Bridging the gap between RTL and software fault injection

J. Laurent, C. Deleuze, F. Pebay-Peyroula, V. Beroulle

Bridging the Gap between RTL and Software Fault Injection

J. Laurent

LCIS, Grenoble Institute of Engineering (INP), Univ. Grenoble Alpes, Valence, France, johan.laurent@lcis.grenoble-inp.fr

C. Deleuze

LCIS, Grenoble Institute of Engineering (INP), Univ. Grenoble Alpes, Valence, France, christophe.deleuze@lcis.grenoble-inp.fr

F. Pebay-Peyroula

CEA-LETI, Univ. Grenoble Alpes, Grenoble, France, florian.pebay-peyroula@cea.fr

V. Beroulle

LCIS, Grenoble Institute of Engineering (INP), Univ. Grenoble Alpes, Valence, France, vincent.beroulle@lcis.grenoble-inp.fr

ABSTRACT

Protecting programs against hardware fault injection requires accurate software fault models. But typical models such as the instruction skip do not take into account the microarchitecture specificities of a processor. We propose in this paper an approach to study the relation between faults at the Register-Transfer Level and faults at the software level. The goal is twofold: accurately model RTL faults at the software level, and materialize software fault models to actual RTL injections. These goals lead to a better understanding of a system's security against hardware fault injection, which is important to design effective and cost-efficient countermeasures. Our approach is based on the comparison between results from RTL simulations and software injections (using a program mutation tool). Various analyses are included in this paper to give insight on the relevance of software fault models, such as the computation of a coverage and a fidelity metric; and to link software fault models to hardware RTL descriptions. These analyses are applied on various single-bit and multiple-bit injection campaigns to study the faulty behaviors of a RISC-V processor.

CCS CONCEPTS

• Security and privacy~Security in hardware~Hardware attacks and countermeasures

KEYWORDS

Fault injection, Fault modeling, Software injection, RTL injection

1 Introduction

Since the Bellcore attack [5], fault injection has been a threat to digital systems. Various means of attack (for example, clock and power glitches, electromagnetic pulses, laser injection) have been used to disrupt the execution of a program in order to bypass security measures, or to expose secrets, such as cryptographic keys. To protect systems against fault injection, various countermeasures have been developed in the past, both on the hardware [2] and on the software side [18]. Generally speaking, hardware countermeasures are effective but involve a big overhead, and the added cost can be prohibitive in some systems. At the other end of the spectrum, software countermeasures can also be used. They aim at thwarting the consequences of the faults in the program. They are often less costly and

more flexible since they can be adapted to particular threats on specific programs. However, they rely on the ability of software fault models to accurately represent how the processor behaves when attacked.

Over the years, various software models have been designed, the most prominent ones being the instruction skip, instruction replacement, test inversion and register corruption. These models are used in many research papers, such as [15][1][16][4]. However, they may poorly represent realistic faulty behaviors. In [7], the authors showed that injecting faults in flip-flops of the pipeline of a processor produces behaviors that can be vastly different from injecting in the register-file or variables of a program, which are faults of a higher abstraction level. In [3], Bergaoui et al. also showed that faults injected in the register-file do not necessarily have much impact due to the importance of hidden registers of the processor microarchitecture. Finally, in [11], we showed that when looking at the microarchitecture of the attacked processor, attacks can be carried out in such a way that cannot be predicted by typical software fault models. As a result, software countermeasures designed against them do not necessarily thwart these attack scenarios.

These limitations in typical software fault models are due to the gap that exists between the software view of a program execution, and its actual hardware execution. The many optimizations of modern processors (pipelining, forwarding, speculative execution, etc) make the execution much more complex than in the software view (where instructions are simply considered atomic), and make software fault modeling more difficult. The gap in abstraction level leads to two caveats in software fault modeling. The first one is about *coverage*: software fault models do not predict every fault effect that can happen in a processor, and ignored effects can threaten the security of the system. The second one is about *fidelity*: effects predicted by software fault models sometimes do not represent effects that can actually happen in the processor; in that case, a developer might waste time protecting against an effect that cannot actually happen in reality, and add unnecessary countermeasures. Having a high-coverage and high-fidelity set of software fault models is important to conduct effective security analyses at the software level. To design this kind of models, the microarchitecture of the processor must be considered.

Several research papers already mention the characterization of faulty behaviors of a specific processor. Their goal was to find out the software fault model(s) that best explained the results they got on a specific processor implementation. In [1], Balasch et al. performed a characterization of the faulty behaviors happening when injecting clock glitches into an 8-bit AVR microcontroller. They concluded that the most likely fault model was the instruction replacement. In [17], Proy et al. injected electromagnetic pulses in an out-of-order superscalar processor running various characterization programs. Amongst the observed behaviors, they found the instruction skip and register corruption, but also combinations of these and more complicated effects. In [10], Dureuil et al. proposed a method to iteratively build fault models from injection experiments. These software fault models consisted in a set of replacements of values loaded by the program. Their goal was not only to model faults, but also to directly link them to injection parameters of their injection platform. This dual goal is also what we aim in our approach.

In all these studies, the processor was considered as a black-box: its inner working was not available, so the models had to be inferred from injection results only. However, with the release of RISC-V, an open-source Instruction Set Architecture, new open-source implementations of processors have been developed, and these RISC-V processors are being more and more adopted. This is interesting regarding security since having access to the inner working of the processor allows to study more accurately the impacts of faults, and in turn to design better software fault models.

We therefore consider the RTL description of the processor available. We can then perform injections in RTL simulations. These injections are less “realistic” than physical injections because of optimizations during the synthesis of the processor, and because of the need to rely on a RTL fault model. However, they still give relevant results earlier in the design flow [14], which means that potential vulnerabilities can be detected and corrected earlier, limiting additional costs and design respins. Furthermore, they offer more controllability and observability than physical attacks; it is easier to inject at a precise point in

space and in time, and it is easier to observe the internal states of the processor. These are desirable characteristics to precisely understand how faults disrupt a program execution.
In this work, we propose an approach based on the comparison between RTL injections and software injections, to help a hardware/software microprocessor designer study the impact of faults early in the design flow. Comparisons between RTL injections and various software fault models have already been studied in the past, such as in [6], but these software fault models are usually limited in their accuracy; for example they consider random bit-flips in data. We aim at giving a more precise description of the behavior of the attacked processor at the software level. Note that we consider the software part to be the binary of an application, rather than its source code, to avoid dealing with compiler optimizations which are not in our scope: we only focus on the actual code run by the processor.
More generally, our approach is a tool to better understand the relation between the RTL and software abstraction levels. This relation goes both ways, as seen in Figure 1. Going from RTL to software level is the Abstraction process: the goal is to model faults at the software (assembly) level, so that they represent what happens in RTL simulations. Going from software level to RTL is the Materialization process: the goal is to link software fault models to actual hardware structures in the processor, for example to realize in a RTL simulation the effects predicted by a software fault model, or to decide where to add hardware countermeasures.

Figure 1 : Two-way relation between RTL and software level

The contributions of this paper can be summed up in these three points:

- Computing a coverage metric and a fidelity metric to evaluate software fault models, and other analyses to help to model faults in the Abstraction process.
- Giving information about the Materialization process, in two ways. Drawing model profiles to help identify which flip-flops need to be faulted to realize a software fault; and drawing a diagram to help decide where to add hardware countermeasures.
- Applying all these analyess to study the faulty behaviors of a RISC-V processor executing various characterization programs.

The paper is structured in the following way. Section 2 presents an overview of the methodology as well as some specifications about the approach. Section 3 discusses the processes of RTL and software injections. Section 4 introduces a mathematical formalism to describe fault injections, and presents various ways to evaluate the relevance of software fault models. Results of fault injection campaigns and their analysis are discussed in Section 5. Finally, Section 6 concludes and mentions some perspectives.

2 Overview of the approach

The general view of the approach is shown in Figure 2. The approach can be divided into three distinct parts: RTL injections, software injections and then comparison and analysis of the results. The same characterization programs are used for both injection methods. They are made to exercize various structures in the processor microarchitecture, in order to discover more fault effects. The analysis and metrics part is not detailed in the diagram because it encompasses many methods and is the subject of the whole section 4.

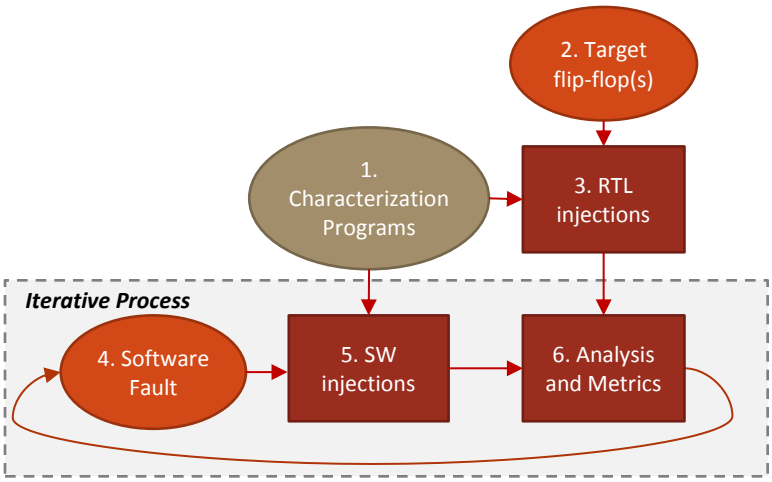

Figure 2 : Overview of the approach

2.1 Objectives of our approach

In this section, we give a broad view of our approach. Each of the points discussed here are explored in more depth in the rest of the paper.

As mentioned in the introduction, the first goal of the approach is the Abstraction process; i.e. to model RTL faults at the software level. In this regard, comparing injection results allows computing metrics about the relevance of fault models, such as coverage and fidelity. The coverage metric shows the proportion of RTL injections that are predicted by software models. The fidelity metric is the reverse: it shows the proportion of software injections that correspond to actual effects in the processor. This second metric can be useful since software fault models are not 100% accurate: they may predict cases that don't exist in reality. The precise definition of these metrics is discussed later in the paper. Finally, comparing injection results also allows easily finding which kinds of faults are not already covered by software fault models. This is helpful information to help proposing new models or improving existing ones.

With these metrics, modeling faults at the software level can be seen as an iterative process, shown in Figure 2. It can be divided in several steps:

1. Specify characterization programs.
2. Specify flip-flop(s) to target for RTL injections.
3. Perform RTL fault injections during the execution of the characterization programs.
4. Specify software fault models by studying RTL fault propagation in the microarchitecture.
5. Perform software fault injections in the characterization programs, according to the specified software fault models.
6. Compare the results of RTL injection and software injection to compute metrics about the relevance of the software fault models. If results are not good enough, return to step 4.

The second goal of the approach is to help the Materialization process. Given a software fault model, how to realize an actual attack that leads to the same effects? Which flip-flop(s) should be faulted? These questions are useful for example to verify that an attack found with a software fault model is real; it is especially useful for software fault models with a low score on the fidelity metric. It is also interesting to give an idea of size of the attack area of a particular attack. The Materialization process has another objective, which is pinpointing the flip-flops that create the most faults not covered by software fault models. This is helpful to choose which flip-flops should be protected in priority at the hardware level.

To achieve these goals, we need to design a method to precisely compare RTL injections to software injections. To have relevant comparison points, we first need to define the circumstances of injection and observation that both RTL and software injections must follow. We explore these in the next two sections.

2.2 Injection specification

A key aspect in our approach is to precisely define the circumstances of injection. In which instruction of which program should we inject faults? Complex faulty behavior are very dependent on the execution environment. We therefore used different kinds of characterization programs made to exercise different structures in the processor. Modeling effects against a single program could lead to an incorrect evaluation of the faulty behaviors. Our characterization programs, written in assembly, are made of three parts: a prologue, a target and an epilogue. The target is the instruction that is targeted during the injections while prologues and epilogues are sets of instructions that represent various situations before and after the target instruction. The structure of characterization programs can be seen in Listing 1.

```
Prologue_instruction_1
Prologue_instruction_2
Target_instruction       // Injection in this instruction
Epilogue_instruction_1
Epilogue_instruction_2
Epilogue_instruction_3
```

Listing 1: Example characterization program

The choice to divide the characterization programs into three parts was made to reduce the aforementioned dependency of results on the input programs. Changing the prologue modifies the internal state of the processor right before the injection, leading to different behaviors. Changing the epilogue on the other hand modifies how the fault propagates in the processor. Fault injection sometimes creates latent effects in the processor that only become visible under specific epilogues.

2.3 Observation specification

Another key aspect is to define what to observe, and when to observe injection results.
In the software side, only the register-file and memory can be observed (contrary to RTL where we can observe any signal in the processor). We therefore only observe fault effects in these two structures.
For the question of when to observe, the hardware side is more complex to deal with because of latent effects. In software, defining the state of memory and register-file at a certain point in the execution is enough to completely describe the state of the execution. In hardware however, the state of the processor is also important because latent effects introduced by a fault can become visible later in the register-file or memory. So, if the observation is made too early, the injected fault does not have time to fully propagate and potential latent effects cannot be seen. If the observation is made too late however, some effects can be masked by instructions of the epilogue. To alleviate the approximations associated with each option, we decided to define two observation points, which are both used for the comparisons between RTL and software injection: one right after the execution of the target instruction (instantaneous effects) and one right after the execution of the last instruction of the epilogue (propagation effects). These observation points are easy to define at the software level since instructions are executed atomically; but slightly more complex in a pipelined processor where instructions are executed in several cycles. For the RTL side, we considered the instructions done one clock cycle after going through the last pipeline stage. The two observation points can be seen in Listing 2.

```
Prologue_instruction_1
Prologue_instruction_2
Target_instruction       // First observation after execution of this instruction
Epilogue_instruction_1
Epilogue_instruction_2
Epilogue_instruction_3   // Second observation after execution of this instruction
```

Listing 2: Observation points in a characterization program

3 Fault injection methodologies

In this section, we present the fault injection methodologies used in our study. First, RTL fault injection is explained, then the software injection method is exposed. To stay generic, we only discuss here the ideas behind our methodologies. The specific flip-flops, software fault models and characterization programs that are used are described in a later section, once we present a use-case of our approach.

3.1 RTL fault injection

3.1.1 Parameters of RTL fault injection

To specify an RTL fault injection methodology, we need to define which RTL fault model is used, how it is used in simulation, where and when the fault is injected, and finally when and where the results are observed. We answer these questions in the following:

- The RTL fault model is the bit-flip, either single-bit or multiple-bit. The bit-set or bit-reset models could also be used, but since our goal is to study what kind of effects can happen, we want to generate many different faulty behaviors; hence the bit-flip model is more appropriate.
- Faults are injected by forcing signals thanks to simulator commands available in QuestaSim.
- To generate lots of faulty behaviors, we target the most sensible part of the processor: its pipeline. We do not inject fauts in the register-file and memory.
- We want to inject faults in the target instruction of the characterization program (as defined in previous section). However, since the execution is pipelined, the injection could correspond to various instants. To solve this issue, each flip-flop in the processor was associated to one pipeline stage (or in some cases multiple stages), depending on when this flip-flop is read. An injection in a flip-flop can only happen when the target instruction is in the associated pipeline stage.
- The two observations (instantaneous effects and propagation effects) are performed when the associated instruction leaves the pipeline (as defined in previous section)
- Both the register-file and memory are observed. The state of the register-file is simply dumped at the two observation instants. For memory, we monitored the writing transactions issued between the injection and the observation point.

To detect the various injection/observation instants, we relied mainly on internal signals: the instruction address in each pipeline stage, as well as validation signals that tell which pipeline stages are active. Faults sometimes corrupt these signals; in that situation, two things can happen. If the instruction where observation should be performed is detected in an early stage, we consider that each pipeline stage takes one cycle to complete. If the instruction is not detected at all (due to an abnormal jump provoked by a fault for example), no observation is made, and the simulation is tagged as “unknown” (see next section). This strategy is a trade-off between result accuracy and complexity: it is difficult to devise a perfect strategy to automate precise observations when faults can corrupt the execution flow in unpredictable ways.

Finally, note that in order to avoid unpredictable effects during simulation (which could be the focus of another study), the code of each characterization program is executed five times, in a loop, and fault injection is only performed during the last iteration. Experimentally, we can see that during this last iteration, branch prediction is correct and relevant data is already loaded in cache memory.

3.1.2 Use of RTL fault injection in our approach

The overview of RTL experiments is shown in Figure 3. For each characterization program, we perform a golden simulation (no fault injected), and faulty simulations. The results of faulty simulations are compared to the golden simulations in order to deduce the precise effects of the fault injection. These effects are then written in a database that is called RTL database for simplicity (RTL DB in Figure 3).

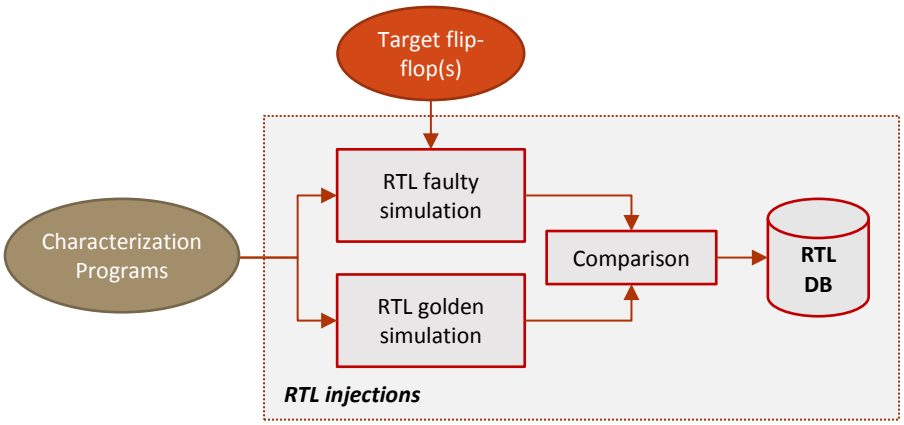

Figure 3: Overview of RTL experiments. Characterization programs are used to perform faulty and golden simulations. The results are then compared and stored in a database.

For every fault injection, the results in the database can be classified in four categories:

- Silent: no visible effect (in the register-file and memory) was detected between the golden and faulty simulation, at both observation points.
- Exception: an exception was raised by the processor during simulation.
- Unknown: one of the observation points was not found.
- Analyzable fault: all faults not in the previous categories.

These categories are similar to those frequently encountered in related papers like [7] or [19]. Only analyzable faults are of interest to us; they are the ones that are compared to software fault injections.

3.2 Software fault injection

3.2.1 A mutation tool to perform software fault injection

Software fault injection can be performed in different ways. One of the most important characteristics we need is flexibility, to be able to inject faults from various software models. For that, we developed a tool to create a mutation of the original program, embedding a software fault model. This mutation tool has already been presented in [17], but we provide here a brief overview of how it works.

The tool takes two inputs: the binary file of the compiled program to attack and a software fault model described in an XML configuration file. To be able to describe a wide variety of effects, we need to represent low-level aspects of the processor; that is why we chose to represent the behavior of the binary file at a higher representation level (in C language). The important point is that the behavior of the binary file is preserved at this level of representation: there is no modification in how the program executes (this is similar to a disassembly, but different from a decompilation). The tool works as follows. First, each instruction in the binary is represented at the C level in three phases that correspond to the Decode, Execute and Write-Back stages of a typical processor pipeline. Then, different sections from the XML file are inserted between these stages to represent a faulty behavior. An example is shown in **Figure 4**, for a fault model that consists in replacing the first argument of the instruction by the result of the previous instruction (fault on the forwarding structure of the processor). Once the mutant is generated, we get a C file that represents the behavior of the binary file under a particular software fault model. It can then be compiled and executed to perform the injection. One limitation of this mutation tool is that it has a limited support of indirect jumps (jumps whose target is not known at compilation time).

Since it is important to have the same injection and observation rules for both the RTL and software fault injections, the mutant is made to comply with the injection and observation rules in use in RTL simulations. In particular, the observation covers the contents of the register-file and the writing transactions to memory.

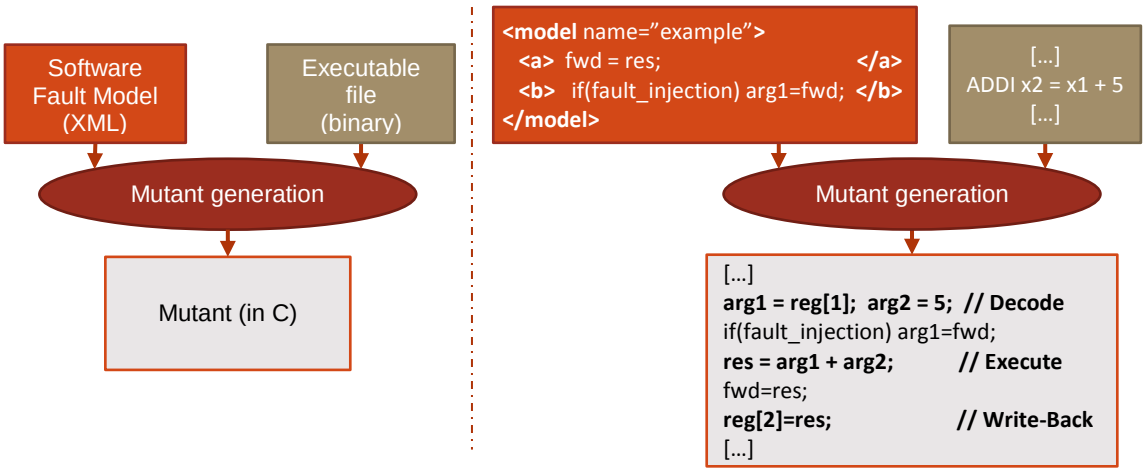

Figure 4: Overview (left) and example use (right) of the mutation tool. The mutated instruction adds 5 to the value in register 1, and stores the result in register 2. The fault model replaces the first argument of the sum by the result of the previous instruction.

3.2.2 Use of software fault injection in our approach

We use software fault injection in a similar way as RTL fault injection, except for the additional preliminary step of mutant generation. The overview is shown in .

The mutants generated by our tool are compiled and executed to obtain results for a golden and faulty runs. These results are then compared to obtain the precise effects of the model, and these effects are written in a second database that is called software database. Note that results from the golden run could also be obtained outside of the mutant, but it is simpler to do it this way because the process is the same as the faulty execution, except fault injection is not activated. Just like in RTL fault injection, results of software fault injection are classified in the same 4 categories: silent, exception, unknown and analyzable.

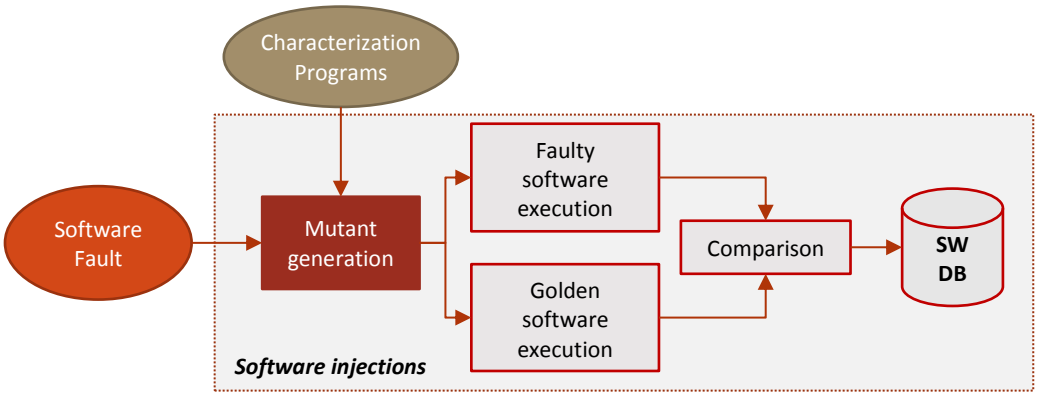

Figure 5: Overview of software experiments. Characterization programs are used to perform faulty and golden simulations. The results are then compared and stored in a database.

4 Cross-layer analyses and metrics

The last step of the approach is to compare results obtained with RTL fault injection with those obtained with software fault injection. In this section, we show various analysis methods, such as computing metrics to evaluate software fault models, selecting the best models, helping to define new models, and realizing attacks from the software models. We start this section by introducing a mathematical formalism to help understand precisely the role of the various analyses. Then we go on to explain these analyses.

4.1 Mathematical formalism

In order to clarify this work, we introduce here a mathematical formalism which is used to describe the processes of fault injection and analysis previously defined. This formalism is inspired from [14], but is adapted to our approach since our means of injection are different.

First, we define $\boldsymbol{C}$, the set of all characterization programs. In these are defined in particular the target instruction as well as the observation instruction (for the propagated effect ; the instantaneous effect being observed at the target instruction, as defined in previous sections).

We also define $\boldsymbol{M}$, the set of all injection methods. In the case of our RTL fault injections, the injection method is simply the name(s) of the faulted flip-flop(s). In the case of software fault injections, the injection method is name of the model that is used.

$\boldsymbol{E}$ is the set of all visible effects of an injection at a certain point in time. We define an effect as the set of differences between a golden execution and a faulty execution of the characterization program, at a certain point in time. For instance, if in a given context, a fault injection results in the same state as a golden execution, except for register 10 that has value 1 instead of 0, and register 12 that has value 42 instead of -42, then the effect $e \in \boldsymbol{E}$ is written "(R10:0:1)(R12:-42:42)". If there are also effects on memory, we add the concatenation of all writing transactions.

We define $\boldsymbol{B} \equiv \boldsymbol{C} \times \boldsymbol{E} \times \boldsymbol{E}$ the set of all injection behaviors. A behavior is defined as a characterization program, an instantaneous effect and a propagated effect, regardless of the means used to obtain these observations (it can be through RTL or software fault injection). A behavior simply describes the effects that can be obtained by attacking in a particular context.

An *injection result* is an element of the set $\boldsymbol{R} \equiv \boldsymbol{B} \times \boldsymbol{M}$. An *injection result* contains the same information as a behavior, except that it also specifies the injection method used to obtain the behavior. Thus, an element in $\boldsymbol{R}$ completely describes the parameters and results of an injection experiment. The distinction with behaviors is made so that injection results obtained from different abstractions (RTL simulation and software injection) can be compared (more on that in the next section).

To clarify, Table 1 summarizes all these set definitions.

Table 1: Summary of the set definitions

Set	Name	Example	Meaning
C	Characterization programs	$c =$ "prog1".	The injection happens in this program, where a target instruction and an observation instruction are defined.
M	Injection methods	$m_1 =$ "$ff1\ ff2$" $m_2 =$ "$model1$"	RTL injection in flip-flop *ff1* and *ff2*. Software injection with *model1*.
E	Injection effects	$e =$ "(R10:0:1)(R12:-42:42)"	At a certain point in the execution, register 10 should have the value 0, but it got value 1; and register 12 got value 42 instead of -42.
B	Injection behaviors	$b = (c, e_1, e_2)$	Injecting a fault in program c led to instantaneous effect e_1, and propagation effect e_2.
R	Injection results	$r = (b, m)$	The behavior b can be obtained with injection method m.

A *fault injection* is a mapping $\boldsymbol{C} \times \boldsymbol{M} \rightarrow \boldsymbol{R}$ which maps an injection context and an injection method to an injection result. In our case, the fault injections are the RTL and software injections that were described in sections 3.1 and 3.2. We call them f_{RTL} and f_{SW} respectively.

In order to be able to compare results of RTL fault injections with software fault injections, we used the same set of characterization programs C_0 for both sides. We however distinguish M_{RTL} and M_{SW}, the injection methods for RTL (i.e. all target flip-flops) and software injections (i.e. all software fault models) respectively. Injecting in the contexts C_0 with the flip-flops defined in M_{RTL} results in the R_{RTL} set. Similarly, injecting in the contexts C_0 with the models defined in M_{SW} results in the R_{SW} set. The injection campaigns are thus represented in Figure 6.

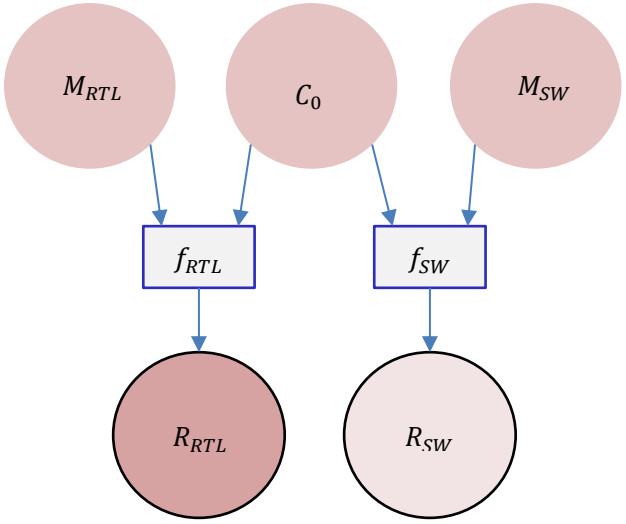

Figure 6: Diagram of the injection campaigns. The process of RTL injection f_{RTL}takes as input flip-flops defined in M_{RTL} and characterization programs in C_0, and outputs injection results in R_{RTL}. Similarly, the process of software injection f_{SW} takes input models in M_{SW} and programs in the same set C_0, and outputs injection results in R_{SW}.

4.2 Comparing injection results

The comparison between RTL and software fault injections only makes sense for analyzable faults (on both sides), as discussed in sections 3.1.2 and 1.1.1. We thus define R_{RTL}^{A} the subset of analyzable faults of R_{RTL}. We define similarly R_{SW}^{A} for R_{SW}.

Since results produced in RTL fault injections and in software fault injections have a different *method* of injection (flip-flops in the former case; software fault models in the latter), these results are necessarily non-overlapping. To compare results obtained in both injections, we therefore define the following projection mapping:

$$\boldsymbol{\pi} : R \rightarrow B$$
$$\boldsymbol{\pi}(b, m) = b$$

This projection takes an injection result and maps it to its corresponding behavior. Multiple injection results can map to the same behavior; in this case, the number of injection results that maps to the same behavior can be seen as the *attack area* of this particular behavior. Since this projection gets rid of the injection method, we can now study the overlaps between results of RTL and software injections. This overlap is shown in Figure 7, between the $\boldsymbol{\pi}(R_{RTL}^{A})$ and $\boldsymbol{\pi}(R_{SW}^{A})$ circles. With this projection mapping, an RTL injection result and a software injection result match if and only if the injection happens at the same point in the execution, and both the instantaneous effect and the propagated effect are identical. This is a rather strict comparison; the comparison with a less strict rule is discussed in a later section.

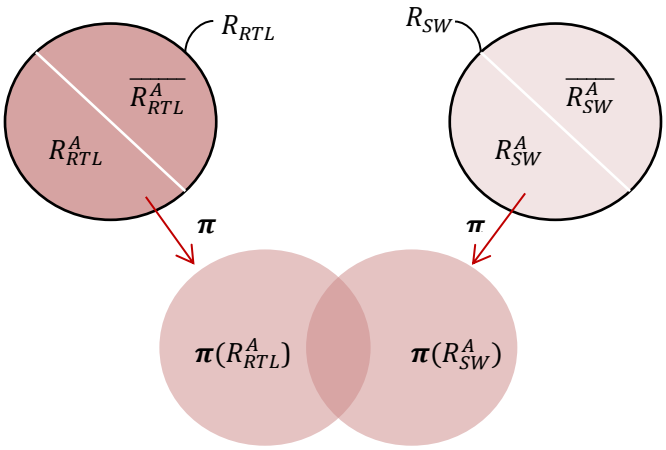

Figure 7: Representation of the various fault spaces. The top left circle represents the RTL injection results and the top right circle the software injection results. Both can be divided in analyzable and non-analyzable results The analyzable parts can be projected into fault behaviors space to study the overlap between RTL injection and software fault models.

We also define the following indicator function, defined on the set of behaviors, with *D* a behavior set:

$$\mathbf{1}_D : B \rightarrow \{0,1\}$$
$$\mathbf{1}_D(b) = \begin{cases} 1 & if\ b \in D \\ 0 & otherwise \end{cases}$$

4.3 Coverage metric

With previous definitions, we can say that the software fault models **cover** an RTL injection result $r \in R^A_{RTL}$ if and only if:

$$\mathbf{1}_{\boldsymbol{\pi}(R^A_{SW})}(\boldsymbol{\pi}(r)) = 1$$

In other words, the software fault models cover an RTL injection result if and only if its projection falls in the overlapping part in Figure 7. The injections have to be in the same characterization program, and they have to produce the same instantaneous and propagated effects. By summing over all elements of R^A_{RTL}, and dividing by the number of analyzable faults, we can define our first metric, the *fault coverage (FC)* of the fault models:

$$FC = \frac{\sum_{r \in R^A_{RTL}} \mathbf{1}_{\boldsymbol{\pi}(R^A_{SW})}(\boldsymbol{\pi}(r))}{|R^A_{RTL}|}$$

The *fault coverage* gives the percentage of analyzable RTL faults that are predicted by software fault models. It embeds a sense of attack area: different injections that lead to the same behavior are counted multiple times. But it can also be interesting to count them only once, and hence focus only on the behaviors. We thus define the *behavior coverage (BC)*:

$$BC = \frac{\sum_{b \in \boldsymbol{\pi}(R^A_{RTL})} \mathbf{1}_{\boldsymbol{\pi}(R^A_{SW})}(b)}{|\boldsymbol{\pi}(R^A_{RTL})|}$$

4.4 Fidelity metric

When performing software fault injection, the goal is to directly produce at the software level the effects that can happen at the hardware level. But due to approximations, a fault model can also predict cases which cannot happen in reality. To give an idea of the fidelity of software fault models, we do the

opposite of a coverage analysis. We want to know whether each entry of the software database is also present in the RTL database. We can then construct a diagram such as Figure 8. In this figure, software fault injection results are grouped in 5 categories:

- Correct faults: software faults that correctly predict an effect obtained in RTL injection.
- Redundant but correct faults: redundant cases of the previous category (since several software fault models can produce exactly the same results).
- Non-analyzable faults: cases in which the software fault injection does not conclude (the model either does not create any visible effect, or the mutant execution encountered a problem).
- Incorrect faults: software faults that predict effects that have not been observed in RTL injection.
- Redundant incorrect faults: redundant cases of the previous category (since several models can produce the exact same result).

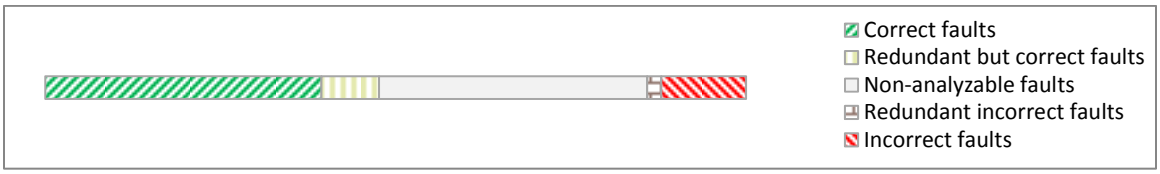

Figure 8: Fidelity diagram of a set of software fault models.

Non-analyzable faults do not bring additional information, nor do redundant faults (getting the same behavior with multiple models does not bring any new information, contrary to RTL injection where this is linked to the concept of attack area). Ignoring these cases and only considering unique correct and incorrect faults, we can construct a metric that gives the proportion of correct faults. We call this the *model fidelity* metric, and it can be expressed as:

$$MF = \frac{\sum_{b \in \boldsymbol{\pi}(R_{SW}^{A})} \mathbf{1}_{\boldsymbol{\pi}(R_{RTL}^{A})}(b)}{|\boldsymbol{\pi}(R_{SW}^{A})|}$$

Note this formula has the same structure as the *behavior coverage*; only the role of RTL and software results is switched. The model fidelity can be computed either on a single model or on a set of models. A set of models having a fidelity of 75% means that in 3 out of 4 cases, the model gives a result that corresponds to an actual fault effect, while in 1 out of 4 cases, the model gives a result that has not been obtained in RTL simulations.

4.5 Search of uncovered faults

After having evaluated the fault coverage of a set of software fault models, a designer can try to improve its coverage by adding new software fault models. In that case it is useful to highlight the uncovered injection results that are the most common. This gives an indication as to where the designer should focus her/his study. This can be done for example by ordering the behaviors of the RTL database from the ones that are the most often seen (behaviors which can be obtained with many different RTL injection methods) to the ones that are the least often seen; and filtering out the ones that are also present in the software database. The resulting behaviors are those that are not predicted by software fault models but that can be provoked by various RTL injection methods.

More formally, the goal here is to find $b \in \boldsymbol{\pi}(R_{RTL}^{A})$ such that $b \notin \boldsymbol{\pi}(R_{SW}^{A})$ and b has the highest number of preimages in R_{RTL}^{A} under $\boldsymbol{\pi}$.

4.6 From software models to RTL simulation: drawing a model profile

So far, the methodology has been used to abstract the fault effects from RTL simulation into software fault models. But it is also possible to do the reverse operation: materialize faults predicted by software

fault models into actual faults in a RTL simulation. This can be useful to guide a hardware fault injection campaign: instead of faulting flip-flops randomly, the most sensible flip-flops can be targeted in priority. In particular, such method is interesting to complement software fault models with a low score on the fidelity metric: if a software model is not very accurate, quick hardware experiments can strengthen a designer's confidence in its results.

This can be done as follows. For each element $r \in R_{SW}^{A}$ obtained with a given software fault model, we can look in the RTL database which flip-flops create the same behavior. Then, for each flip-flop, we can sum the number of cases where software and RTL injections match. This results in a *model profile* that highlights the flip-flops that most likely produce the effects of a given software fault model. Two profiles are shown and discussed in section 5.2.8.

5 Results

In this section, we discuss results obtained by conducting several fault injection campaigns. Note that since our approach aims at studying a specific processor implementation, the results may vary greatly by using a different one. We present here the results obtained on a RISC-V processor called LowRisc, with several characterization programs and several software fault models. The same approach can be used to study other processors, programs and fault models. We first introduce the inputs that were used (characterization programs and injection methods), and then analyze results obtained with these inputs.

5.1 Inputs

As shown in Figure 6, in order to use the approach, three types of inputs need to be defined: the input programs (the C_0 set), the target flip-flops (the M_{RTL} set) and the software models (the $M_{SW}\ set$).

5.1.1 Input characterization programs

5.1.1.1 Programs for modeling: assembly contexts

The first characterization programs we used were small assembly contexts we designed to use different types of instructions of the RISC-V base integer instruction set (RV64I), in different situations (most notably forwarding situations). All prologues, targets and epilogues are shown in . Combining these prologues, targets and epilogues creates 105 different contexts. Deciding whether there are enough assembly contexts is up to the designer; note that a good understanding of the microarchitecture can help specifying relevant contexts that make use of various parts of the processor.

Table 2: Lists of prologues, targets and epilogues

Prologue name	Instructions
Nofwd	ADDI a0, x0, 5; ADDI a1, x0, -15; NOP; NOP;
Fwdone	ADDI a0, x0, 5; ADDI a1, x0, -15; NOP;
Fullfwd	ADDI a0, x0, 5; ADDI a1, x0, -15;
Target name	**Instruction**
Add	ADD a2, a0, a1;
Add_r	ADD a2, a1, a0;
Addi	ADDI a2, a1, 37;
Lw	LW a2, 52(tp);
Sw	SW a1, 16(tp);
Bltz	BLTZ a1, label;
Jal	JAL label;

Epilogue name	Instructions
Nop3	NOP;NOP;NOP;
Fwdfst	ADDI a3, a2, 78; NOP;
Fwdsnd	NOP; ADDI a4, a2, 78; NOP;
Fwdboth	ADDI a3, a2, 78; ADDI a4, a2, 78; NOP;
Sb	SB a2, 16(tp); NOP;

5.1.1.2 Programs for validation

In order to verify that the models designed with the assembly contexts were useful in real-world situations, we also used two actual applications: VerifyPIN and LittleXorKey. Note that although we used these two applications for validation purposes, they could also be used in the modeling phase.

The VerifyPIN application

VerifyPIN is a secured application that comes from the FISSC library [9]. It is a 4-digit PIN verification which compares a PIN entered by the user with a secret PIN. It comes in different versions with various software countermeasures. We consider here version 6, protected in particular with hardened Booleans and duplicated Boolean tests. The user PIN was set to 2709 and secret PIN to 2019. The program was compiled with GCC, with option –Og to avoid deleting countermeasures. This application is rather control-flow-centric since the program consists in comparisons of several values. During an execution of this program, 103 instructions are executed. All instructions of this program were targeted during injection campaigns, and the observation of the propagated effects was performed at the end of the program.

The LittleXorKey application

LittleXorKey is a simple program that XORs two 2x2 matrices. This is a simplified version of the XORing function used in the AES. The code of the program is presented in Listing 3. It was also compiled with GCC –Og. During an execution, 71 instructions are executed. Like for VerifyPIN, all instructions were targeted, and the observation point was set at the end of the program.

```
int loopCount=0;
for(i=0 ; i < 2 ; i++){
       for(j=0 ; j < 2 ; j++){
              usr[i][j] ^= key[i][j];
              loopCount++;
       }
}
```

Listing 3: Code of the LittleXorKey application, which XORs the *usr* and *key* matrices. A counter is also incremented at each iteration, to "complexify" the program and generate more fault behaviors.

5.1.2 RTL injection method: the M_{RTL} set

RTL fault injection was performed on a processor with a RISC-V architecture. The chosen implementation was LowRISC version 0.4, which has a 5-stage pipeline: Instruction Fetch, Instruction Decode, Execute (EX), Memory (MEM) and Write-Back (WB).

Like in [7], fault injection was focused on pipeline flip-flops. No flip-flop belonging to the register-file nor the memories were targeted since these faults have already been studied [3][8], and countermeasures exist to thwart these types of faults. We also did not inject faults in the Decode and Fetch stages since these faults would mostly result in wrong instructions being fetched, or a corruption of the instruction word; while the last three stages (EX, MEM and WB) are more interesting since this is where the actual computations are performed in the processor. As previously mentioned, each flip-flop in the processor

was associated to a specific pipeline stage: EX, MEM or WB. Some flip-flops were associated to multiple stages. In our processor core, there were 1116 flip-flops, and we used 1308 stage/flip-flop pairs.

5.1.3 Software injection method: the M_{SW} set

Finally, the last inputs to specify are the software fault models. In Table 3, we show the set of models that was used for this paper. This set is a mix of typical models (like **skip, res0** or **test_inv**) and complex models designed by studying the microarchitecture of the processor (like **skip_mem** or models based on forwarding: **fwd**) . Since the modeling phase of our approach is iterative, this set is not set in stone: models can be added, modified or deleted to better represent the behavior of the processor. Note that each software fault model is applied exhaustively to all characterization programs.

Table 3: Models of the M_{SW} set

Model	Description
Arg2_4	Replace the second argument of the ALU by 4.
Arg2_rs	Replace the second argument of the ALU by a value that depends on a previously computed result.
Flipbit*x*	Flip one bit of the result at position *x* (there are 32 models in total)
Fwd_0	Replace first argument to the ALU by 0
Fwd_1	Replace first argument to the ALU by the last computed value
Fwd_2	Replace first argument to the ALU by the second-to-last computed value
Fwd_3	Replace first argument to the ALU by the last value loaded from memory
Fwd_4	Replace second argument to the ALU by 0
Fwd_5	Replace second argument to the ALU by the last computed value
Fwd_6	Replace second argument to the ALU by the second-to-last computed value
Fwd_7	Replace second argument to the ALU by the last value loaded from memory
Lsb0	Replace first argument to the ALU by a value that depends on a previous computed value
Res0	Result of the instruction is 0
Skip	Skip the instruction
Skip_mem	Reverse the results of the targeted instruction after 1 instruction
Skip_wb	Reverse the results of the targeted instruction after 2 instructions
Test_inv	Inverse the condition of a test
Wrong_load	Replace the value loaded from memory by 0

5.2 Detailed analysis

In this section, results of various analyses are reported, for the inputs presented in previous sections.

5.2.1 Single-bit injection campaigns

First, we discuss the results of single-bit injection campaigns. Since the number of flip-flops in the pipeline is rather small, we could perform exhaustive single-bit campaigns. By that we mean that the target instruction in each assembly context was faulted using all flip-flop/stage pairs. Since there are 1308 flip-flop/stage pairs, the 105 assembly contexts led to a total of 137340 RTL injections. For VerifyPIN, faulting all flip-flop/stage pairs in the 103 instructions led to 134724 RTL injections, and finally, for LittleXorKey and its 71 instructions, 92868 RTL injections were carried out. For information, the single-bit exhaustive campaign of the assembly contexts took 68 hours, on a server running 4 concurrent QuestaSim

simulations (12 cores with 32GB of RAM). The outcomes of these campaigns are shown in Table 4, as well as the *fault coverage FC* of the set of software fault models defined in the previous section.
The outcomes of the various programs were rather similar, except for the number of unknown cases in VerifyPIN, which was much higher than the others. This fact means that at least one of the observation points was not found during simulation, which is not really surprising because an execution of VerifyPIN has a lot of jumps/branches, and a fault modifying the jump address can end up in a distant address. In any case, the number of analyzable faults was around 15% (as a reminder, silent faults are not considered analyzable since they do not need to be modeled). The fault coverage was also very similar for the various cases. This shows in particular that the assembly contexts are useful for describing the behaviors that happen in real applications. The discussion on the coverage values is postponed to section 5.2.5.

Table 4: Results of exhaustive RTL single-bit injection campaigns

	Silent	Exception	Unknown	Analyzable	FC
Assembly contexts	80.6%	3.3%	0.6%	15.4%	24.0%
VerifyPIN	65.5%	2.3%	22.4%	9.8%	28.7%
LittleXorKey	76.6%	2.7%	2.8%	17.9%	29.0%

5.2.2 Multiple-bit injection campaigns

For multiple-bit injections, due to combinatorial explosion, not every combination of flip-flops could be injected. We instead used statistical fault injection as explained in [13]. The context to fault was chosen randomly. The sets of flip-flops were chosen randomly as long as they were associated to the same pipeline stage. This constraint ensured that the injections happened at the same instant in the execution.
Several campaigns were performed for each characterization program. Each campaign consisted in injecting between 35000 and 100000 faults. Results are shown in Table 5, Table 6 and Table 7, with the 95% confidence interval defined in [13]. We can see multiple-bit injections create less silent cases, which was expected: the more bits faulted the more analyzable cases are generated. For the same reason, the fault coverage also decreases , but the results are still is the same vein as those of single-bit attacks.

Table 5: Outcomes of multiple-bit injections for assembly contexts

	Silent	Exception	Unknown	Analyzable	FC
1-bit	80.6%	3.3%	0.6%	15.4%	24.04%
2-bit	66.2 ± 0.3%	6.4 ± 0.2%	1.2 ± 0.1%	26.1 ± 0.3%	~ 22.51%
3-bit	54.8 ± 0.4%	9.4 ± 0.2%	1.7 ± 0.1%	34.0 ± 0.3%	~ 20.21%
4-bit	46.6 ± 0.5%	12.0 ± 0.3%	2.3 ± 0.2%	39.2 ± 0.5%	~ 18.41%
5-bit	40.0 ± 0.5%	14.4 ± 0.4%	2.5 ± 0.2%	43.1 ± 0.5%	~ 17.68%

Table 6: Outcomes of multiple-bit injections for VerifyPIN.

	Silent	Exception	Unknown	Analyzable	FC
1-bit	65.5%	2.3%	22.4%	9.8%	28.74%
2-bit	54.4 ± 0.4%	4.5 ± 0.2%	24.4 ± 0.3%	16.9 ± 0.3%	~ 27.09%
3-bit	45.4 ± 0.4%	6.6 ± 0.2%	25.7 ± 0.3%	21.9 ± 0.3%	~ 25.63%
4-bit	39.3 ± 0.5%	8.4 ± 0.3%	27.1 ± 0.4%	25.2 ± 0.4%	~ 24.18%
5-bit	34.4 ± 0.5%	9.9 ± 0.3%	28.3 ± 0.5%	27.4 ± 0.5%	~ 22.21%

Table 7: Outcomes of multiple-bit injections for LittleXorKey.

	Silent	Exception	Unknown	Analyzable	FC
1-bit	76.6%	2.7%	2.8%	17.9%	29.00%
2-bit	59.7 ± 0.5%	5.0 ± 0.3%	5.1 ± 0.3%	30.2 ± 0.5%	~ 25.73%
3-bit	47.2 ± 0.6%	7.2 ± 0.3%	7.6 ± 0.3%	38.0 ± 0.5%	~ 22.56%
4-bit	37.4 ± 0.5%	9.8 ± 0.4%	9.4 ± 0.3%	43.5 ± 0.6%	~ 20.34%
5-bit	30.1 ± 0.5%	12.3 ± 0.4%	11.6 ± 0.4%	46.0 ± 0.6%	~ 18.28%

5.2.3 Instruction skip

One of the most famous software fault models is the instruction skip. Running our analysis shows that it indeed has one of the highest coverages. However, as shown in Table 3, there are two other slightly more advanced skip models, namely **skip_mem** and **skip_wb**. These two models take into account a processor optimization called forwarding. They behave like a typical instruction skip when there is no forwarding, but behave differently when there is forwarding involved. In other words, their behavior depends on the epilogue. Running the coverage analysis on the various campaigns shows that these two models are consistently better than the typical instruction skip. The fault coverages of these three models are presented in Table 8. Note however that even though there are many overlaps between the three models, each predicts some cases not predicted by others.

Table 8: Fault coverage of the skip, skip_mem and skip_wb models (in %), for single and multiple-bit RTL injection campaigns.

Skip/skip_mem/skip_wb	1-bit	2-bit	3-bit	4-bit	5-bit
Assembly contexts	2.8/3.3/3.3	2.9/3.6/3.5	3.1/3.6/3.5	3.0/3.5/3.4	3.2/4.2/4.0
VerifyPIN	8.0/8.6/8.4	8.2/8.7/8.5	8.1/8.8/8.5	8.5/9.1/8.8	8.3/8.8/8.5
LittleXorKey	3.2/3.5/4.2	3.0/3.4/4.5	3.0/3.3/4.1	2.9/3.2/4.0	2.7/3.0/3.9

5.2.4 Behavior coverage

So far, the discussion has been focused on *fault coverage*. An interesting thing is also to compare it to the *behavior coverage*. Table 9 shows the values obtained for the various campaigns.

In general, our models have a fault coverage twice as high as the behavior coverage. For single-bits attacks against VerifyPIN, we cover 28.7% of the faults, but that only accounts for 14.4% of the possible behaviors. Obtaining a fault coverage higher than the behavior coverage makes sense: the faults we want to cover in priority are those that are seen more often (those that have a big attack area).

Table 9: Comparison of fault and behavior coverage for various campaigns (in%).

FC/BC	1-bit	2-bit	3-bit	4-bit	5-bit
Assembly contexts	24.0 / 13.6	22.5 / 12.6	20.2 / 9.6	18.4 / 10.2	17.7 / 9.6
VerifyPIN	28.7 / 14.4	27.1 / 13.4	25.6 / 10.1	24.2 / 10.8	22.2 / 10.5
LittleXorKey	29.0 / 16.5	25.7 / 17.8	22.6 / 13.7	20.3 / 11.2	18.3 / 9.9

5.2.5 Discussion on the coverage

The obtained coverages seem rather low despite the relatively high number of software fault models. This gives evidence of the diffculty to model faults on a processor pipeline at the software level. These values are also the consequence of these several points:

- The RTL injection strategy sometimes causes strange results. As mentioned previously, it is difficult to automate simulations when faults can cause unpredictable effects.
- We inject faults in the processor core, where all the logic happens (and not in the register-file for example). Impacting a single flip-flop here can have a wide range of effects. Furthermore, the processor has no hardware countermeasure.
- The projection mapping defined in section 4.2 is very strict. An RTL injection maps to the same behavior as a software injection only if both instantaneous and propagated effects are identical.

The first reason is a technical point that coud be improved (although this concerns only a few cases, so the impact on the coverage is low). The other two reasons are choices that we made for this study, but could be changed. Since our goal is to study very precisely the various behaviors that can be provoked in a processor, we adopted a cautions point of view: the second reason ensures that the fault space to cover is big, while the third reason imposes a high accuracy on the software fault models to match these RTL results. This makes fault covering a difficult, but more complete task.

To illustrate this, we can examine a less strict projection mapping. For example, we could define the following one, which does not consider the instantaneous effects:

$$\boldsymbol{\pi}'\big((c, e_1, e_2), m\big) = (c, \emptyset, e_2)$$

Using this new projection, the fault coverage of exhaustive single-bit RTL campaigns goes from to 28.7% to 44.2% for VerifyPIN and from 29.0% to 49.4% for LittleXorKey. The reason behind this huge increase in coverage is that the observation point is set at the very end of the application. By getting rid of the instantaneous effects, we can see that many faults get masked before the end of the execution, and thus have no final impact on the program. The change in projection mapping however does not affect much the fault coverage for assembly contexts, which goes from 24.0% to 24.7%, because both observation points are close to each other anyway. The choice of the projection mapping is a matter of accuracy: the more comparison points, the more accurate software models need to be to cover faults. To stay coherent with previous analyses, we continue the paper with the original projection mapping, $\boldsymbol{\pi}$.

Before closing this discussion on the coverage, a last question should be examined: by how much the coverage could be increased by adding other software fault models. Currently, if we restrict ourselves to the single-bit injection campaigns on assembly contexts, the most common uncovered behavior, found by applying the method shown in section 4.5, has an attack area of 5 (meaning that 5 flip-flops can provoke this behavior). If each of the 105 contexts had a behavior with this same attack area, then a perfect additional software fault model could at most cover 525 new faults, which means increasing the fault coverage by only +2.5% (going from 24.0% to 26.5%). Of course, in reality such a new model would be very difficult to achieve. Currently, the only models that individually cover more than 400 faults are the skip models. Moreover, there is a diminishing return on the fault models: the higher fault coverage, the more difficult it gets to specify models that cover many new faults.

So it is possible to increase the coverage, but each new model can only improve it by small increments. A lower bound of the number of software fault models required can also be extracted from the databases. The context that creates the highest number of analyzable behaviors (this context uses prologue “fullfwd” with target “addi” and epilogue “fwdboth”), generates 292 different analyzable behaviors. Consequently, to cover every possible single-bit scenario, at least 292 software fault models would be required. That lower bound can only be reached if there is absolutely no overlap between the models, and if each covers a different behavior (and these models also need to cover all behaviors for the other assembly contexts). Currently, with our software fault models of, 37 of these 292 behaviors are covered.

5.2.6 Finding where to add hardware countermeasures

To achieve a better coverage with fewer models, hardware countermeasures could be added to protect the pipeline of the processor. Figure 9 shows the twenty flip-flops that create the most analyzable faults for the exhaustive single-bit injection campaign on assembly contexts. On this figure, we can see that there are no flip-flop that clearly detach itself from the rest. This again shows the difficulty to protect a processor effectively: the origin of software behaviors is very varied. On the figure are also highlighted the cases that are predicted by software fault models. This is helpful to decide which structures to protect in hardware: the effects provoked by some flip-flops can be effectively studied at the software level and hence also protected at this level; while others do not have this possibility.

This is an interesting aspect of the approach: its role is cross-layer; it can give information both to the software and hardware sides, to enhance security in a global way. Here, knowing that some flip-flops are well taken into consideration at the software level can influence decisions in the hardware layer.

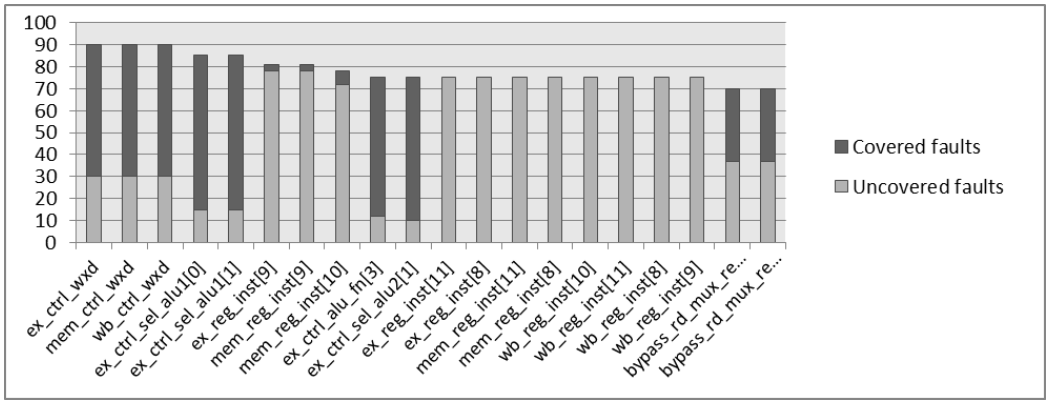

Figure 9: Number of analyzable faults produced by some flip-flops for the exhaustive single-bit campaign on assembly contexts. Since there are 105 different contexts, a flip-flop only creates at most 105 analyzable faults.

5.2.7 Fidelity analysis

Overall, by applying every fault model of M_{SW} to every injection context in C_0 (assembly contexts, VerifyPIN and LittleXorKey), we got the fidelity diagram of Figure 10. This corresponds to a 76.5% fidelity, which means that when performing software fault injection with the models of M_{SW}, 3 out of 4 behaviors correspond to actual results that can be obtained through RTL simulation. In this diagram, the non-analyzable faults represent a huge percentage of all faults. For the most part, these cases are silent cases: the model does not produce a result different from the golden run.

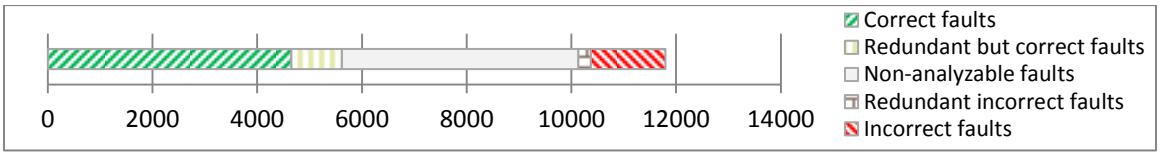

Figure 10: Global fidelity diagram. The axis shows the number of injection results.

The fidelity of various software fault models varies a lot. The **res0** and **fwd_0** models have a fidelity of 100% and 96.8% respectively, which is almost perfect. This justifies the common use of the value zero in typical software fault models: it is a value that is seen very frequently when injecting actual faults. If a vulnerability in a program is discovered with one of these models, the vulnerability should be taken very seriously, since it is highly probable that it can be reproduced in reality (or at least in RTL simulations). Models **fwd_1**, **skip** and **skip_mem** have a fidelity of 82.1%, 81.2% and 81.1% respectively, which is still a good score. Finally, models **arg2_4** and **arg2_rs** have the worst fidelity, at 60.7% and 41.4% respectively.

5.2.8 Model profiles

In section 4.6, we introduced the notion of model profile as a way to go from a software fault model to an actual RTL fault injection. In Figure 11, two profiles are shown, for models **arg2_4** and **skip**. These profiles are computed considering all single-bit injection campaigns.

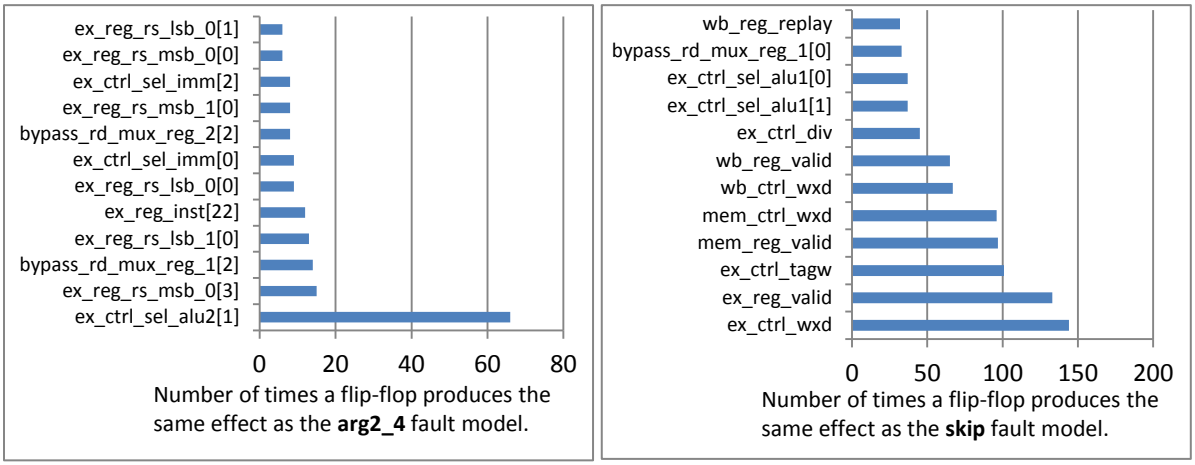

Figure 11: Profiles of the arg2_4 (left) and skip (right) models.

The profile shape depends on the characterization programs and the accuracy of the model. A very accurate model has a peak with a low dispersion. This is the case of model **arg2_4**, which models a very specific effect in the processor microarchitecture (modifying the selection signal of a multiplexor). Its profile highlights this specific structure. On the other hand, a more abstract model, like the instruction skip, has a more even profile because it represents a more general behavior not tied to a particular structure or flip-flop in the processor. In Figure 11 we can see there are several ways to produce the skipping effect. The most likely flip-flops are validation signals (valid) and write-enable signals (wxd). Note that the numbers shown in Figure 11 should not be compared directly: they are related to the coverage (the **skip** model has a higher coverage than **arg2_4**).

5.2.9 Attacking VerifyPIN

Since VerifyPIN is a protected application, it is interesting to see if the countermeasures succeed in thwarting all RTL attacks, and if not, if the software fault models predict these attacks. In VerifyPIN, an attack is considered successful if the attacker manages to authenticate with a wrong PIN code and without triggering the countermeasures. By analyzing the results of the exhaustive single-bit RTL campaign, we can see that over the 134724 injections, 54 are successful attacks. Among these 54 attacks, 15 are predicted by software fault models (which is in line with the fault coverage of around 25% seen previously); the others are mostly due to very specific jumps, which are difficult to model in our mutation tool. Once an attack is predicted by a software fault model, one might want to verify that this attack is feasible, particularly for low-fidelity models such as **arg2_4** and its 60.7% fidelity. That is where model profiles come into play. By highlighting the most likely flip-flops to inject, a quick RTL simulation can be performed to verify whether the predicted attack is a false positive or not.

In Table 10, the flip-flops responsible for the attacks are shown, as well as the rank in which they appear in the *model profiles* of the software fault model that predict the attack. The table should be read as follows: there is a successful attack when faulting flip-flop “ex_ctrl_sel_alu2[1]” (second line of the table); this attack is predicted only by model **arg2_4** and if we look at its profile (in Figure 11), the faulty flip-flop appears at the first place.

Here, many attacks are predicted by two models. Some flip-flops are seen twice in the table because they can be the recipient of the fault at multiple instants. In the table, we can see that the flip-flops responsible of the attacks are amongst the most likely candidates in the profiles of the models that predict these attacks, which confirms the usefulness of model profiles.

Table 10: Ranks of flip-flops in the profiles of models that lead to successful attacks in VerifyPIN.

Attack	arg2_4	flipbit2	arg2_rs	flipbit1	Fwd_2
ex_ctrl_sel_alu2[0]	-	-	1	12	-
ex_ctrl_sel_alu2[1] (1st)	1	-	-	-	-
ex_ctrl_sel_alu2[1] (2nd)	1	7	-	-	-
bypass_rd_mux_reg_1[1]	-	-	9	1	-
bypass_rd_mux_reg_1[2]	3	1	-	-	-
bypass_rd_mux_reg_2[1]	-	-	9	2	-
bypass_rd_mux_reg_2[2]	8	2	-	-	-
ex_reg_inst[21]	-	-	2	4	-
ex_reg_inst[22]	5	3	-	-	-
ex_reg_rs_lsb_0[1] (1st)	-	-	9	5	-
ex_reg_rs_lsb_0[1] (2nd)	-	-	-	-	4

In the table, only 11 attacks are shown. The remaining four are predicted by two models: **skip** and **skip_wb**. As we have discussed in previous section, these models are very general, so their profile is not very pronounced. In this case, that means the flip-flops responsible for the attacks do not stand out.
As a side note, if we look from the software perspective, there are 9 predicted attacks, but only 8 correspond to a real attack ; the last one is due to approximations in the software models (this echoes the fact that the fidelity is not 100%). The 8 predicted attacks correspond to the 15 actual attacks (numbers differ because it is not the same perspective; in RTL, several flip-flops provoke the same effects).

6 Conclusion and perspectives

In this paper, we have presented an approach to study the relation between RTL and software fault injections. The goal was to propose various analyses and metrics in order to go beyond current software fault models by providing feedback on the relevance of complex software models; while keeping a link between the two abstraction levels. As a result, the transition from RTL to software or from software to RTL is made easier; the approach eases both the abstraction process and the materialization process. The approach is based on the comparison between injections in RTL simulations and injections at software level thanks to a tool that mutates a program according to a configurable fault model. Various analyses have been shown to study the pertinence of software fault models: coverage metrics, fidelity metric, selection of the best software fault models, model profiling and highlighting of the uncovered scenarios. Finally, some results have been discussed, to show use-cases of the various analyses. In particular, we discussed about the coverage achieved in our experiments and showed that, under our cautious point of view, achieving a good coverage would require lots of software fault models. We also showed that our approach can help in deciding which flip-flops to protect in priority in hardware. Finally, model profiles

have also proved useful for complementing the fidelity metric: they can effectively be used to help targetting the right structures in the processor.

Once the models are validated, they can directly be used with the same mutation tool to analyze the security of various programs, and in particular to apply efficient methods such as static analysis [12]. Indeed, the approach validates the implementation of the model in our mutation tool. And the fact that the mutant is written in C means that it can be fed into various analysis tools.

Several perspectives come out of this work. The first one would be to think about new ways to analyze the databases, for example enhancing model selection to optimize other criteria such as fidelity instead of focusing on coverage. Another interesting perspective would be to add other databases for experimental results such as glitches, electromagnetic or laser injections. Here we have only compared software injection with RTL simulations; using experimental results would have two advantages: design more realistic fault models (for faults that have been produced in reality and not only in RTL simulation), and link experimental parameters to software fault models (so that a fault model can quickly be reproduced in reality). To enable comparisons between these various abstraction levels, the central point would be to define the same injection rules (same injection and observation points).

ACKNOWLEDGMENTS

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

REFERENCES

[1] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. 2011. An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 105–114. DOI:https://doi.org/10.1109/FDTC.2011.9

[2] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. 2006. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE* 94, 2 (February 2006), 370–382. DOI:https://doi.org/10.1109/JPROC.2005.862424

[3] Salma Bergaoui, Pierre Vanhauwaert, and Regis Leveugle. 2010. A New Critical Variable Analysis in Processor-Based Systems. *IEEE Transactions on Nuclear Science* 57, 4 (August 2010), 1992–1999. DOI:https://doi.org/10.1109/TNS.2010.2043540

[4] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky, and J. F. Lalande. 2012. High Level Model of Control Flow Attacks for Smart Card Functional Security. In *2012 Seventh Int. Conf. on Availability, Reliability and Security*, 224–229. DOI:https://doi.org/10.1109/ARES.2012.79

[5] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. 1997. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology — EUROCRYPT '97* (Lecture Notes in Computer Science), Springer, Berlin, Heidelberg, 37–51. DOI:https://doi.org/10.1007/3-540-69053-0_4

[6] Chun-Kai Chang, Sangkug Lym, Nicholas Kelly, Michael B. Sullivan, and Mattan Erez. 2018. Evaluating and Accelerating High-Fidelity Error Injection for HPC. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 577–589. DOI:https://doi.org/10.1109/SC.2018.00048

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

[8] Brice Colombier, Alexandre Menu, Jean-Max Dutertre, Pierre-Alain Moëllic, Jean-Baptiste Rigaud, and Jean-Luc Danger. 2019. 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)*, 1–10. DOI:https://doi.org/10.1109/HST.2019.8741030

[9] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. 2016. FISSC: A Fault Injection and Simulation Secure Collection. . 3–11. DOI:https://doi.org/10.1007/978-3-319-45477-1_1

[10] Louis Dureuil, Marie-Laure Potet, Philippe de Choudens, Cécile Dumas, and Jessy Clédière. 2015. From Code Review to Fault Injection Attacks: Filling the Gap Using Fault Model Inference. In *Smart Card Research and Advanced Applications* (Lecture Notes in Computer Science), Springer, Cham, 107–124. DOI:https://doi.org/10.1007/978-3-319-31271-2_7

[11] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. 2019. Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor. *Microprocessors and Microsystems* 71, (November 2019), 102862. DOI:https://doi.org/10.1016/j.micpro.2019.102862

[12] Johan Laurent, Christophe Deleuze, Vincent Beroulle, and Florian Pebay-Peyroula. 2019. Analyzing Software Security Against Complex Fault Models with Frama-C Value Analysis. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 33–40. DOI:https://doi.org/10.1109/FDTC.2019.00013

[13] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. 2009. Statistical fault injection: Quantified error and confidence. In *Automation Test in Europe Conference Exhibition 2009 Design*, 502–506. DOI:https://doi.org/10.1109/DATE.2009.5090716

[14] R. Leveugle and K. Hadjiat. 2002. Multi-level fault injection experiments based on VHDL descriptions: a case study. In *Proceedings of the Eighth IEEE International On-Line Testing Workshop (IOLTW 2002)*, 107–111. DOI:https://doi.org/10.1109/OLT.2002.1030192

[15] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. 2013. Formal verification of a software countermeasure against instruction skip attacks. Retrieved March 18, 2018 from https://hal-emse.ccsd.cnrs.fr/emse-00869509

[16] M. L. Potet, L. Mounier, M. Puys, and L. Dureuil. 2014. Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections. In *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*, 213–222. DOI:https://doi.org/10.1109/ICST.2014.34

[17] Julien Proy, Karine Heydemann, Fabien Majéric, Albert Cohen, and Alexandre Berzati. 2019. Studying EM Pulse Effects on Superscalar Microarchitectures at ISA Level. *arXiv:1903.02623 [cs]* (March 2019). Retrieved March 12, 2019 from http://arxiv.org/abs/1903.02623

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

[19] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. 2004. Characterizing the effects of transient faults on a high-performance processor pipeline. In *International Conference on Dependable Systems and Networks, 2004*, 61–70. DOI:https://doi.org/10.1109/DSN.2004.1311877