view can however lead to misinterpretations of faulty behaviours because microscopic side effects might get masked by macroscopic behaviours.

From this difficulty to correctly model faults at the software level results the difficulty to design software countermeasures that would correctly thwart actual fault attacks. This assumption has been illustrated by Yuce et al. in [14]. They demonstrated that a single clock-glitch could bypass some typical software countermeasures. The last section of our paper presents a similar work; more precisely, we show that single-bit attacks can be a threat to these countermeasures.

In this paper, we illustrate some shortcomings of current software fault models. In [15], we already showed such shortcomings on a few examples. The present paper is an extension of this first work. It provides a description of how we extracted faulty behaviours from the processor microarchitecture, as well as a more thorough report of faulty behaviours we have observed, and an analysis of more software countermeasures. Our contribution is twofold. We first deduce, by analysing its RTL architecture, some faulty behaviours which can be expected from a processor attacked with a single-bit fault. Some of these behaviours are not covered by current software fault models. Then, with that knowledge, we attack typical software countermeasures, and expose the fact that they are not adapted to actual faults injected in a lower abstraction level. Our goal is to demonstrate that a precise analysis of the microarchitecture is a required step to design accurate software fault models. This analysis can in turn lead to design more effective countermeasures that are better suited to the end application.

This work does not intend to show an exhaustive analysis of all possible unexpected faulty behaviours in a processor, but rather to expose some that can happen, and that an attacker could take advantage of, so as to bypass countermeasures. These faults are specific to the hardware implementation of the processor, and cannot be anticipated at the software level.

Section 2 describes the fault model that is used, as well as the simulation methodology. Section 3 presents the processor under attack, and its microarchitecture. Section 4 shows faulty behaviours (corresponding to software faults) that have been observed in simulation. Finally, section 5 shows how some common software countermeasures can be attacked with these software faults.

2. Fault Injection Methodology

This section shows the methodology used in this study. First, the fault model is discussed and then the simulation methodology is described.

2.1. Fault Model

There are many ways to model a fault, depending on the abstraction level under consideration. In the present work, faults were single bit-flips in RTL descriptions. That means that we only considered faults in a single flip-flop at a time. We did not consider electrical masking or any phenomenon that happens before the fault is captured in a flip-flop.

We chose to use single-bit faults since even these simple faults can highlight vulnerabilities of the processor that are not thwarted by software countermeasures. The single-bit fault model is a common model [16] that can be viewed as a preliminary work for more complex models. It is used by both the hardware and software communities.

This paper is focused solely on faults that affect the control signals in the pipeline. Faults could also have been injected in instruction words or in data, like in [17] or [5]. However, these latter faults are also visible in higher levels of abstraction than RTL. In fact, the analysis of fault propagation on data is completely independent of any hardware consideration. As for faults on instruction words, they are only dependent on the Instruction Set Architecture (ISA), but not its implementation. Both can be analysed at ISA level.

2.2. Fault simulation methodology

There are many ways to perform a fault injection campaign, as described in [18]. In this work, we used a simulation-based technique. All faults were injected using a simulator command that sets a signal to a specified value, until this signal is driven by its source.

Injection targets and clock cycles were not randomly chosen. They were carefully selected to show potential vulnerabilities that were found during microarchitecture analysis. This analysis told us what faulty behaviours could be expected, and simulation was used to verify that these behaviours were indeed possible. Faults were first injected in the simulation of the execution of isolated assembly instructions. Simulation was used to propagate each fault in a state visible to the software developer (i.e., a general purpose register). In a second phase, we attacked typical software countermeasures. This time, simulation told us if the attack was successful: we could check that the fault corrupted the protected data while bypassing the countermeasure.

For the sake of simplicity, we did not check every flip-flop at every clock cycle. As a consequence, it is possible that faults that are highlighted here also create collateral effects that we are not aware of. The goal here is not to give an exhaustive view of all possible faults and of their effects, but to highlight the existence of these sorts of

behaviours. Besides, the knowledge we have of these faults is sufficient to successfully attack typical countermeasures, as we show in section 4. Exhaustiveness of the fault effects could be a perspective of this work. We would like to emphasize that aside from checking the faulty behaviour, we also made sure that no exception was raised in the processor. We considered that a fault is detected if an exception is raised. The simulation ended a thousand cycles after the injection, to make sure that there was no latent state that raised an exception.

3. Hardware Architecture

This section shows the processor microarchitecture, and how it executes instructions. This presentation is made in the aim of better understanding the faults that are highlighted thereafter. Finally, a brief discussion on hardware countermeasures is made.

3.1. RISC-V architecture

RISC-V is an open Instruction Set Architecture [19] developed by the University of California, Berkeley since 2010. Like its name implies, it is a Reduced Instruction Set Computer. It uses a load/store architecture, which means that memory and register operations are decoupled. Finally, it can operate on various data widths: 32, 64 or even 128 bits.

The RISC-V architecture is modular: it is composed primarily of a base integer instruction set (I), to which several extensions can be attached. Amongst these extensions are: multiplication/division (M), atomic operations (A), and floating-point operations with single (F) or double precision (D). The base integer set with these four extensions (IMAFD) is commonly called the general-purpose processor (G).

The basic implementation of RISC-V is called Rocket core. It is composed of a 5-stage pipeline that implements the G variant of RISC-V (64-bit in our case). The pipeline stages are: Instruction Fetch (IF), where instructions are fetched from instruction memory; Instruction Decode (ID), which decodes the instruction, drives control signals and reads data from the register-file; Execute (EX), where operations are executed by the ALU; Memory (MEM), which undertakes memory reads and writes; and finally Write-Back (WB), where results from previous stages are written in the register-file.

The Rocket core is used in an implementation called LowRISC. Specifically, we used LowRISC v0.2 [20], which was released in December 2015. To lift any ambiguity, in the rest of the paper, registers that are visible to the developers are called GPR (General Purpose Registers), while internal registers that are not visible to the programmers are simply called “registers”. The latter are microarchitectural states that exist only at RTL. GPRs are grouped into the register-file.

3.2. Microarchitecture

Before diving into the different faulty behaviours that have been obtained in RTL simulation, it is necessary to present in detail the microarchitecture of the processor and explain how it works. Figure 1 shows the last three

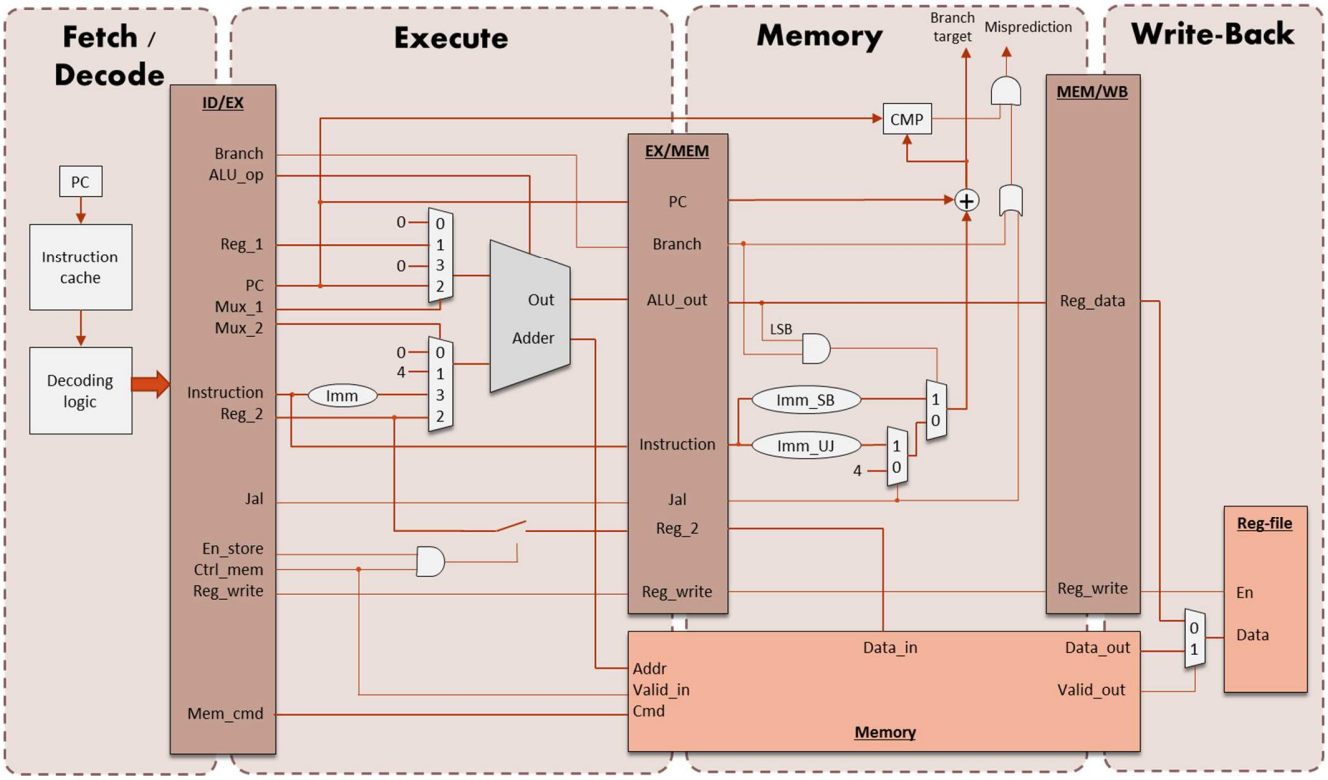


Figure 1: Simplified view of EX, MEM and WB pipeline stages in LowRISC v0.2

pipeline stages (EX, MEM and WB) of the specific implementation which has been used. The Instruction Fetch and Instruction Decode stages are not detailed because few faults were injected in these two stages. Indeed, most of the control logic is located in the last three stages, and hence more interesting behaviours are obtained there. Faults in the Fetch and Decode stages often result in a wrong instruction being fetched, or an instruction being mutated. These models are not our focus since they have already been studied in the past. Figure 1 is a simplified view of the actual architecture, but it displays all signals that are necessary to understand how the processor behaves when attacked.

This figure has been obtained by looking at the hardware description of the Rocket core. The ID/EX, EX/MEM and MEM/WB rectangles represent registers that store signals between different pipeline stages (the execute stage is between ID/EX and EX/MEM registers for example). Faults were only injected in these signals. Anything in-between is combinational logic, and thus outside of our fault model.

3.3. Fault-free execution of a BEQ instruction

To better understand how the pipeline works, we fisrt show a simple fault-free example by examining a BEQ instruction. BEQ stands for "branch if equal". This instruction compares the values of two GPRs, and branches to another part of the program if they are equal. The syntax of such an instruction is "`beq a1, a2, offset;`", where a1 and a2 are the two GPRs to compare, and offset indicates where to jump in the code. This is an offset to the current Program Counter (PC).

The processor uses a gshare branch predictor, so, depending on the branch history, the processor dynamically speculates either a branch taken or a branch not taken. For simplicity, we consider here the case where branches are predicted not taken. So the processor continues to fetch the following instructions, even if it does not know yet if they should be executed or not. If the branch is finally taken, the processor flushes the pipeline to disable the instructions fetched as a result of the misprediction.

During the execution of a BEQ instruction, ID/EX registers "branch", "ALU_op", "Reg_1", "PC", "Mux_1", "Mux_2", "Instruction" and "Reg_2" are used. In the Execute stage, input values read from the register-file are called "Reg_1" and "Reg_2". These inputs are transmitted to the ALU through two multiplexors that select entries 1 and 2 respectively (through signals "Mux_1" and "Mux_2"). "ALU_op" is used to tell the ALU which operation to perform. In this case, it is "set if equal", which sets the output of the ALU to 1 if the inputs are equal; 0 otherwise. The result of the ALU is memorized in one of the EX/MEM registers: "ALU_out". At the same time, "Branch" is simply transferred from ID/EX to EX/MEM registers, as it is only used in this stage.

The more complex part of a BEQ instruction takes place in the MEM stage. It is in this stage that the decision to branch is taken. First, it is necessary to compute the branch target. As can be seen in the MEM stage of Figure 1, branch target is the sum of current PC, and an offset. This offset is selected by a series of two multiplexors. By default, the offset is four. That means that the next instruction should normally be at address PC+4 (which is correct considering 32-bit instructions). When the comparison of the branch instruction is true (LSB of ALU_out is one), the second multiplexor selects the other entry. So the offset of branch target is modified: it is not four anymore, but a value that is deduced directly from the instruction word (Imm_SB, which corresponds to an immediate value for a conditional branch). The new branch target is then compared with the PC of the instruction currently in the EX stage. If they are not equal, that means that there has been a misprediction: the following instruction in the pipeline is not the next one to execute. In this case, the pipeline is flushed and the global PC is set to the branch target. Finally, nothing is done in WB stage: no value has to be written in the register-file.

3.4. Other instructions

Of course, other classes of instructions can be executed by the processor. Here we give a quick overview of these instructions.

R-type instructions refer to instructions that operate on two GPRs, and write the result in a third GPR. These instructions include the addition, subtraction, as well as logical operations like AND, OR, XOR and shifts. Operations are made by the ALU in the EX stage, and the result is written into the register-file during the WB stage.

Loads and stores are operations that are used to read or write in RAM. Memory address is computed by the ALU and then sent to the memory. During the EX stage, the memory also receives a validation signal and the type of operation to perform (read/write). The MEM stage is only useful for store operations: data is transmitted to the memory. Finally, the WB stage is only useful for load operations when data is written into the register-file.

Jumps are used to unconditionally jump to another part of the program. The specific instruction used here is JAL, which means "jump and link". In addition to jumping, it also stores the return address (PC of next instruction) in a GPR. This is similar to the sub-routine "call" in x86 architectures. Note that the other jump instruction, JALR, is not represented in Figure 1 to not make the figure too complex.

Finally, to appreciate one of the most interesting faults that were found, it is useful to understand a concept used in most modern processor implementations: forwarding. Forwarding, also called bypassing, is a technique used to solve data hazards in an efficient way. A data hazard happens when there are dependencies between consecutive instructions: an instruction needs to use the

result of a previous instruction that has not finished executing. Listing 1 gives an example of such a data hazard. To solve this problem, modern processors can directly feed the value of a pipeline stage to the ALU. So this value can be re-used directly, and there is no need to wait for the value to be written in the register-file. This process is called forwarding. R-type instructions can forward values to the next two cycles, while load instructions can only forward to the next one. For more details on how hazards and forwarding work, see [21].

```
ADD a2 = a0 + a1
ADD a3 = a2 + a0
```

Listing 1: Example of a data hazard. Here, a2 is used in the second instruction, but its value is computed in the previous instruction, and thus is not comitted to the register-file yet.

3.5. Discussion on hardware and software countermeasures

The Rocket core implementation does not have hardware countermeasures built in. So, one could argue that this study is not relevant, since an application with security concerns should preferably be run on a secured processor. There are however several considerations to temper this point of view. First, it is difficult to completely protect a processor in hardware, because it often induces high costs and performance overheads. There will often remain parts of the processor that are still vulnerable. As a side note, because this study is focused solely on faults injected in the processor pipeline, and not in the register-file or memories (both instruction and data), we can consider that these structures are protected. Faulty behaviours that are shown here would still be present if these structures were hardened.

Second, it may be better to protect in software rather than in hardware. Indeed, software countermeasures are cheaper and more flexible (they can be adapted to the system more easily). If a processor has to be hardened against fault injections, it is better to first evaluate if these faults can be mitigated effectively in software. For example, if floating point operations are rarely used in an application, and are not very critical, it is better to protect them in software (instead of paying expensive hardware countermeasures). To evaluate how to protect a system effectively, precise software fault models are needed. This is another reason why we claim a microarchitecture analysis is needed.

Instruction	Origin	Faulty behaviour
Branch	Branch	Prevent the branch from being taken
	Mux_1 or Mux_2	Comparison to 0 instead of one of the arguments.
	ALU_op	Test inversion(4)
	Write_enable	Normal operation, plus set one GPR to zero or one(3)
	(not represented)	Execution of the following instruction, even if branch is taken(2)
R-type	Write_enable	Prevent the result from being written into register-file(1)
	Branch	Jump in addition to the normal operation (only if the result of the ALU is odd)(3)
	Mux_1 or mux_2	Replace one argument with 0
	ALU_op	Perform another operation(4)
Load	Write_enable	Prevent the value read from being written into register-file(1)
	Ctrl_mem	Prevent the reading and write the address into destination GPR.
	ALU_op	Subtraction instead of addition for address calculation.
	Mem_cmd	Write last written data in memory, and write the address into destination GPR.
	Mem_cmd	Normal load operation, then write last written value in memory.
	Mem_cmd	Normal load operation, then write in memory the sum of loaded value and the last written value.
Store	Ctrl_mem	Prevent the store operation.
	ALU_op	Subtraction instead of addition for address calculation.
	Write_enable	Normal store operation, and write the address into a GPR (depending on the address offset).
	En_store	Write last written value instead of the new one.
	Mem_cmd	Write new value XOR last written value.
Jump (jal)	Write_enable	Prevent return address from being written in destination GPR.
	Mux_2	Write PC instead of PC+4 for the return address.
	Jal	Prevent the jump from happening
	(not represented)	Execution of the following instruction(2)

Table 1: Faulty behaviours for different instructions. Faults marked with a number can have side effects or consequences that are explained in the part about complex faults.

4. Fault propagation analysis

Now that some parts of the microarchitecture and a fault-free behaviour have been explained, the fault propagations can be easily understood. We now disrupt a correct execution by changing one of the control signals. To illustrate this, we first describe two simple faults on the BEQ instruction. Then, we display some faulty behaviours that we obtained for different instructions. They are shown in Table 1, as well as the origin of the error, i.e., where the fault was injected. We then describe a few more complex faulty behaviours. Finally, from the different behaviours obtained in simulation, we discuss the relevance of typical software fault models.

4.1. Faults on a BEQ instruction

A simple fault on BEQ can prevent the branch from being taken, regardless of the result of the comparison. By faulting the "branch" signal, we can make sure that the multiplexors in the MEM stage select the default entry (value 4). In this case, the branch target is thus PC+4 whatever the result of the comparison. Hence, the branch is never taken.

Another fault can modify the comparison in the EX stage by faulting the selection signal of one of the two multiplexors "Mux_1" or "Mux_2". Thereby, the ALU compares a GPR to 0 instead of comparing the two GPRs. This fault has a side consequence: pseudo-instructions like "BEQZ" (branch if equal to zero) can be forced to branch. Indeed, in this case, one argument is already equal to zero, and we can inject the fault described to set the other argument to zero, and thus make the equality true.

These two faults are quite straightforward. More interesting faulty behaviours are shown in Table 1, for different classes of instructions.

4.2. *More complex faults*

In this section, we study in more detail the behaviours that are numbered in Table 1. These paragraphs give precisions, side effects or consequences of these faults. In particular, we show how they can interact with some optimisation structures in the pipeline, namely forwarding and speculative execution. These structures, which have a clear impact on the processor performance, also make the study of fault effects more complex.

(1) First, an interesting fault uses the forwarding capability of the processor. In Table 1, the fault *R-type/write_enable* can prevent a result from being committed into the register-file. However, in case there is a data hazard on this faulted instruction, the forwarding can still happen with the correct value. This is due to the fact that forwarding happens before the value is written into the register-file. The forwarding can be deactivated or not, depending on when the attack is performed (EX, MEM or WB stages). For the sake of clarity, we quickly examine the code example in Listing 1. By attacking the first instruction, an attacker can prevent the processor from writing the result of the addition into a2. Thus, a2 keeps its previous value. But the value computed by the second instruction can still be correct; as if the fault on a2 did not happen (i.e., a3 would be equal to a0+a1+a0).

This behaviour can have some interesting uses. For example, in the next section, we show that it is possible to pass through a comparison with a wrong value.

There are other uses of forwarding in the context of fault injection. It is possible to activate forwarding when it should not be used (or use it to forward the wrong value). As a result, it is possible, for any instruction, to replace one of its arguments (that should be read from the register-file) by the result of the previous instruction or the instruction before. Contrary to the preceding fault, where forwarding was used passively, as a side consequence of the code; here we use forwarding actively.

(2) A second interesting fault can allow an attacker to execute an instruction following a jump or a branch instruction. This is due to speculative execution. When the processor realises that wrong instructions have been speculated, it deactivates these instructions. It is possible to inject a single-bit fault to re-activate one of these instructions, and allow it to finish executing. More precisely, in case of a fault attack, the first or the third instruction of the wrong branch can be executed. For example, in Listing 2, instructions 11 or 13 could be executed before instruction 1; and conversely, instructions 1 or 3 could be executed before instruction 11. This fault can be used to execute an instruction that is not intended in a particular context.

```
    BEQ a1, a2, label;
    Inst_1
    Inst_2
    Inst_3
    [...]
Label:
    Inst_11
    Inst_12
    Inst_13
    [...]
```

Listing 2: Example assembly code

(3) Another fault consists in writing in a GPR during a branch instruction (*Branch/write_enable*). The value written is the result of the comparison: 0 if it is false; 1 if it is true. Target GPR depends on the offset of the branch. Only GPRs whose number is a multiple of four or GPRs equal to 1 modulo 4 can be targeted (otherwise, there is a problem in the code since the offset results in a misaligned PC). Indeed, the target GPR is selected by interpreting some bits of the offset of the branch. It is a consequence of the way instruction words are designed in the ISA.

A similar fault can be executed on R-type instructions: *R-type/branch*. In some way, it could be considered as the

symmetric of the previous one. With this fault, the instruction is correctly executed, but there is also a branch to another part of the program, depending on the operation and the destination GPR. This fault can only be exploited when the result of the ALU is odd. To have a correct jump target, GPRs multiple of four or equal to 1 modulo 4 have to be used (others raise an exception for misaligned PC). So these GPRs are more susceptible to faults.

These two faults execute a hybrid instruction that executes both the specified instruction, and a side effect from another instruction. Both faults put the processor into an intermediate state that is neither a branch nor an R-type instruction, but something in-between.

(4) It is possible to fault the operation executed in the ALU by targeting the ALU_op signal. This type of fault looks like an instruction replacement that could be analysed directly with the instruction word, but it is different. ALU_op is an internal signal, and thus, its mutations are different from those in an instruction word.

"ALU_op" is a four bits wide register, so an array of the operations ordered in Gray code (Table 2) can be used to easily see all single-bit mutations for each operation, by looking at adjacent cells. For example, this array shows that we can change an addition (add) into a XOR, a left shift (sl), a "set if equal" (seq) or even an unrecognized operation.

LSB→ / MSB	00	01	11	10
00	add	sl	-	-
01	xor	sr	and	or
11	slt	sge	sgeu	sltu
10	seq	sne	sra	sub

Table 2: ALU operations ordered in Gray code. Operations 2 and 3 are not specified in the RTL description (they should not happen), but would in most cases result in 0 and 1, respectively.

There are two interesting things about faults on the ALU_op signal. First, we can see that every comparison operation is right next to its inverse (seq next to sne; slt next to sge; sltu next to sgeu). This is an efficient way to design the ALU, but it is unfortunately also more vulnerable to single bit faults. A single-bit fault can easily reverse the condition of a test (note that the test inversion behaviour is already known in typical software fault models; here we see a justification of this model). The second interesting thing about this table is that any arithmetic or logic operation can be transformed into a comparison. As a consequence, the result of the ALU can easily be set to 0 or 1 instead of the correct result.

4.3. Discussion

This study has shown that when examining the microarchitecture of a processor, in presence of a single bit-flip, we can find unexpected behaviours that are not covered by typical software fault models. These faulty behaviours can put the pipeline into an intermediate state, where the control signals do not perfectly match any instruction. This is perhaps the main difference compared to typical software fault models. In typical software fault models, instructions are often replaced by another one (instruction skip, test inversion, instruction replacement) or some data are corrupted (in memory or the register-file). All these behaviours, even if faulty, are still valid (in the sense that the processor could theoretically reach these states). Here, the processor is put in an unknown state which can have multiple effects and which is impossible to anticipate at the software level. For example, the fault *R-type/write_enable* looks like a typical instruction skip, i.e. a replacement with a NOP. However, it has a side effect (forwarding) that can impact security and that is not considered in any software fault model (as far as we know). Another important example is *Branch/write_enable* which modifies a register after the comparison. Its side effect can have a huge impact on security since comparison instructions are at the heart of many security measures. Even if these processor states are not attainable in a normal execution, it is still possible to model them at the software level, by adding different instructions. Examples are shown in next section, to model attacks on various countermeasures.

Some of the faults shown here are only activated when specific conditions are met. Thus, they can be easily avoided by taking into account these characteristics when designing the software. Conditions can be on the ordering of instructions (forwarding faults), on the values manipulated (branching when the result is odd), or the offset of a branch for example. These faults also show that some values or GPRs are more prone to errors than others. It is possible to inject a fault to force an argument to 0 or to force the result of the ALU to 0 or 1. These values should be handled with caution in an application. The use of Hardened Booleans (Booleans whose values are different from 0 and 1; for example 0x55 and 0xAA), as defined in [22], would be a good practice in the considered processor. Likewise, GPRs multiples of four seem more susceptible to faults, so they should be used with that in mind when designing software. All these remarks lead to the idea that to assess system security, hardware and software should be analysed together. This co-analysis could detect potential vulnerabilities or lack thereof.

5. Software countermeasure analysis

To prevent faults from impacting the execution of a program and creating vulnerabilities, many software countermeasures have been developed over time. In this section we analyse some of these countermeasures and see how they handle precise faults extracted from the processor microarchitecture in section 4.

In [23], the authors did an extensive evaluation of the efficiency of software countermeasures in simulation. This is a quantitative evaluation of the consequences of

faults in protected software. We propose here to do a qualitative evaluation using the knowledge we have from previous section. We will be able to understand precisely why some countermeasures do not necessarily work.

Software countermeasures can be divided into two categories: those that target data-flow integrity and those that target control-flow integrity. The former consist in ensuring that data are manipulated correctly while the latter consist in ensuring that no spurious jumps can corrupt the system.

The purpose of this last section is to show that single-bit faults found in previous section can be used to defeat typical countermeasures while corrupting protected data.

5.1. Data-flow integrity

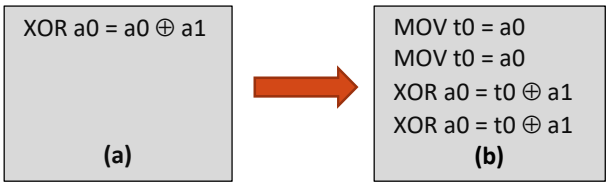
Data-flow countermeasures often consist in adding redundancy in the code. Redundancy can be used to detect, or even correct a fault. Here, we first examine the duplication scheme described in [24], which aims at ensuring a correct code execution, even in presence of an instruction skip. Then, we examine the duplication and triplication proposed in [25]. The schemes in this second paper can be used against more general fault models: instruction skips or any kind of data or computation corruption. However, the duplication can only detect and not correct a fault, contrary to [24].

In these schemes, redundancy is used at instruction level. There is also a common countermeasure that is algorithm duplication. In that case, the whole algorithm is executed, and then executed a second time (the second execution can differ from the first; for example, it can consist in ciphering and then deciphering to verify that there was no problem). However, as pointed out in [25], it is harder to inject the same fault in two subsequent instructions than it is to inject them in two executions of an algorithm. The former requires high-end means. That is why it can be preferred. We show however that they are not immune against the faulty behaviours we have extracted in section 4.

5.1.1. Duplication

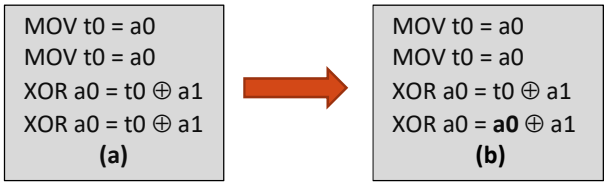
In [24], Moro et al. proposed a countermeasure to thwart instruction skip attacks. This countermeasure consists in duplicating every instruction, so that if one of the two instructions is skipped, the other one still manages to compute the correct result. Idempotent instructions are simply duplicated, while non-idempotent instructions first need to be replaced by an idempotent sequence. Listing 3 shows how a non-idempotent XOR is protected (the instruction in Listing 3.a is not idempotent because executing it twice changes the result).

In their paper, the authors formally prove that this countermeasure is resistant against instruction skips. This is true for direct instruction skips. However, other attacks in the hardened code can behave like if there was an instruction skip in the original code. In the protected



Listing 3: (a) Unprotected xor instruction; (b) Xor protected with Moro countermeasure

sequence in Listing 4, it is possible to do a forwarding attack on the last XOR, so that the first argument is replaced by the result of the previous instruction. In that case, the final result is equal to the initial value (XORing twice with the same value is equivalent to doing nothing). Hence, this fault in the protected code is equivalent to an instruction skip in the unprotected code.

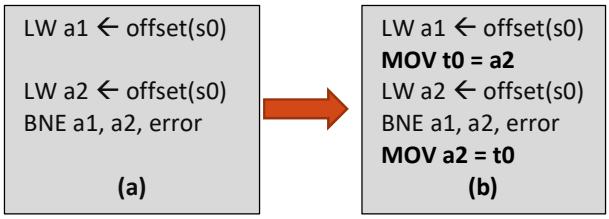


Listing 4: Forwarding attack on a duplicated XOR

5.1.2. Duplication - Comparison

In [25], Barenghi et al. propose different methods to protect a code by adding redundancy.

One of their countermeasures consists in duplicating an instruction and then comparing their results. We examine this countermeasure in the case of a load instruction. Listing 5.a provides a code example.



Listing 5: *Load/write_enable* attack on load duplication. BNE stands for "branch if not equal". The attack is modelled by saving the value of register a2 and then restoring it after the comparison.

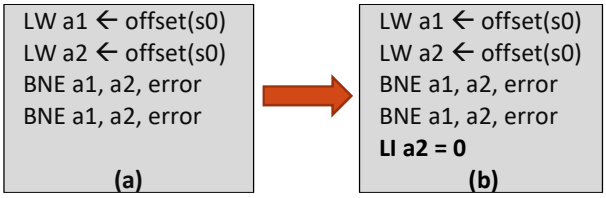
Offset(s0) is the memory address (addition of an offset to the content of GRP s0). The content of this memory address is read twice, and stored in GPR a1 and a2. Then there is a comparison between these GPR and a jump to label "error" in case they are different. Against this countermeasure, an attack can be performed on the second load. Indeed, with the fault *Load/Write_enable* in Table 1, it is possible to prevent a load from writing into the destination GPR. But due to the forwarding, which is activated in the final BNE instruction, the correct value is used in this instruction. Thus, the value read from memory is directly forwarded to the comparison instruction, and this value (which is the correct one) is

compared with a1. As both values are the same, the program does not detect any error. However, a2 keeps its previous value. Listing 5.b shows how the code behaves under the attack. We added two instructions to the original code (in bold in Listing 5.b): one for storing the initial value of a2 in an unused temporary GPR before the load modifying a2, and one for restoring this value into a2 after the comparison. It is interesting to note that the single hardware fault results in two distinct effects in the code: corrupting an instruction and bypassing another at the same time.

5.1.3. *Duplication and double comparison*

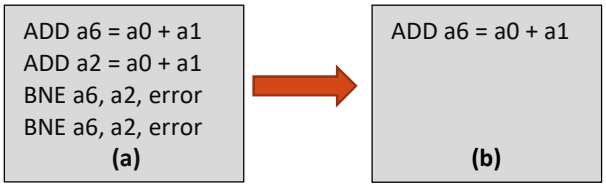
One criticism of the previous countermeasure could be that only the instruction to protect is duplicated, and not the comparison. A duplication of the comparison, as shown in Listing 6.a, could prevent the attack shown, because a load instruction can only forward its value to the next cycle.

However, the completely duplicated countermeasure is not entirely safe either. The *Branch/write_enable* fault can modify a GPR during a branch operation. a2 could be this GPR, depending on the offset of the branch instruction (as described in section 4.2). Listing 6.b shows how this fault could behave if a2 were impacted. Both comparisons are executed normally, and right after that, a2 is modified. It is important to emphasize that: all the code is executed normally, but the last comparison changes the value. With more comparisons, the result would be the same.



Listing 6: *Branch/write_enable* attack on load duplication/comparison. a2 is modified if the error label is at the right location. Otherwise, another GPR would be targeted. "LI" is short for "load immediate".

One interesting thing to note is that the efficiency of the countermeasure depends on the protected instruction. For the same countermeasure, different instructions would need different attacks. Here we have presented two attacks on a load operation, but if we were protecting an addition instead, different attacks would also be possible. For example, the forwarding attack would still be possible with the duplication of the comparison, because the forwarding can happen during two cycles, as explained in section 3. Another possible attack would be to skip all the duplicated section of the code (*R-type/branch*), thus leaving a single ADD instruction. This example is presented in Listing 7. However, this attack could only work under specific conditions (the result of the addition is odd, and its destination GPR is a6 for example).



Listing 7 : *R-type/branch* attack on add duplication. Three instructions are skipped because a6 is the destination GPR. Another destination would skip a different number of instructions.

5.1.4. *Triplication*

The next countermeasure studied is the triplication of an instruction, coupled with a majority vote. The advantage of this countermeasure is that it can correct single software faults (instruction skip or data/computation corruption). But this feature, which improves fault robustness, can also become the source of new fault attacks. An attacker could use this feature to correct a value into a wrong result. We saw in Table 1 that there are multiple ways to modify the contents of the memory during a load operation (faults *Load/Mem_cmd*). A fault during the first load operation can modify the memory. So, the next two load operations read the corrupted value. Amongst the three load instructions, the second and third read the same (wrong) value. Then, because the majority of reads have the same corrupted value, the countermeasure considers that it is the correct one. What is interesting here is that if the triplication were only used to detect faults, the attack would not be possible. Allowing the code to correct faults can be counter-productive.

This attack would not work if the memory were also triplicated. However, this would need a triplication of the preceding store operation, which also has vulnerabilities. A faulty store instruction can modify a GPR (*Store/Write_enable*). So if the first store operation modifies its GPR, the two following operations write a wrong value in memory.

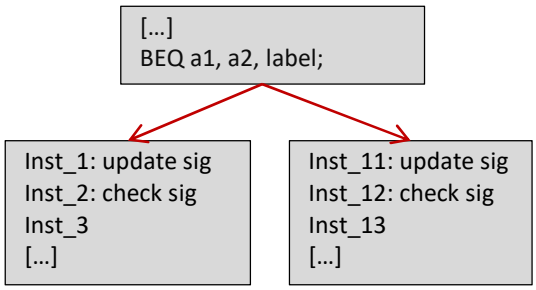
5.2. Control Flow Integrity

5.2.1. *Overview*

Aside from all previous countermeasures that aim at protecting the data-flow integrity, there is also another kind of countermeasures that target control-flow integrity. Control-flow integrity countermeasures are designed to ensure that no spurious jumps can happen during code execution. When studying control-flow integrity, the code is first separated into basic blocks. These are pieces of code where no jump can happen (as origin or target of the jump), except at the beginning or at the end. Control-flow integrity can be divided into two problems: inter-block and intra-block jumps. The former refers to jumps from one basic block to another, wrong, one, while the latter refers to jumps inside the same basic block (which is

inherently wrong since basic blocks are jump-free by definition).

Protecting against intra-block jumps is difficult to do in software since the finest granularity level is the instruction: we need to add instructions to ensure that each instruction is executed... The common countermeasure is to increment a counter after each instruction and check its value at the end of the basic block. On the other hand, inter-block jumps have been subject to a lot of research. Many different schemes exist to protect against these attacks. They are mainly based on the computation of a run-time signature which is checked against a pre-computed signature, in each basic block. [26] surveys these schemes and compares their detection ratio. Most of the time, some instructions are added at the beginning and/or the end of a basic block to update the signature and check its value (not necessarily in this order). Listing 8 shows how the protected code is structured. After the BEQ instruction, there are two basic blocks (one if the condition is true; the other if it is false). In each basic block, we add two instructions to update and check the run-time signature.



Listing 8: Code example with control-flow integrity countermeasure

5.2.2. Attack on inter-block control-flow integrity

The schemes used in inter-block control-flow integrity are designed to thwart "direct" jumps that result for example from a modification of the PC. However, the control-flow can also be attacked by more complex faults that we have shown in section 4. Using speculative execution, it is possible to commit the third instruction of the wrong branch. So if the control-flow integrity countermeasure only adds two instructions at the beginning of a basic block, the attack can reach an instruction that should have been protected.

In Listing 8, it is possible to execute inst_13 before going into the left basic block, and likewise, it is possible to execute inst_3 before going into the right basic block. In either case, there is a violation of the control-flow that is not detected by the countermeasure.

5.3. Discussion

In this section, we have seen that it is necessary to exercise caution when using typical software countermeasures. While certainly effective at thwarting general software fault models, they can be ineffective or even detrimental to system security when considering actual faults.

The question that arises from all this study is: is there a way to protect a code effectively against the complex faulty behaviours our analysis describes? As we have seen in Table 1, various complex faulty behaviours can happen in a processor, and these behaviours are often very different from each other. In addition, some of these faulty behaviours can happen only under specific circumstances. It is possible to give some general advices to protect a program, like the ones shown in section 4 (using hardened Booleans, avoiding common values like 0…). Another example is to add dummy operations in order to lessen the impact of some structures like forwarding or speculative execution (note that this advice goes exactly against the principles behind these optimizations: they are built to avoid wasting cycles, but for security reasons, it is better to reintroduce those wasted cycles). Indeed, these dummy operations can reduce dependences between instructions, thereby preventing some complex faulty behaviours.

While these advices can help, finding a general software countermeasure that can effectively counter every faulty behaviour in every circumstance seems to be very difficult. Making hardware and software countermeasures interact with each other could be a good way to improve the overall security (again, this is a hint to use a cross-layer approach). However, the end goal in security is not to counter every fault, but those that give the attacker an advantage. In the end, the design of software countermeasures should not rely "blindly" on the microarchitecture and the software, but should be geared towards specific security goals.

6. Conclusion and perspectives

In this paper, we have shown some simulated faulty behaviours that can be observed when injecting single-bit faults into a LowRISC v0.2 processor. These faults were injected in unprotected sections of the processor pipeline, and created behaviours that are not thwarted by software countermeasures either. This was shown on several countermeasures targeting either data-flow or control-flow integrity.

Most of these faults are difficult to predict without undertaking a precise analysis of the RTL architecture of the processor. The ISA level is not sufficient to really understand how the processor behaves under an attack. An analysis of processor microarchitecture can bring realism to software fault models, and that knowledge can in turn be used to design better countermeasures to enhance system security.

Some faults depend heavily on the software context; the forwarding fault and the speculative execution fault in particular, which depend on the previous or the next instructions to execute. Some faults have different effects

depending on the values manipulated, or the GPRs used. These observations lead to the idea that security assessment should rely conjointly on a hardware and software cross-layer analysis.

Several perspectives come out of this study. First, it would be interesting to study the impact of faults on an actual application. Indeed, some single-bit faults could have interesting effects that cannot appear on isolated countermeasures. Another perspective would be to have a methodology to automatically model faulty behaviours from the RTL description of a processor. This search for exhaustiveness could lead to the design of more effective software countermeasures. Another interesting perspective would be to take into account multi-bit faults and see which behaviours can be obtained under this more general fault model. To cope with the huge fault space of multi-bit attacks, the methodology described in [27] would be a good starting point.

Finally, we have pointed out the idea that countermeasures should primarily be designed to thwart attacks on specific security goals instead of trying to counter every possible attack. To do that, some techniques from static code analysis can be exploited. Indeed, they can be used to prove the correctness of properties in the code. So we could use them to ensure that security properties in the code are unharmed against faults extracted from the microarchitecture. The use of static analysis to automatically detect vulnerabilities in a system is the focus of our current work.

References

[1] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," in *Advances in Cryptology — EUROCRYPT '97*, 1997, pp. 37–51.

[2] Joint Interpretation Library, "Application of Attack Potential to Smartcards." Jan-2013.

[3] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky, and J. F. Lalande, "High Level Model of Control Flow Attacks for Smart Card Functional Security," in *2012 Seventh Int. Conf. on Availability, Reliability and Security*, 2012, pp. 224–229.

[4] G. Barthe, F. Dupressoir, P.-A. Fouque, B. Grégoire, and J.-C. Zapalowicz, "Synthesis of Fault Attacks on Cryptographic Implementations," presented at the ACM CCS 2014, 2014, p. 16.

[5] A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner, "QEMU-Based Fault Injection for a System-Level Analysis of Software Countermeasures Against Fault Attacks," in *2015 Euromicro Conference on Digital System Design*, 2015, pp. 530–533.

[6] E. Touloupis, J. A. F. Member, V. A. Chouliaras, and D. D. Ward, "Study of the Effects of SEU-Induced Faults on a Pipeline Protected Microprocessor," *IEEE Trans. Comput.*, vol. 56, no. 12, pp. 1585–1596, Dec. 2007.

[7] H. Cho, S. Mirkhani, C. Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–10.

[8] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *International Conference on Dependable Systems and Networks, 2004*, 2004, pp. 61–70.

[9] J. Espinosa, C. Hernandez, and J. Abella, "Modeling RTL fault models behavior to increase the confidence on TSIM-based fault injection," in *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2016, pp. 60–65.

[10] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2013, pp. 77–88.

[11] L. Dureuil, M.-L. Potet, P. de Choudens, C. Dumas, and J. Clédière, "From Code Review to Fault Injection Attacks: Filling the Gap Using Fault Model Inference," in *Smart Card Research and Advanced Applications*, 2015, pp. 107–124.

[12] L. Dureuil, "Analyse de code et processus d'évaluation des composants sécurisés contre l'injection de faute," phdthesis, Communauté Université Grenoble Alpes, 2016.

[13] M. S. Kelly, K. Mayes, and J. F. Walker, "Characterising a CPU fault attack model via run-time data analysis," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2017, pp. 79–84.

[14] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software Fault Resistance is Futile: Effective Single-Glitch Attacks," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2016, pp. 47–58.

[15] J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, "On the Importance of Analysing Microarchitecture for Accurate Software Fault Models," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, 2018, pp. 561–564.

[16] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proc. IEEE*, vol. 94, no. 2, pp. 370–382, Feb. 2006.

[17] J. B. Machemie, C. Mazin, J. L. Lanet, and J. Cartigny, "SmartCM a smart card fault injection simulator," in *2011 IEEE International Workshop on Information Forensics and Security*, 2011, pp. 1–6.

[18] H. Ziade, R. Ayoubi, and Velazco, "A Survey on Fault Injection Techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, Jul. 2004.

[19] "The RISC-V Instruction Set Manual," *RISC-V Foundation*. [Online]. Available: https://riscv.org/specifications/. [Accessed: 28-Mar-2018].

[20] "lowrisc-chip: The root repo for lowRISC project and FPGA demos," *Github*, 16-Jun-2018. [Online]. Available: https://github.com/lowRISC/lowrisc-chip. [Accessed: 19-Jun-2018].

[21] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

[22] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, "FISSC: A Fault Injection and Simulation Secure Collection," 2016, pp. 3–11.

[23] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 404–409.

[24] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, "Formal verification of a software countermeasure against instruction skip attacks," presented at the PROOFS 2013, 2013.

[25] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures against fault attacks on software implemented AES," 2010, p. 7.

[26] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random Additive Signature Monitoring for Control Flow Error Detection," *IEEE Trans. Reliab.*, vol. 66, no. 4, pp. 1178–1192, Dec. 2017.

[27] A. Papadimitriou, D. Hély, V. Beroulle, P. Maistri, and R. Leveugle, "A multiple fault injection methodology based on cone partitioning towards RTL modeling of laser attacks," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2014, pp. 1–4.