Scripting the Unpredictable: Automate Fault Injection in RTL Simulation for Vulnerability Assessment

William Pensec, Vianney Lapotre, Gogniat Guy

Scripting the Unpredictable: Automate Fault Injection in RTL Simulation for Vulnerability Assessment

William PENSEC
Université Bretagne Sud
UMR 6285, Lab-STICC
Lorient, France
william.pensec@univ-ubs.fr

Vianney LAPÔTRE
Université Bretagne Sud
UMR 6285, Lab-STICC
Lorient, France
vianney.lapotre@univ-ubs.fr

Guy GOGNIAT
Université Bretagne Sud
UMR 6285, Lab-STICC
Lorient, France
guy.gogniat@univ-ubs.fr

***Abstract*—This paper presents FISSA, an open-source software tool that facilitates the building of fault injection campaigns based on well-known HDL simulation tools. The proposed solution relies on two software modules that encapsulate an existing HDL simulator. The first module generates TCL scripts to drive the simulation process and automatically inject faults according to the user's needs. The second module is dedicated to fault analysis, enabling users to assess the resilience of their design. The proposed approach allows the designers to seamlessly integrate fault injection simulations into their workflow. To demonstrate the solution's capacity, this paper proposes a case study to evaluate the robustness of a Dynamic Information Flow Tracking mechanism integrated into a RISC-V processor against different fault injection scenarios. For that purpose, a total of 360,747 simulations have been performed.**

***Index Terms*—Hardware security, Physical Attacks, Fault injection simulation, Open-Source tool, Vulnerability Assessment**

I. Introduction

Internet of Things (IoT) devices have transformed the way we interact with our environment by enabling seamless data collection and analysis for increased efficiency in various domains. However, this increased connectivity also raises concerns about potential physical attacks, such as fault injection attacks (FIA) [1], [2] gaining attention in recent years.

Many studies have shown the vulnerabilities of critical systems against FIAs. In [3], authors demonstrate the possibility to recover computed secret data using FIA targeting hidden registers on the RISC-V Rocket processor. In [4], Electromagnetic Fault Injection (EMFI) attack is used to recover an AES key by targeting the cache hierarchy and the MMU. In [5], it is shown that laser fault injection (LFI) allows the replay of instructions. This leads to the overwriting of a whole program section. Finally, authors of [6] have shown that one can combine side-channel attacks (SCA) and FIAs to bypass the PMP mechanism in a RISC-V processor. It is therefore necessary, when designing a circuit, to assess its sensitivity to FIA as soon as possible.

This paper focuses on automating the evaluation of circuit design robustness against FIA using simulations at an early stage of the development cycle. The use of simulations for fault injection campaigns enables researchers to systematically explore the impact of injected faults on the targeted system in a controlled and repeatable environment. In this context, a fault injection campaign refers to an organised series of simulated events designed to assess the system's robustness, identify vulnerabilities, and evaluate its resilience under various fault conditions. This approach allows for a comprehensive examination of the system's behaviour across multiple scenarios, providing valuable insights into potential weaknesses and helping to develop robust countermeasures.

In this paper, we present FISSA (Fault Injection Simulation for Security Assessment), a new tool, designed to generate fault injection campaigns relying on existing simulation environments (HDL simulator) such as Questasim, Vivado or Verilator. The proposed approach allows the designer to rely on the same environment for both design flow and security assessment against FIA attacks, providing valuable information for the development of effective defence mechanisms. The interest of FISSA is demonstrated in a case study focusing on the D-RI5CY processor [7]. We present the tool workflow when using the Questasim HDL simulator. Finally, we present and discuss a set of results obtained through FISSA.

The rest of the paper is structured as follows. Section II presents related work. Section III presents the proposed tool. Section IV details our case study and discusses obtained results. Finally, Section V concludes the work and draws some perspectives.

II. Related Work

This section presents recent works related to methods and tools for vulnerability assessment when considering fault injection attacks. For such vulnerability assessment, main strategies include actual fault injections, emulations, formal methods and simulations. In this paper, we are focusing into fault simulation tools for application security and not for safety, i.e [8].

Actual FIAs involve physically injecting faults into the target hardware using techniques such as variations in supply

TABLE I
FAULT INJECTION BASED METHODS FOR VULNERABILITY ASSESSMENT COMPARISON

	References	Cost	Control over fault scenarios	Scalability	Speed of execution	Realism	Expertise
Formal Methods	[9]–[12]	Very low	Very high	Very low	Low	Low	Very high
RTL Simulations	[13]–[15]	Very low	Very high	Low	Low/Moderate	Moderate	Low
Emulations	[16]–[19]	High	Moderate	High	Very high	High	Moderate
Actual FIA	[1], [20]–[22]	Very high	Very low	Very high	Very high	Very high	Very high

voltage or clock signal [1], [20], laser pulses [1], [22], electromagnetic emanations [1] or X-rays [21]. This approach offers valuable insights into the real impact of faults on hardware components. However, a significant drawback of actual fault injections is that they demand considerable expertise to prepare the target, involving intricate setup procedures. Additionally, this approach can only be executed once the physical circuit is available, potentially delaying the vulnerability assessment process until later stages of development.

Fault emulation can, for instance, rely on FPGA [16], or on an emulator such as QEMU [17], [18] to perform fault injection campaigns. This approach is four times faster than simulation-based techniques [19], and unlike simulation-based or formal method-based fault injections techniques, the size of the evaluated circuit has no major impact on the fault injection campaign timing performances. However, configuring an emulation environment can be complex and time-consuming. Achieving an accurate representation of the target system may require detailed configuration and parameter tuning. The accuracy of emulation is contingent on the quality of the models used to replicate the target hardware. If the models are inaccurate or incomplete, the results of fault injections may not precisely reflect actual behaviour.

Formal methods provide an advantage with mathematical proofs, ensuring a rigorous verification of the system's behaviour during fault injection experiments. Formal methods approaches such as [9] allow the analysis of a circuit design in order to detect sensitive logic or sequential hardware elements. [10], [11] and [12] present formal verification methods to analyse the behaviour of HDL implementation. However, this type of tool usually suffers from restrictions limiting its actual usage on a complete processor. Conventional formal approaches encounter scalability challenges due to limitations in verification techniques. In particular, the circuit structure it can analyse is usually limited.

Fault Injections simulations can be performed at processor instructions level. Authors of [13] explore the impact of fault injection attacks on software security. They evaluate four open-source fault simulators, comparing their techniques and suggest enhancing them with AI methods inspired by advances in cryptographic fault simulation. [14] is an open-source deterministic fault attack simulator prototype utilising the Unicorn Framework and Capstone disassembler. [15] introduces VerFI, a gate-level granularity fault simulator for hardware implementations. For instance, it has been used to spot an implementation mistake in ParTI [23]. However, this tool has been developed to check if implemented countermeasures can really protect against fault injection on cryptographic implementations, but it cannot evaluate components such as registers or memories. In this paper, we focus on RTL simulations at Cycle Accurate Bit Accurate (CABA), which provides a controlled virtual environment for injecting faults. There are several solutions of simulations in an HDL simulator like Questasim, Vivado, etc. *Behavioural* simulation is used to detect functional issues and ensuring that the design behaves as expected. *Post-synthesis* simulation verifies that the synthesised netlist matches the expected functionality. *Timed* simulation is used to ensure that the design meets timing requirements and can operate at the specified clock frequency. And finally, *post-implementation* simulations are used to verify that the implemented design meets all requirements and constraints, including those related to the physical layout on the target. Simulation-based fault injection offers the advantage of enabling designers to test their system throughout the design cycle, providing valuable insights and uncovering potential vulnerabilities early in the development process. However, a limitation lies in the potential lack of absolute fidelity to actual conditions, as simulations might not perfectly replicate all hardware intricacies, introducing a slight risk of overlooking certain faults that could manifest in the actual hardware.

Table I shows a comparison between these four methods for vulnerability assessment when considering FIA regarding six metrics. These metrics are the financial cost of setting up the fault injection campaign, the control over fault scenarios (how configurable are the scenarios), scalability which refers to the method capacity to be applied to systems of different sizes or complexities, speed of execution of the campaign, realism of the fault injection campaign and the level of required expertise. Table I shows that no method is completely optimal. Each method has its own advantages and disadvantages and must be chosen by the designer according to the requirements and the available financial and human resources. Indeed, setting up an actual fault injection campaign requires much more expertise in this domain and also requires costly equipment, whereas setting up a simulation campaign can be easier for a circuit designer familiar with HDL simulation tools such as Questasim. Table I shows that RTL simulation offers a good compromise to assess the security level of a circuit design.

In particular, it provides an efficient solution for investigating security throughout the design cycle, enabling the concept of "Security by Design".

III. FISSA

This section presents our open-source tool, FISSA[1], to help circuit designers to analyse, throughout the design cycle, the sensitivity to FIA of the developed circuit. Figure 1 presents the software architecture of FISSA. It consists of 3 different modules: *TCL generator*, *Fault Injection Simulator* and *Analyser*. The first and third modules correspond to a set of Python classes.

The TCL generator, detailed in Section III-B, relies on a configuration file and a target file to create a set of parameterised TCL scripts. These scripts are tailored based on the provided configuration file and are used to drive the fault injection simulation campaign.

Fault Injection Simulator, detailed in Section III-C, performs the fault injection simulation campaign based on inputs files from *TCL generator* for a circuit design described through HDL files and memory initialisation files. For that purpose it relies on an existing HDL simulator such as Questasim [24], Verilator [25], or Vivado [26].

The Analyser, detailed in Section III-D, evaluates the outcomes of the simulations and generates a set of files that allows the designers to examine fault injection effects on their designs through various information.

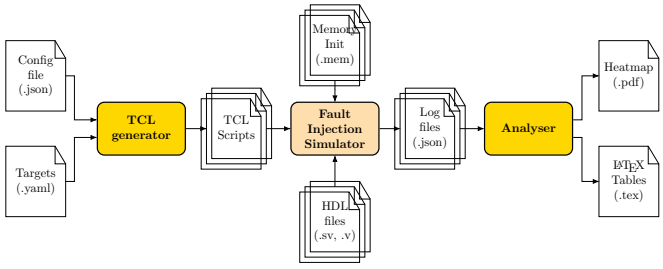

Fig. 1. Software architecture of FISSA

Algorithm 1 shows a representation of a fault injection campaign. The algorithm requires a set of targets (i.e. hardware elements in which a fault should be injected), the fault model and the considered injection window(s) which identifies the period(s), in number of clock cycles, in which fault injections are performed. Then, it runs a first simulation with no fault injected, which is used as a reference for comparison with the following simulations to determine end-of-simulation statuses. Then, for each target, each fault model and for each clock cycle within the injection window, the corresponding simulation is executed, and the corresponding logs are stored in a dedicated file. Customising end-of-simulation statuses allows for adaptation to the specific requirements of each design assessment. To configure these statuses, adjustments need to be made either directly in FISSA's code or the HDL code. This process may involve evaluating factors such as:

[1]https://github.com/WilliamPsc/FISSA/tree/main

- hardware element content (signal, registers, ...),
- simulation time (e.g. the simulation exceeds a reference number of clock cycles),
- simulation's end (e.g. an assert statement introduced in the HDL code is reached)

Algorithm 1 Simulated FIA campaign pseudo-code

Require: $targets \leftarrow list(targets)$
Require: $faults \leftarrow list(fault\_model)$
Require: $windows \leftarrow list(injection\_windows)$
$ref\_sims = simulate()$
for $target \in targets$ **do**
  for $fault \in faults$ **do**
    for $cycle \in windows$ **do**
      $logs = simulate(target, fault, cycle)$
    end for
  end for
end for

A. Supported Fault models

A set of fault models has already been integrated into FISSA. For a given fault injection campaign, the relevant fault model is defined in the input configuration file and is applied to targets during the simulation phase. Currently, supported fault models are:

- target set to 0/1,
- single bit-flip in one target at a given clock cycle,
- single bit-flip in two targets at a given clock cycle,
- single bit-flip in two targets at two different clock cycles,
- exhaustive multi-bits faults in one target at a given clock cycle,
- exhaustive multi-bits faults in two targets at a given clock cycle.

B. TCL Generator

The *TCL Generator* is used to generate the set of TCL script files which drive the *fault injection simulator*. This module requires two input files.

Figure 2 details the *TCL Generator*. Each blue box represents a python class used to generate the set of output TCL scripts. The initialisation class gets inputs from a configuration file. This JSON-formatted file includes various parameters such as the targeted HDL simulator, the considered fault model and the injection window(s). Furthermore, it encompasses parameters such as the clock period (in ns) of the HDL design and the maximum number of simulated clock cycles used to stop the simulation in case of divergence due to the injected fault. Moreover, one extra parameter defines the quantity of simulations per TCL file, allowing a simulation parallelism degree. The *Targets* file contains, in YAML format, the list of the circuit elements (e.g. registers or logic gates) that need to be targeted during the fault injection campaign. For each target, its HDL path and bit-width are specified. *TCL Script Generator* class gets the configuration parameters from *Initialisation* class, reads the *Targets'* file and calls three

others classes. The first one, *Basic Code Generator*, undertakes the fundamental generation of TCL code for initialising a simulation, running a simulation, and ending a simulation. The second one, *Fault Generator*, produces the TCL code related to fault injection. The *TCL Script Generator* provides specific parameters to the *Fault Generator* to produce code for a designated set of targets and a specified set of clock cycles for fault injection. The third one, *Log Generator*, produces the TCL code to produce logs after each simulation. Logs comprise the simulation's ID, fault model, faulted targets, injection clock cycle(s), end-of-simulation status, values for all targets, and the end-of-simulation clock cycle. This data constitutes the automated aspect of logging. Finally, the *TCL Script Generator* outputs a set of TCL files, each one correspond to a batch of simulations. It is worth noting that each batch starts with a reference simulation (i.e. without fault injection). This allows the user to perform a per batch results analysis. Furthermore, it produces a target file used by TCL scripts to get the target list (see Subsection III-C).

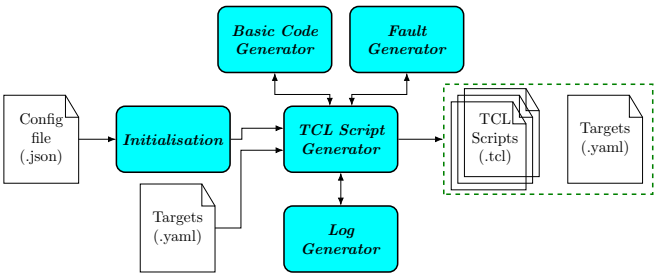

Fig. 2. Software architecture of the TCL Generator module

Algorithm 2 depicts a fault injection simulation pseudo-code, showcasing requirements, each state with essential parameters, and the corresponding Python class from Figure 2. Line 5 in Algorithm 1 corresponds to Algorithm 2. This algorithm is executed multiple times with different inputs to build a TCL script.

Algorithm 2 FIA simulation pseudo-code

Require: $target$
Require: $cycle$
Require: $fault\_model$
1: $tcl\_script = init\_sim(fault\_model, cycle, target)$ // generated by Basic Code Generator
2: $tcl\_script+ = inject\_fault(fault\_model)$ // generated by Fault Generator
3: $tcl\_script+ = run\_sim()$ // generated by Basic Code Generator
4: $tcl\_script+ = log\_sim(fault\_model)$ // generated by Log Generator
5: $tcl\_script+ = end\_sim()$ // generated by Basic Code Generator
6: $tcl\_file.write(tcl\_script)$

C. Fault Injection Simulator

The *Fault Injection Simulator* mainly relies on an existing HDL simulator to perform simulations by executing the TCL scripts produced by the *TCL generator*. The log files, in JSON format, are generated by the TCL script for each simulation. This file encompasses data such as the current simulation number, the executed clock cycle count, the values of the targets' file, the targets faulted, the fault model and the end-of-simulation status. It is worth noting that the set of calls to the generated TCL scripts has to be integrated into the designer's existing design flow, allowing the design compilation, initialisation, and management of input stimuli. The use of TCL scripts simplifies such an integration. Once all the fault injection simulations have been performed, the log files can be sent to the *Analyser* which, is described in the following subsection.

D. Analyser

The *Analyser* reads all log files and generates a set of LaTeX tables (.tex files) and/or sensitivity heatmaps (in PDF format) according to the fault models, allowing the user to identify the sensitive hardware elements in the circuit design. The generated tables can be customised through modification in the *Analyser* Python code. The current configuration captures and counts the diverse end-of-simulation status (see Section IV-C for an example). Heatmaps are generated for multi-target fault models. For instance, when considering a 2 faults scenario disturbing two hardware elements, a 2-dimension heatmap allows the user to identify sensitive couples of hardware elements leading to a potential vulnerability. Their configuration can be adapted by modifying the *Analyser* Python code. Heatmaps generation is based on *Seaborn* [27] which relies on *Matplotlib* [28]. This library provides a high-level interface for drawing attractive and informative statistical graphics and save them in different formats like PDF, PNG, etc. In the current configuration, heatmaps highlight the targets leading to a specific end-of-simulation status (e.g. a status identified by the designer as a successful attack). An example of such heatmap is presented in Subsection IV-C. Once the results have been generated, they can easily be inserted into a vulnerability assessment report.

E. Extending FISSA

In order to extend FISSA for integrating an additional fault model, some modifications to the *TCL Script Generator*, the *Basic Code Generator*, the *Fault Generator* and *Log Generator* modules are necessary. It requires the extension of the *init_sim*, *inject_fault* and *log_sim* functions presented in Algorithm 2 to implement the new fault model from initialisation to logging. For instance, these extensions should define the targets for each simulation, the impact of the injections (set to 0/1, bit-flip, random, etc) and the set of data to be logged for this fault model. The *Log Generator* automates the extraction of specific segments from the ongoing simulation. However, it is customisable, enabling the modification of logged elements, such as incorporating memory content or a list of signals.

Analyser can be extended to produce additional LaTeX tables, heatmaps or any other way of results visualisation. This can

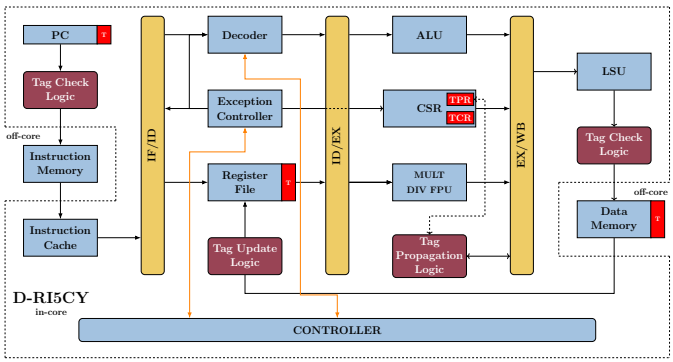

Fig. 3. D-RI5CY processor architecture overview

be achieved by either modifying the existing methods or by developing new ones.

An integral aspect of expanding FISSA involves adjusting functions depending on the used HDL simulator. Despite the definition of the TCL language, specific commands vary between simulators. For instance, in Questasim, injecting a fault into a target can be accomplished with the command: "*force <object_name><value>-freeze -cancel <time_info>*" [29], whereas in Vivado, the equivalent command is: "*add_force <hdl_object><values>-cancel_after <time_info>*" [30].

IV. Use Case

This section presents a case study to demonstrate the interest of the proposed solution. It focuses on the evaluation of the robustness of the Dynamic Information Flow Tracking (DIFT) mechanism integrated to the D-RI5CY [7] processor.

A. D-RI5CY DIFT mechanism

The D-RI5CY core is a 4-stage in-order 32-bit RISC-V processor. It introduces a Dynamic Information Flow Tracking (DIFT) mechanism to protect the processor against software attacks such as buffer overflows or SQL injections. Figure 3 presents an overview of the D-RI5CY processor. DIFT-related modules are highlighted in red. These modules allow storing, propagating and checking tags during the execution of an application. The security policy is configured through two CSRs (*Configuration and Status Register*) named TPR (*Tag Propagation Register*) and TCR (*Tag Check Register*). The *Tag Update Logic* module is used to initialise or update the tag in the register file according to the tagged data. Then, when a tag is propagated in the pipeline, the *Tag Propagation Logic* module propagates tags according to the security policy defined in the TPR. Once a tag has been propagated and its data has been sent out of the pipeline, the *Tag Check Logic* modules check that it conforms to the security policy defined in the TCR. If not, an exception is raised.

In this paper, we rely on FISSA presented in Section III to study the behaviour of the DIFT D-RI5CY mechanism against different fault injection scenarios.

For this use case, we consider a software application in which a buffer overflow can be exploited to perform a Return-Oriented Programming attack (ROP)[2]. Thanks to the DIFT mechanism, such an exploitation is detected and stopped. However, a circuit designer may want to study the effect of FIA on such a mechanism.

B. FISSA's configuration

This subsection presents FISSA's configuration to address the considered use case.

We have configured four end-of-simulations statuses. End-of-simulation statuses will be used to automatically generate results tables. Examples will be provided in Subsection IV-C. The initial status is labelled as a *crash* (status 1), signifying that the fault injection has led to a deviation in program flow control, causing the processor to execute instructions different from what is expected. The second status, identified as a *silent* fault (status 2), indicates that the fault has occurred but has not affected the ongoing simulation behaviour. Status 3, termed a *delay*, signifies that the fault has delayed the DIFT-related exception (i.e. the exception is not raised at the same clock cycle compared to the reference simulation). The final status is denoted as a *success* (status 4), highlighting a bypass of the DIFT mechanism and consequently marking a successful attack. This status corresponds to the detection of the end of the simulated program, while no exception has been raised.

In the input configuration file, a single injection window is set between cycles 3428 and 3434, the maximum number of simulated clock cycles is set to 100 from the start of the injection window, this allows us to detect if there were a control flow deviation, the design period is set to 40 ns, the number of simulations per TCL script is set to 2,200. The considered fault models are the seven fault models defined in Section III : *target set to 0*, *target set to 1*, *single bit-flip in one target at a given cycle*, *single bit-flip in two targets at a given cycle*, *single bit-flip in two targets at two different cycles*, *exhaustive multi-bits faults in one target at a given cycle*, *exhaustive multi-bits faults in two targets at a given cycle*.

Seven FIA simulation campaigns are performed for the design and the seven fault models. We choose to log the values of the *Targets'* file, the simulation's number, targets' value after the injection, the injection cycle and the end-of-simulation status. The *Targets'* file is filled with 55 registers representing a total of 127 bits.

C. Experimental results

This section presents results obtained using FISSA on the considered use case. All experiments are performed on a server with the following configuration: Xeon Gold 5220 (2,2 GHz, 18C/36T), 128 GB RAM, Ubuntu 20.04.6 LTS and Questasim 10.6e.

Table II summarises the outcomes of the seven previously described fault injection campaigns, with each row representing a distinct fault model. Table II's columns delineate the potential end statuses for each simulation. This table is an

[2]https://github.com/sld-columbia/riscv-dift/tree/master/pulpino_apps_dift/wilander_testbed

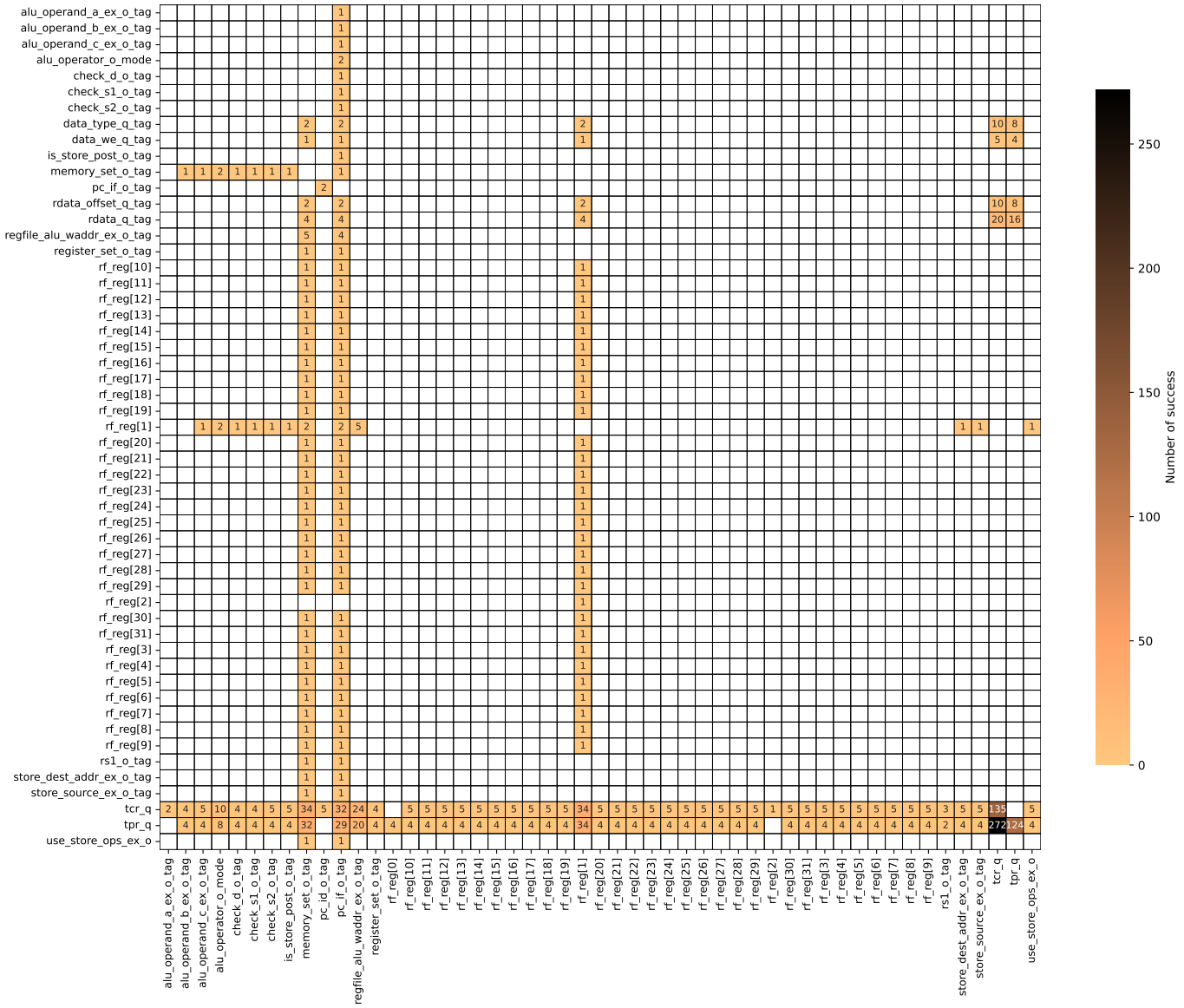

Fig. 4. Heatmap generated according to the single bit-flip in two targets at a given clock cycle fault model

essential tool for the designers, enabling them to analyse the vulnerabilities associated with each fault model within their design. Consequently, the designers can determine the necessity for additional protective measures or design alterations.

For instance, Table II illustrates that the '`set to 1`' fault model results in only three successful outcomes, whereas the '`single bit-flip in two targets at two different clock cycles`' fault model leads to 2,159 successes. These findings guide the designers in evaluating the significance of protecting against specific fault models.

To further assess vulnerabilities, the designers can utilise Table III, which provides detailed information on the register and cycle locations of faults for models with fewer successful outcomes. For fault models with a high number of successes, where the table may become unwieldy, Figure 4 serves as a more accessible reference. This figure helps in visualising and interpreting the spatial distribution of vulnerabilities effectively.

Table III is produced by FISSA and details the successes from three distinct fault injection campaigns: `set to 0`, `set to 1` and `single bit-flip in one target at a given cycle`. Table III specifies successes for each fault model, correlated with the cycle and the affected target. For example, a `set to 0` fault at cycle 3428 on `tcr_q` would lead to a successfully attack. It highlights which targets are sensitive to fault attacks at a cycle-accurate and bit-accurate level, providing the designers precise information on critical elements requiring protection based on their specific needs. Table III only covers the most basic fault models. Indeed, producing a table for more complex scenarios, such as

TABLE II
RESULTS OF FAULT INJECTION SIMULATION CAMPAIGNS

Fault model	Crash	Silent	Delay	Success	Total
Set to 0	0	320	1	9 (2.73%)	330
Set to 1	0	320	7	3 (0.91%)	330
Single bit-flip in one target at a given clock cycle	0	738	12	12 (1.57%)	762
Single bit-flip in two targets at a given clock cycle	0	45,097	1,503	1,406 (2.93%)	48,006
Single bit-flip in two targets at two different clock cycles	0	238,633	1,143	2,159 (0.89%)	241,935
Exhaustive multi-bits faults in one target at a given clock cycle	0	927	6	3 (0.32%)	936
Exhaustive multi-bits faults in two targets at a given clock cycle	0	67,072	926	450 (0.66%)	68,448

TABLE III
BUFFER OVERFLOW: SUCCESS PER REGISTER, FAULT TYPE AND SIMULATION TIME

	Cycle 3428			Cycle 3429			Cycle 3430			Cycle 3431			Cycle 3432		
	set 0	set 1	bit-flip	set 0	set 1	bit-flip	set 0	set 1	bit-flip	set 0	set 1	bit-flip	set 0	set 1	bit-flip
pc_if_o_tag										✓		✓			
memory_set_o_tag		✓	✓												
rf_reg[1]							✓		✓						
tcr_q	✓			✓			✓			✓			✓		
tcr_q[21]			✓			✓			✓			✓			✓
tpr_q	✓	✓		✓	✓										
tpr_q[12]			✓			✓									
tpr_q[15]			✓			✓									

simultaneous faults in two targets within a same or multiple cycles, would be intricate and challenging to interpret. Consequently, we opted for an alternative method and developed a heatmap representation (e.g. Figure 4).

To further explore the impact of FIA on a design, a designer can study heatmaps generated by FISSA. These heatmaps are tailored to a fault model with two faulty registers, where each matrix intersection shows the number of successes with that target pair.

Figure 4 shows the heatmap generated for the single bit-flip in two targets at a given clock cycle fault model. The colour scale represents the number of fault injections targeting a couple of hardware elements (i.e. registers for this use case) leading to a *success* as defined in Subsection IV-B. We can note that this colour scale, in our case, range from 1 to 272 with 0 excluded. This figure highlights the registers that are critical to a specific fault model, allowing the designer to assess his design and choose which protection and where a protection is required, from low need to very high need. To give an example, it can be noted that the horizontally displayed registers `tcr_q` and `tpr_q` are critical registers, because a success will occur regardless of the associated register. Similarly, the registers shown vertically, `memory_set_o_tag`, `pc_if_o_tag`, and `rf_reg[1]`, are also critical because they lead to many successes with almost all tested registers.

To provide an analytical perspective from the buffer overflow use case presented in Section IV, the five previously mentioned registers are critical as they either store the DIFT security policy configuration (`tpr_q` and `tcr_q`) or store (`rf_reg[1]` represents the tag associated with the value of the Program Counter (PC), which is stored in the register file at index 1 for RISC-V ISA) and propagate the tag (`pc_if_o_tag`) associated with the PC. This is particularly important in our example, which demonstrates an ROP attack via a buffer overflow. The colour scale indicates the impact of the fault injections on the combination of registers tested. For example, a pair associated with a high number such as 272, 124, and 135 for `tcr_q` and `tpr_q` are very high priority as they lead to 37.77% success on this fault model. In addition, we can see that several registers produce a low number of successes, such as `alu_operand_a_ex_o_tag` and `rf_reg[2]`; these registers are then not the highest priority for protection for the designer.

It allows the designer to identify the critical hardware elements to be protected for the use case under consideration. All of this information allows the designer to prioritize countermeasures according to allocated budget, protection requirements, etc.

While Table II provides the total number of *successes* for each fault model and Table III gives the successes for each fault model (`set to 0`, `set to 1`, and `a single bit flip in a target at a given cycle`) correlated with the cycle and affected target, Figure 4 shows that fault injections in 246 register pairs result in a *success*. This information allows the designer to focus on specific simulation traces to understand the effect(s) of the fault(s) and improve the robustness of his design by implementing adapted countermeasures.

To conclude this section, we aim to provide an overview of the time invested in these simulations. The entire fault injection campaign comprises 360,747 simulations, involving the injection of 1 or 2 faults per simulation. The duration of

these simulations was about 67 hours, equating to an average of 0.725 seconds per simulation, spanning from initialisation to data recording. The execution time is contingent upon various parameters, including the design's size, the specific simulation case, and the number of targets involved. In emulation campaigns, FPGA-based fault emulation is four times faster than simulation-based techniques, as noted in paper [19]. Actual FIAs are faster than simulations, taking about 0.35 seconds per injection in our tests, relying on the ChipWhisperer-lite platform for clock glitching injection. While simulations may be slower, they offer the benefit of not requiring an FPGA prototype or the final circuit. Furthermore, it allows integrating vulnerability assessment in the first stages of the development flow and provides a rich set of information for the designer in order to understand sources of vulnerabilities in his design.

V. Conclusion

This paper introduces a flexible open-source tool, FISSA, to automate fault injection campaigns and illustrates its usage through a case study. FISSA seamlessly integrates with well known HDL simulators such as Questasim. It generates TCL scripts for simulation execution and produces JSON log files for subsequent security analysis. We have demonstrated that this tool can be incorporated to evaluate a design at the conceptual stage. The designer is able to select the parameters of the assessment, such as the fault model and targets, to meet their specific requirements. Results generated by the tool will assist the designer in creating a more secure design, embodying the principles of *Security by Design*.

As a perspective, we plan to extend FISSA to support new fault models and HDL simulators such as Vivado or Verilator. Furthermore, we intend to enhance integration into the design workflow by adding functionalities. This may include the management of HDL sources compilation, design's input stimuli or the development of a graphical user interface to improve the overall user experience.

References

[1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, 2006. DOI: 10.1109/JPROC.2005.862424.

[2] D. Karaklajić, J.-M. Schmidt, and I. Verbauwhede, "Hardware Designer's Guide to Fault Attacks," *IEEE Transactions on Very Large Scale Integration Systems*, 2013. DOI: 10.1109/TVLSI.2012.2231707.

[3] J. Laurent, V. Beroulle, C. Deleuze, and F. Pebay-Peyroula, "Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures," in *Design, Automation & Test in Europe Conference (DATE)*, 2019. DOI: 10.23919/DATE.2019.8715158.

[4] T. Trouchkine, S. K. K. Bukasa, M. Escouteloup, R. Lashermes, and G. Bouffard, "Electromagnetic Fault Injection Against a Complex CPU, toward new Micro-architectural Fault Models," *Journal of Cryptographic Engineering*, 2021. DOI: 10.1007/s13389-021-00259-6.

[5] V. Khuat, J.-M. Dutertre, and J.-L. Danger, "Analysis of a Laser-induced Instructions Replay Fault Model in a 32-bit Microcontroller," in *Digital System Design (DSD)*, 2021. DOI: 10.1109/DSD53832.2021.00061.

[6] S. Nashimoto, D. Suzuki, R. Ueno, and N. Homma, "Bypassing Isolated Execution on RISC-V using Side-Channel-Assisted Fault-Injection and Its Countermeasure," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021. DOI: 10.46586/tches.v2022.i1.28-68.

[7] C. Palmiero, G. Di Guglielmo, L. Lavagno, and L. P. Carloni, "Design and Implementation of a Dynamic Information Flow Tracking Architecture to Secure a RISC-V Core for IoT Applications," in *High Performance Extreme Computing*, 2018. DOI: 10.1109/HPEC.2018.8547578.

[8] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "Rt-level fault simulation techniques based on simulation command scripts," in *DCIS 2000: XV Conference on Design of Circuits and Integrated Systems*, 2000, pp. 21–24.

[9] J. Richter-Brockmann, A. Rezaei Shahmirzadi, P. Sasdrich, A. Moradi, and T. Güneysu, "FIVER – Robust Verification of Countermeasures against Fault Injections," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021. DOI: 10.46586/tches.v2021.i4.447-473.

[10] V. Arribas, S. Nikova, and V. Rijmen, "VerMI: Verification Tool for Masked Implementations," in *25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2018. DOI: 10.1109/ICECS.2018.8617841.

[11] G. Barthe, S. Belaïd, G. Cassiers, P.-A. Fouque, B. Grégoire, and F.-X. Standaert, "maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults," in *Computer Security – ESORICS 2019: 24th European Symposium on Research in Computer Security, Proceedings, Part I*, 2019. DOI: 10.1007/978-3-030-29959-0_15.

[12] S. Tollec, V. Hadžić, P. Nasahl, *et al.*, "Fault-resistant partitioning of secure cpus for system co-verification against faults," 2024. [Online]. Available: https://eprint.iacr.org/2024/247.

[13] A. Adhikary and I. Buhan, "SoK: Assisted Fault Simulation," in *Applied Cryptography and Network Security Workshops*, Springer Nature Switzerland, 2023. DOI: 10.1007/978-3-031-41181-6_10.

[14] Riscure, *FiSim: An open-source deterministic Fault Attack Simulator Prototype*. [Online]. Available: https://github.com/Riscure/FiSim.

[15] V. Arribas, F. Wegener, A. Moradi, and S. Nikova, "Cryptographic Fault Diagnosis using VerFI," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020. DOI: 10.1109/HOST45689.2020.9300264.

[16] G. Canivet, P. Maistri, R. Leveugle, J. Clédière, F. Valette, and M. Renaudin, "Glitch and laser fault attacks onto a secure AES implementation on a SRAM-based FPGA," *Journal of cryptology*, 2011. DOI: 10.1007/s00145-010-9083-9.

[17] F. Hauschild, K. Garb, L. Auer, B. Selmke, and J. Obermaier, "ARCHIE: A QEMU-Based Framework for Architecture-Independent Evaluation of Faults," in *Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, 2021. DOI: 10.1109/FDTC53659.2021.00013.

[18] Y. B. Bekele, D. B. Limbrick, and J. C. Kelly, "A Survey of QEMU-Based Fault Injection Tools & Techniques for Emulating Physical Faults," *IEEE Access*, 2023. DOI: 10.1109/ACCESS.2023.3287503.

[19] R. Nyberg, J. Nolles, J. Heyszl, D. Rabe, and G. Sigl, "Closing the Gap between Speed and Configurability of Multi-bit Fault Emulation Environments for Security and Safety-Critical Designs," in *17th Euromicro Conference on Digital System Design*, 2014. DOI: 10.1109/DSD.2014.39.

[20] C. Bozzato, R. Focardi, and F. Palmarini, "Shaping the Glitch: Optimizing Voltage Fault Injection Attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019. DOI: 10.13154/tches.v2019.i2.199-224.

[21] P. Grandamme, L. Bossuet, and J.-M. Dutertre, "X-Ray Fault Injection in Non-Volatile Memories on Power OFF Devices," in *2023 IEEE Physical Assurance and Inspection of Electronics (PAINE)*, 2023. DOI: 10.1109/PAINE58317.2023.10318018.

[22] B. Colombier, P. Grandamme, J. Vernay, *et al.*, "Multi-Spot Laser Fault Injection Setup: New Possibilities for Fault Injection Attacks," in *Smart Card Research and Advanced Applications*, V. Grosso and T. Pöppelmann, Eds., 2022. DOI: 10.1007/978-3-030-97348-3_9.

[23] T. Schneider, A. Moradi, and T. Güneysu, "ParTI–towards combined hardware countermeasures against side-channel and fault-injection attacks," in *Advances in Cryptology–CRYPTO: 36th Annual International Cryptology Conference, Proceedings, Part II 36*, 2016. DOI: 10.1007/978-3-662-53008-5_11.

[24] Siemens, *QuestaSim*. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/.

[25] Verilator, *Verilator*. [Online]. Available: https://github.com/verilator/verilator.

[26] Xilinx, *Vivado Design Suite*. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html.

[27] M. L. Waskom, "Seaborn: statistical data visualization," *Journal of Open Source Software*, 2021. DOI: 10.21105/joss.03021.

[28] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, 2007. DOI: 10.5281/zenodo.7697899.

[29] Microsemi, *Modelsim reference manual 10.4c*. [Online]. Available: https://www.microsemi.com/document-portal/doc_view/136364-modelsim-me-10-4c-command-reference-manual-for-libero-soc-v11-7.

[30] Xilinx, *Vivado reference manual 2023.2*. [Online]. Available: https://docs.xilinx.com/r/en-US/ug835-vivado-tcl-commands/add_force.