# SoK: The apprentice guide to automated fault injection simulation for security evaluation

Asmita Adhikary<sup>1</sup>, Giacomo Tommaso Petrucci<sup>1</sup>, Philippe Tanguy<sup>2</sup>, Vianney Lapôtre<sup>2</sup>, and Ileana Buhan<sup>1</sup>

<sup>1</sup> Radboud University, Nijmegen, The Netherlands {asmita.adhikary,giacomo.petrucci,ileana.buhan}@ru.nl <sup>2</sup> Université Bretagne Sud, Lorient, France {philippe.tanguy,vianney.lapotre}@univ-ubs.fr

**Abstract.** Identifying and mitigating vulnerable locations to fault injections requires significant expertise and expensive equipment. Fault injections can damage hardware, cause software crashes, and pose safety and security hazards. Simulating fault injections offers a safer alternative, and fault simulators have steadily developed, though they vary significantly in functionality, target applications, fault injection methods, supported fault models, and guarantees. We present a taxonomy categorizing fault simulators based on their target applications and development cycle stages, from source code to final product. Our taxonomy provides insights and comparisons to highlight open problems.

Keywords: Fault injection  $\cdot$  Fault simulators  $\cdot$  Fault models  $\cdot$  Sensitive region  $\cdot$  Mutation.

### 1 Introduction

Fault injection [41] is the practice of forcing an application, whether hardware or software, outside its specified functioning range, thus inducing computation errors. However, not all faults lead to a successful attack. Combining fault injection and exploiting the *fault effects* caused is known as a *fault attack*.

Some well-known examples of fault injection would be DRAM Rowhammer [67], CPU overclocking [63], etc. Fault attacks can be used to circumvent security mechanisms, such as a secure boot mechanism, since fault injections corrupt the execution flow of an application. Hence, fault injections/attacks must be eliminated as much as possible.

Recently, fault testing using simulators has been taking precedence. Using fault simulators could be advantageous since they are cost-effective and require less expertise than performing fault injection on a physical target. Furthermore, fault simulators can be used during design time, i.e., throughout the development cycle as illustrated in Figure 1, which provides the additional benefit of protecting against fault injections right from the source code to the binary in case of software applications, and from the hardware description to the FPGA/ASIC for hardware applications. Systematically addressing vulnerable points at each level can significantly reduce overall vulnerabilities. Furthermore, it diminishes the need for the feedback loop to resolve faulty points.

Naturally occurring faults have simpler, fewer, and first-order *fault models*, and hence, systems susceptible to such faults are mostly checked for dependability to prevent fault effects from getting manifested and causing the system to malfunction. However, from the security perspective, faults can be induced using complex fault models to intentionally leverage effects induced to design an

attack ranging from privilege escalation to secret key extraction. Undoubtedly, safety-driven simulators [35,30,38,36,20,14,3,9,60,44,58,42,62,13,23,71,31,8,19] would protect applications from targeted faults, but only to a limited extent, as illustrated in [70]. In this paper, we mainly focus on fault simulation tools catered towards the security of an application.

Fault simulators are designed to inject faults into hardware/software applications. Certain specialized fault simulators are designed for cryptographic applications [39,2,73,11,37,54,56,10,53,52,55,29], [49,4,34,61,50,46,51,33], which induce faults to leak sensitive data, that could be used to retrieve the secret key used for encryption via differential fault analysis (DFA) [57,32]. Using such cryptographic fault simulators requires prior knowledge in the field of cryptography. Furthermore, while some tools work on the algorithmic level, others work on cryptographic implementations. These tools target only block ciphers and utilize different cryptanalytic attack techniques as the foundation of such tools. Even though cryptographic algorithms lie at the heart of security applications, they are not the only attackable targets and, certainly, not the weakest link. There are other aspects of such applications which can be attacked and bypassed to cause the application to malfunction. Usually, bypassing such security measures might be more attractive than knowing the cryptographic algorithm to retrieve the key. Hence, these remain out of scope because they focus on cryptographic techniques and properties to analyze the faults and use cryptanalytic attacks to exploit them, which deems them too specific to be used elsewhere. The tools we explore in this paper have a more extensive scope and fewer prerequisites. Hence, a generic fault injection simulator can evaluate cryptographic designs, but the converse is not true.

A study on fault simulators in [1] showcases four *open-source* simulators working upon binaries only and classifies tools under hardware/software categories. It provides an in-action comparison, having run experiments. In this work, we investigate the state-of-the-art fault simulators from the perspective of a novice prospective user and provide a global view by considering fault simulators fit for different stages in the development cycle.

We introduce a classification for fault simulator tools based on the target application development stage. Recognizing the right tool and its application stage is essential for conducting efficient and comprehensive fault simulations. Our classification helps identify the appropriate fault injection tools and the specific stages at which they can be most effectively applied. We conclude with messages to prospective users.

The reader of this paper is a software/hardware developer who needs to test a design's resilience to fault attacks, has little expertise with fault injection tools, and is looking for a cost-effective, efficient, and user-friendly testing approach. However, our work is not intended to guide the target selection for a fault attack.

**Contribution.** In this paper, we make the following contributions:

- 1. We provide a novel taxonomy for fault simulators, classifying them according to the development stages they are used in.
- 2. We delve into how they are built and their functioning and showcase the key contributions of each tool.
- 3. We outline fault simulation limitations, current progress, and promising research directions.
- 4. We give recommendation for developing fault simulators to the community, developers, and researchers.

The paper is organized as follows. In Section 2, we present a taxonomy to classify fault simulators based on their usage in the different stages of development. Section 3 and Section 4 focus on describing fault simulators for software and hardware targets, respectively. We elaborate on the

functioning of each tool, focusing on sensitive regions and fault models used to perform mutation and support the order of faults, the method of execution, and the ways results are stored. The simulators that consider both software programs and hardware descriptions are dealt with in Section 5. Section 6 highlights the open problems in the area of fault injection simulation tools, followed by conclusions in Section 7.

### 2 A new taxonomy for classifying fault simulation tools

We adopt a practical method to classify fault simulation tools, emphasizing the different development stages illustrated in Figure 1. We differentiate based on whether the target application is a software implementation  $(\mathbf{S})$ , a hardware circuit  $(\mathbf{H})$ , or a combination of both  $(\mathbf{C})$ . A soft-



Fig. 1. Overview of the design steps, moving from the target application to the final product.

ware application can be categorized as *target independent* when it is described using a high-level programming language (S1), as an intermediate representation (S2) when we compile the target application, or as an assembly program (S3). The application becomes *target dependent* when it is in the form of a binary file (S4) compiled for a specific target.

For a *hardware* implementation, the design is described in a hardware description language (HDL) such as VHDL or Verilog (test environments and behavioural models) (H1). This step is similar for both ASIC and FPGA design flows. The next development stage is the design synthesis, where the RTL code is translated into a gate-level netlist, which describes the circuit in terms of gates and connections (H2). The next step is the design implementation (or physical synthesis). For FPGA application development, the netlist is mapped onto a particular device's internal structure (place & route), and we get the layout (H3). In contrast, for ASIC implementations, netlists are transformed by the backend tools into the chip's layout (H3). Finally, the hardware design is implemented on the target component (H4)).

A combined target application ( $\mathbf{C}$ ) is typically a software application running on a hardware core, where the designer has access to the core description in one of the design stages described above.

When studying the fault simulation tools in the literature, we found that they share similar building blocks. In the following, we define the necessary concepts and then explain the flow in Figure 2, which depicts the anatomy of a typical fault simulation tool. We use this figure to reference the tools we found in the literature.



Fig. 2. Anatomy of a typical fault simulation tool.

**Definition 1 (Sensitive Region).** Given a target application A, the sensitive region  $S \in A$  is a subset of atomic elements whose modification might lead to a security breach.

For example, consider the target application in Listing 1.1, which shows a straightforward password authentication implementation. There are several options for choosing a sensitive region. One possibility could be the instruction on line 2, which could return a "1" if bypassed. Another possibility could be the instruction on line 13, which, if bypassed, could trigger the execution of the instruction on Line 15 and result in an unauthorized authentication. A non-expert user could also define the entire **verify\_password** function as a sensitive region. The sensitive region is a set of hardware elements when considering hardware implementations. For defining the concept of a fault model, we extend the definition of [65] to cover software, hardware, and combined target applications.

Listing 1.1. Naive password authentication implementation.

```
int verify_password(int psw_value) {
1
       if (psw_value == 1234) {
2
           return 1;
3
       } else {
4
           return 0;
5
       }
6
   }
7
   void main() {
8
       int pass, x = 3;
9
       while (x != 0) {
           printf("\nInput the password: ");
11
           scanf("%d", &pass);
12
           if (verify_password(pass)) {
13
14
               printf("Correct password");
               x = 0;
```

**Definition 2 (Fault model).** Given a target application A, a fault model is a set where each element is a triplet (s, t, e). Here,  $s \in S$  represents an element of the sensitive region, and  $t \in T$  is a natural parametrization of the sensitive region, so the pair (s, t) uniquely identifies the fault injection objective. The fault effect  $e \in E$ , represents a transformation applied to the target application A.

Considering the example in Listing 1.1, s can be an *instruction* or an *operand*, t could be the instruction number in the sequence of executed instructions, and e, the transformation, could be either an instruction replacement or a bit flip (or any other transformation). For hardware implementations, s can be a register or a logic gate, t the number of clock cycles, and e, the transformation, could be any data, signal, or circuit behaviour modification.

**Definition 3 (Mutation).** Given a target application A and a fault model (S, T, E), a mutation  $\mathcal{M}\{(s,t,e),A\} \rightarrow A'$  is a surjective transformation of the target application A into a new application A' where  $(s,t,e) \in (S,T,E)$ .

For example, replacing or removing the instruction on line 2 in Listing 1.1 would result in the function verify\_password returning the value 1, which signifies a positive authentication.

We observe that it is possible that two different fault instances (s, t, e) and (s', t', e') could lead to the same mutation of a given target application A. Figure 3 depicts this event using two arrows pointing to the same outcome. When using a fault simulator, we can control the mutation function, and when a successful attack occurs, we know the mutation that causes it. In a fault injection attack involving hardware modifications of the final target, determining the mutation function is typically tricky.



Fig. 3. The mutation function

We note that in **S1-S4** and **H1-H2**, the user has complete control of the mutation effects, which can be either data or control flow changes. However, considering **H3-H4**, the user partially controls the mutation effects due to glitches and circuit timing violations.

**Definition 4 (Mutation Order).** Given a target application A and a fault model (S, T, E), the mutation order is the number of fault model instances applied simultaneously to the application A.

For example, a mutation of order 2, will apply two fault instances  $(s_1, t, e)$  and  $(s_2, t, e)$  to produce mutation A'.

**Definition 5 (Execution).** Given a target application A and stimuli (input) I, the execution function,  $\mathcal{E}(A(I)) = O$  can produce a functionally correct output O.

Different strategies can be used to execute a mutated application: execute the whole application for every mutation or use saved states to reuse part of the execution. In definition 5, the only requirement we set for the execution is to produce a correct output O for the target application when input I is given.

**Definition 6 (Record).** Given a mutated target application A' and stimuli (input) I, the record function will document the output of the  $\mathcal{E}(A'(I))$  as one of three cases: 1) no fault  $\mathcal{E}(A'(I)) = \mathcal{E}(A(I))$ ; 2) abort  $\mathcal{E}(A'(I)) = \{\}$ ; 3) fault  $\mathcal{E}(A'(I)) \neq \mathcal{E}(A(I))$ .

**Definition 7 (Fault simulation campaign).** Given a target application A and a fault model (S, T, E), a fault simulation campaign will generate, execute, and record output for all mutations of the target application.

The input to the fault simulation tool depicted in Figure 2 is a target application A. According to a predefined fault model, the fault simulation tool will *mutate* the target application A and produce a mutated target application denoted by A'. The target application is *executed*, for a stimuli (input data) I. The result of the execution *records* the effects produced by the mutation.

The next section outlines the design and application of the fault simulators we encountered in the literature. Next to the core components defined above, we also document the following:

- Use Cases: Specifies the applications, platforms, or environments where the tool has been demonstrated to work as described in the paper.
- Dependencies: Lists the dependencies required for the tool to function, including libraries, frameworks, or other software components.
- Open source: Indicates whether the tool is open source (Yes/No). If yes, we provide a link to the repository.

### 3 Fault simulation for software targets

Understandably, no tools accept high-level source code as input. When high-level code is executed, it gets translated into machine code. This process breaks down each high-level instruction into multiple lower-level instructions. Therefore, performing fault attacks on high-level instructions would affect all the corresponding lower-level instructions.

#### 3.1 Tools for software targets (S2)

Lazart [45] aims to evaluate the robustness of code against multiple, volatile faults targeting the modification of the execution flow. The target application is represented in LLVM intermediate form. Lazart takes, as input, the program Control Flow Graph (CFG) and the *sensitive region* 

of the target (determined manually). To generate *mutations*, Lazart implements a block colouring algorithm and uses mutation operators (optional or mandatory transformation) to create mutation patterns. As part of the *fault model*, the transformation is *test inversion*, which involves flipping the outcome of a conditional test. Given the sensitive region of the target application, the block colouring algorithm will identify and label sensitive blocks. These are labelled *to be reached* - corresponding to instructions which lead to gaining of privileges, or *not to be reached* - corresponding to countermeasures. Mutations are generated by modifying the LLVM bytecodes.

With the attack objective defined and the coloured graph, the tool will try all mutations and record the combinations that lead to success (or the lack of successful attacks). For the *execution* of mutants, Lazart uses KLEE [12], the symbolic test generator. Lazart is tested with a PIN verification algorithm and a cryptographic detector for SSH packets.

**Takeaway:** Graph colouring is a clever method for selecting sensitive regions. The combination of graph colouring with a symbolic execution tool generation such as KLEE can determine the absence or existence of attacks for a given set of inputs.

#### **3.2** Tools for software targets (S3)

**Formal Verif** [39], presents a countermeasure scheme to protect embedded programs from instruction skip fault attacks. This scheme is specifically tailored for the ARM Thumb-2 instruction set, providing fault-tolerant replacement sequences for nearly all instructions. The proposed countermeasure creates these sequences by duplicating or transforming instructions to ensure that even if a fault occurs, the correct execution path can be maintained. Examples include *idempotent instruction* sequences (instructions that have the same effect when executed once or multiple times), *separable instruction sequences* (instructions which can be rewritten as a sequence of idempotent instructions), and *specific instruction sequences* (instructions which cannot be replaced by idempotent instructions) which are duplicated to enforce robust execution of a program. In terms of execution, instead of proving that a complete program is fault-tolerant, the model will only focus on the replacement sequences, checking the code's functional equivalence and resilience to fault attacks.

The presented case studies focus on the results of implementing the countermeasure scheme. The paper presents data on the overhead costs and effectiveness of the scheme for two different AES implementations and a SHA implementation. It concludes that while there is a performance cost, the security benefits justify the trade-off. The discussion acknowledges that while the proposed scheme significantly enhances fault tolerance, it does not offer protection against all fault attacks. The approach is effective against single instruction skips but may not cover more sophisticated multi-fault scenarios. The authors also suggest that the overhead can be managed by focusing on the most critical parts of the code.

**Takeaway:** The authors of FormalVerif accept that all programs are vulnerable to fault attacks, propose to forego testing and add countermeasures for the instructions in the sensitive region.

#### **3.3** Tools for software targets (S4)

**FIES** [27,28] is a framework originally developed for safety applications and extended to address security. The framework is based on the  $QEMU^3$  emulator. The *mutation function* is applied inside

<sup>&</sup>lt;sup>3</sup> https://www.qemu.org/

the emulator before the QEMU binary translation operation. Indeed, the application is profiled to have memory/register usage. Then, a fault library is built according to fault scenarios and fault models. Finally, this library is applied to the target application to provide a *mutated* application, which will be executed by QEMU. The tool supports different *fault models* (stuck-at-x, bit flip, given value) and *sensitive regions* such as the CPU, the memory, and the registers. The framework has been tested for access control applications, control flow errors, and memory-related attacks. **Takeaway:** The tool allows developers to benefit from the different targets (COTS processors) supported by QEMU. It uses an extensible markup language (XML) to describe the fault model and the parameters for the sensitive regions, which improves the automation of the fault simulation campaign.

**FiSim**, based on the Unicorn emulator<sup>4</sup> and the Capstone disassembler<sup>5</sup> is an open-source<sup>6</sup> deterministic fault attack simulator supporting ARM32/ARM64 architectures. It enforces two transient fault models, namely a NOP instruction model and a single-bit flip instruction model to execute the injection first-order faults to the sensitive region, i.e., the entire binary. FiSim is preconfigured with an use-case comprising of an implementation of a secure bootloader. Before mutating the implementation to run the fault simulation campaign, it compiles to generate an ARM binary and proceeds to perform a golden run (without injecting faults). Subsequently, to perform mutation, it injects faults exhaustively throughout the binary and compares its execution trace with that of the golden run. For a bootloader, the two traces would differ due to the difference in their signatures, which prompts FiSim to decide whether the authentication can be deemed successful. If the binary is vulnerable, leading to successful authentication. FiSim compares the result of the mutated binary with the golden run to report whether an injected fault has been successful. It records a list of assembly instructions where the faults have been successfully injected and redirects it back to locations in the high-level program.

**Takeaway:** FiSim could be useful to developers if one intends to check for fault resilience in ARM32/64 implementations, following the program structure of a secure bootloader while being reasonably small in size and cares about first-order faults. Since its mutation function analyzes first-order fault simulation for every instruction with both the fault models serially and exhaustively, its runtime accelerates at an alarming rate for larger real-world applications. Due to the input to the simulator being hard-coded, it poses difficulty loading other implementations.

**ARMORY** [26], *built upon* M-ulator<sup>7</sup> and requiring the Meson build system<sup>8</sup> and the ARM GCC toolchain [15], is an *open-source*<sup>9</sup> instruction-accurate fault emulator for ARMv6-M, ARMv7-M, and ARMv7-EM instruction set architectures. Using M-ulator has its advantages since it can handle faulty assembly instructions and has been shown to outperform the Unicorn emulator [26]. ARMORY can inject faults of twenty-four *fault models* encompassing instruction-level (permanent, transient) and register-level (permanent, transient, active until overwrite) faults. ARMORY, akin to FiSim, explores the fault space (i.e., all possible faults) exhaustively, however, it diverges from

<sup>&</sup>lt;sup>4</sup> https://www.unicorn-engine.org/

<sup>&</sup>lt;sup>5</sup> https://www.capstone-engine.org/

<sup>&</sup>lt;sup>6</sup> https://github.com/Riscure/FiSim

 $<sup>^{7}\</sup> https://github.com/emsec/arm-fault-simulator/tree/master/subprojects/m-ulator/subprojects/subp$ 

<sup>&</sup>lt;sup>8</sup> https://mesonbuild.com/

<sup>&</sup>lt;sup>9</sup> https://github.com/emsec/arm-fault-simulator

FiSim, by providing support for parallel execution of the fault emulation by utilizing all the available CPU cores. ARMORY takes as input an M-ulator instance consisting of a high-level ARM-M code along with start and halt symbols to mark the *sensitive region*, a set of fault models to be injected, an exploitability model, and halting points, i.e., a set of addresses at which exploitability is evaluated.

ARMORY starts its *execution* with a golden run to keep track of the exploitable locations in order as well as the emulation time required. ARMORY *comes embedded* with DFA (differential fault attack) on AES and a secure bootloader as case studies. For the *fault simulation campaign*, M-ulator is run until the first injection point to save its state followed by which a fault model is applied. Via the golden run, M-ulator retrieves the sequence of executed instructions and the registers used, providing all the injection points. Once encountered, ARMORY adds faulty locations to a list of exploitable locations and starts injecting faults to *mutate* the saved binary version backed up in M-ulator, thus saving emulation time. If not found exploitable, the next fault model is applied to the backed-up state stored in M-ulator. When applying a fault model, a halting point is reached, ARMORY verifies against the exploitability model to determine whether the fault has been successful to be added to the list.

It *furnishes* a detailed view with the number of faults (injected/successful), the fault model, the elapsed time, position, and timestamp, along with the number of threads used and the faulted instruction in the assembly code depicting the affected instruction, register, bit or byte.

**Takeaway:** ARMORY supports *multivariate* fault simulation. The comprehensive report in a log file also helps in future referencing. ARMORY illustrates that checking for resilience against faults on high-level implementation might not guarantee the same implementation being secure against faults in low-level representations. It also emphasizes that a countermeasure against a fault of a certain model could increase its vulnerability against other fault models. Due to its exhaustive nature of injecting faults, it has a complexity of  $\mathcal{O}(n^m)$ , and for multivariate faults,  $\mathcal{O}(n^m.m!)$ , where n is the number of injection points and m is the number of fault models.

**ARCHIE** [24], an open-source<sup>10</sup> fault simulator, dependent on QEMU to support numerous architectures, approaches simulating fault injections from a different angle as compared to other tools in this section. Unlike the other tools, which explore the fault space, ARCHIE requires the developer to provide it with the fault configuration, which defines the *sensitive region*. It verifies if an injected fault results in a different output than the expected output. It takes a compiled binary as well as a QEMU configuration and a user-defined fault configuration, which essentially could support higher orders of mutation. Essentially, ARCHIE harbours the capability to support all the architectures supported by QEMU. While the QEMU configuration defines the binary's location and the DuT's architecture, the fault configuration describes the fault campaign. It supports four *fault models*, both permanent and transient ones, with user-defined parameters for individual test cases. It functions via a controller script that takes the QEMU configuration, the fault configuration, and the compiled binary to launch multiple worker tasks, each executed in parallel to run the fault simu*lation campaign.* Each of these parallel processes starts a QEMU instance and provides the fault configuration and the location of the binary to ensure that each worker task handles a single fault model from the fault configuration to perform the *mutation*. After completing the simulation, these worker tasks are collected, processed, and forwarded to the logger, which saves it in an HDF5 file. As use-cases, ARCHIE runs fault simulations on TinyAES and a secure bootloader implementation. **Takeaway:** ARCHIE needs the user to provide the fault locations instead of exploring the fault

<sup>&</sup>lt;sup>10</sup> https://github.com/Fraunhofer-AISEC/archie

space itself. However, it provides a detailed output on the console, which could be handy for debugging if needed. Though it facilitates parallel execution, its high memory usage [1] could be concerning.

**SAMVA** [21], a static analysis-based framework, *built upon* the angr framework<sup>11</sup>, assesses ARM binaries' resilience to multi-fault attacks. SAMVA attempts to tackle the problem of fault space exploration. Instead of executing fault campaigns, SAMVA establishes a way to accelerate determining fault locations by discovering attack paths, which would alleviate the fault space exploration problem. It focuses on identifying attack paths in a binary that could be affected by the multiple instruction-skip *fault model* of varying widths, implying multivariate *mutation order*. It also quantifies the vulnerability of a binary based on the minimum number of faults to deem them exploitable.

SAMVA takes an ARM binary, the exploit specifications, defining the possible number of faults, and the attacker's goals, describing *targeted basic blocks*, that must be executed, thus acting as the *sensitive region*, and *forbidden basic blocks*, that must not be executed, as its inputs to *obtain* a set of N attack paths satisfying the user-specified fault injection constraints. These attack paths take the form of a list of basic blocks, including the faults to inject, depicting the fault's location and width.

SAMVA initiates the *fault simulation campaign* by generating control-flow graphs (CFGs) of the binary, followed by extending and annotating the CFGs to illustrate the effects of possible faults induced to form an extended CFG (ECFG). To *execute* faults on the targetted basic blocks, SAMVA delves into the ECFGs to generate attack paths that comply with the attacker's goals. Once the ECFGs depict the attack paths, SAMVA builds an execution trace for each candidate path and determines whether a fault configuration can be induced for an execution trace per the exploit specifications by applying *mutation*. Finally, it uses a backtracking algorithm to explore all the possible fault configurations to single out a valid configuration for the candidate paths. It also performs fault trimming by reducing the widths to prevent skipping critical instructions. SAMVA has been shown to *work upon* eight PIN verification programs from the FISSC [16] suite.

**Takeaway:** SAMVA explores attack paths instead of executing exhaustive fault campaigns, which can be a significant step towards solving the fault space exploration problem. SAMVA supports the (multiple) instruction-skip fault model with varying widths. It takes a unique approach by generating attack paths while remaining customizable by letting users define the exploit specifications and the attacker's goals.

**CELTIC** (CEsti-LeTi Integrated Circuit) [17,69], a simulation-based fault injection tool, *structured around* a micro-architectural simulator, takes the target binary code, the execution trace, the set of fault models, the attack order and an oracle as inputs. The oracle is a boolean condition to classify attacks as successful. CELTIC has been shown to work on ARM Cortex-M4 32-bit micro-controller, the results of which were verified using laser fault injection (LFI).

To perform the *fault simulation campaign*, CELTIC generates target-specific *fault models* (TSFMs) from ISA-level fault models, which determines the *mutation order* and also deals with combined fault attacks. The TSFMs are inferred by parallelly simulating fault injections and running characterizations via a grid search on the target. Not all TSFMs are applied. However, only those whose probability of occurrence is above an arbitrary threshold are considered. The lower the threshold, the more exhaustive the fault analysis, but the longer it takes to simulate, the greater the possibility

<sup>&</sup>lt;sup>11</sup> https://angr.io/

of generating more false positives. TSFMs attempt to solve the combinatorial explosion problem of multi-fault and combined-fault attacks.

CELTIC first *executes* a golden run of the *sensitive region*, i.e., executes the reference trace without injecting any fault. To perform a mutation, for each saved state of the execution trace and each fault model, it generates all possible faults and injects each of those faults to produce a faulty execution trace and *reports* it if it could be deemed as successful as per the oracle or else continues with the fault injection process. The injection delays of the successful attacks are converted to microseconds via a linear relationship, and an error margin is added to configure the equipment for successful fault attacks. CELTIC demonstrates its capabilities on an *use-case* consisting of VerifyPIN on an embedded device. It supports parallel execution for characterization and fault injection simulation.

**Takeaway:** CELTIC takes a unique approach to address the fault space exploration problem by executing fault characterization and fault injection simulations parallelly to distinguish fault-prone points. However, the effectiveness of TSFMs could be hindered because it could be challenging to generate them. The fault injection simulation, being entirely dependent upon the TSFMs, the accuracy of fault injections would be as good as the TSFMs. The conversion from simulation settings to configuration for the hardware setup may not reflect practical setups and may not result in successful fault exploitation. Since CELTIC does not operate exhaustively, it would likely run to its completion in most scenarios and only consider TSFMs instead of all fault models, thus advancing towards recognizing the likeliness of different fault models.

### 4 Fault simulation for hardware targets

#### 4.1 Tools for hardware targets (H1)

**FISSA**<sup>12</sup> (Fault Injection Simulation for Security Assessment) [43] is an open-source tool to analyze the sensitivity of the circuit to fault injections. It consists of three modules: TCL generator, Fault Injection Simulator, and Analyzer. The TCL generator relies on a configuration file and a target file defining the considered fault model(s) to generate a set of parameterized TCL scripts. These scripts drive the fault injection simulation campaign and apply mutations to the target hardware elements when required during the simulation process. The Fault Injection Simulator performs the fault injection simulation campaign based on input files from the TCL generator for a circuit design described through HDL and memory initialization files. For that purpose, it relies on QuestaSim HDL simulator<sup>13</sup>. Finally, the Analyser module records the outcomes of the simulations to generates a set of report files allowing the designers to examine fault injection effects on their designs.

FISSA can inject faults into hardware registers with a maximum mutation order of two. To configure the fault model, FISSA provides a list of fault effects such as set to 0/1, single and multiple bit-flips. The sensitive region is defined through a list of target hardware registers. Finally, an injection window parameter identifying time intervals into which fault injections are performed completes the fault model configuration. The FISSA user is responsible for providing JSON-formatted input files that include these parameters.

**Takeaways:** FISSA can be used in the first development stage of hardware designs. Since it relies on a well-known HDL simulator, it is easy for a hardware designer to handle. However, the user must manually define the list of target registers and the sensitive region. Furthermore, even though

 $<sup>^{12}\</sup> https://github.com/WilliamPsc/FISSA/tree/main$ 

 $<sup>^{13}\</sup> https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/$ 

the tool automates the generation of fault injection results from statistics and identifying sensitive registers, the user must provide the tool with information to classify the execution outcomes (i.e., faults).

#### 4.2 Tools for hardware targets (H2)

**SoFI** [68] is an automated framework that takes as input a gate-level design with a set of security properties and a set of attributes describing the faults considered for the fault injection campaign. These attributes are time, location, type, and duration. This set of attributes is used to derive the *fault model* considered during the campaign.

The security properties are used to guide the vulnerability assessment: if a fault does not violate any security property of the design, its effect is assumed to be limited. To simulate the effect of a fault on a circuit, it uses a logical model of the faults' physical impact. By using this model, it simulates the faults' propagation into a digital circuit.

After receiving these inputs, the framework generates a fault model by considering the set of security properties and the specified fault attributes. This framework assumes the attacker can inject a fault at any specific clock cycle. Regarding the *sensitive region*, faults are injected at the sub-circuits fan-in cells that check a security property. If the framework is used to consider a global fault, then it only considers the output of sequential cells in modelling fault locations to reduce the workload's complexity. The *mutation order* allows to consider up to two concurrent sensitive regions. After generating the fault list, SoFI uses the Z01X simulator<sup>14</sup> to simulate the fault's effects on the circuit under analysis. Then, this framework has an optional fault feasibility analysis step.

For faults that aim to violate the setup time of flip-flops, SoFI checks the gate-level simulation traces to see if the value at the flip-flop's input at the time of fault injection is different than the previous clock cycle. If there is a setup time violation, the flip flop will latch the value of the previous clock cycle. So, this kind of fault injection will effectively cause a bit flip only if there is a state transition in the flip-flop's input.

**Takeaways:** SoFI uses a set of security properties to guide the fault injection campaign: only faults that can break a security property are considered relevant. SoFI cannot evaluate whether a fault is feasible based on local fault injection techniques. It operates on a circuit's gate-level design, and evaluating such fault requires layout information about the design. However, SoFI's results can still inform later design stages to prevent such vulnerabilities. To avoid false negatives, a powerful attacker is considered, but this comes at the cost of generating false positives.

**SYNFI**<sup>15</sup> [40] is a formal fault verification framework operating on synthesized netlists. It tests the effects of faults on the input-output relationship of circuits by taking as inputs the circuit's gate-level netlist, the standard cell library used for synthesis, and a fault specification.

The fault specification is split into two sections: a description of the sub-circuit to be evaluated and a fault model. The sub-circuit is determined by specifying its input and output nodes, values for the input nodes, and expected output values (what the circuit should produce under normal functioning). The *fault model* is described by specifying the *mutation order* and the nodes where to inject the faults, which constitute the *sensitive region*, as well as a fault mapping describing the fault's effect on the gate: the gate will behave as the gate or boolean expression specified in the mapping.

<sup>&</sup>lt;sup>14</sup> https://www.synopsys.com/verification/simulation/vc-z01x.html

<sup>&</sup>lt;sup>15</sup> https://github.com/lowRISC/synfi

First, SYNFI translates the netlist into a directed multi-graph and the standard cell library into a format that the rest of SYNFI supports. Then, it extracts from the directed multi-graph, the target graph representing the sub-circuit to analyze. Successively, a new process for each fault mapping and location is started. Each process uses two target graphs: a non-faulty and a faulty one. The non-faulty target graph represents the original sub-circuit, whereas the faulty one models one or more faults by substituting the boolean expressions of the involved gates as specified in the fault mapping. These two copies are combined into a differential graph by adding an input and an output layer. The differential graph is used to see if a fault was effective, i.e., if it changed the output of the faulty graph compared to the non-faulty one, without triggering an error signal. To check for this, the differential graph is handed out to an SAT solver that evaluates the fault's effectiveness. In the end, SYNFI provides a report summarizing the fault analysis results.

**Takeaways:** SYNFI is a versatile tool that lets the user specify any possible type of fault in a gatelevel circuit design. It supports parallel execution. SYNFI guarantees no false negatives as long as the fault specification reflects the intended threat model. Its main shortcoming is an absence of evaluation of a fault's feasibility: it is left entirely to the user to specify in the fault mapping logical expressions corresponding to faults that can be injected in practice.

#### 4.3 Tools for hardware targets (H3)

**SimpliFI** framework [22] relies on post-layout netlist simulations to study the impact of fault injection techniques such as clock glitches. Indeed, timing-based faults (e.g. setup time violation) can be studied using a post-layout netlist since it provides device-specific signal propagation delays. It is worth noting that the proposed approach specifically targets the evaluation of embedded processors executing sensitive software. To capture the impact of faults, SimpliFI proposes to record, during the simulation, hardware-level signals to correlate faulty software-level outcomes with corrupted hardware elements. At the end of each fault simulation, software-level outcomes are determined by collecting the processor state, which includes general registers, program counter, and final processor hardware state.

To automate a fault injection campaign, the user provides a fault configuration file, defining the *sensitive region* as a list of target instructions and the set of mutations to be applied to the clock signals. For that purpose, a custom and flexible file format is specified. Together with the compiled program, it generates a script driving a hardware simulation tool in charge of executing the programmed netlist for each test case. SimpliFI analyzes the impact of faults at hardware and software levels by leveraging the hardware state information recorded during simulations. It computes the hamming distance between the outputs of the current fault simulation and the outputs obtained from a reference simulation. This allows the identification of corrupted registers for a given injection event and enables the user to link it to the software-level outcome determined by analyzing the final processor state.

**Takeaways:** The SimpliFI framework is dedicated to the evaluation of embedded processors. It focuses on fault injection simulation due to timing violations. The user manually defines the *sensitive region* through a set of memory addresses indicating the tool to which instructions must be targeted. It is worth noting that the user does not control the *mutation order* or provide a complete *fault model* as defined by definition 2 since the tool relies on timed simulation to evaluate the impact of clock glitching. However, the user must provide the parameters determining the clock glitch shape and location.

### 4.4 Tools for hardware targets (H4)

**Chiffre**<sup>16</sup> [18] is a framework to instrument a hardware design with fault injection capabilities. A Chisel<sup>17</sup> library provides hardware fault injector examples and allows the designer to annotate the circuit description to identify a *sensitive region* (i.e. registers or wires). Chiffre relies on both Chisel and FIRRTL<sup>18</sup> (Flexible Intermediate Representation for RTL) compilers to generate instrumented hardware descriptions in Verilog and target circuit emulation on FPGA. Notably, it provides dedicated FIRRTL passes to extend a circuit with run-time programmable fault injectors responsible for applying mutations to annotated circuit elements. Fault injectors are configured through a scan chain bit-stream generated parallel to the circuit design. Supported *fault models* are implemented into fault injectors. Chiffre supports the following fault effects: pseudorandom bit flips, stuck-at-zero/one in words, and bit-flips at a specific time location. It is worth noting that the last one is the most relevant for a security analysis since time location is configurable. The *mutation order* is theoretically unlimited. However, in practice, it is limited by the maximum number of fault injectors that can be implemented on the target FPGA.

**Takeaways:** Chiffre automates circuit extension with hardware fault injectors to perform fault injection emulation on FPGA. The user must manually define the circuit's sensitive regions where injectors must be implemented. The choice of the fault model and the outcomes analysis is also the user's responsibility.

### 5 Hybrid and combined tools

**µarchiFI**<sup>19</sup> [65] provides a solution for the formal analysis of mixed hardware/software systems under fault injection. The proposed approach allows the analysis of the impact of faults, modelled at the microarchitectural level, on the software outcomes. µarchiFI is based on Yosys synthesis suite<sup>20</sup> and a third-party model checker. µarchiFI extends Yosys to generate the system's formal model from the RTL implementation of the hardware, an input binary program, and the attacker model. The circuit (i.e. processor design) is modelled as a transition system as described in [5] and the software program is encoded in the initial state of a memory modelled simultaneously with the processor. Then, a model checker is used to analyze the system's robustness under fault injection.

In parchiFI, faults are injected through the mutation of the original transition system. The considered *fault model* is configured by selecting a fault effect among the following: set, reset, simple, and multiple-bit flips. The *sensitive region* is defined as a list of hardware registers. Furthermore, the *mutation order* is restricted through a user-defined parameter. It limits the number of faulty transitions an attacker can introduce. The attacker goal is a reachability property defined on the transition system. It represents a vulnerability that the attacker wants to reach. Finally, bounded verification techniques determine if a sequence of states allowing an attacker to reach its goal exists considering a limited number of faulty transitions. If such a sequence exists, the tool outputs a VCD (Value Change Dump) file that reports where the fault is injected and when the attacker's goal is reached. Human expertise is necessary to analyze the propagation faults and their software outcomes.

 $<sup>^{16}</sup>$  https://github.com/IBM/chiffre

<sup>&</sup>lt;sup>17</sup> https://www.chisel-lang.org/

<sup>&</sup>lt;sup>18</sup> https://github.com/chipsalliance/firrtl

<sup>&</sup>lt;sup>19</sup> https://zenodo.org/records/7958412

<sup>&</sup>lt;sup>20</sup> https://github.com/YosysHQ/yosys

**Takeaways:** parchiFI provides an elegant solution for pre-silicon evaluation against FIA (fault injection attacks). It allows us to formally analyze and study the propagation of faults (thanks to VCD file generation) in the microarchitecture and their consequences on the system behaviour. However, the user has to determine the attacker's goal manually. Furthermore, the user needs to find a trade-off between the size of the hardware design, the size of the analyzed program, and the complexity of the fault model.

**EFS** [48] proposes to evaluate the robustness of smart card source codes by combining software instrumentation and on-target execution. The proposed approach relies on the high-priority process property of programmable interrupts to mutate the manipulated data and the control flow. The *sensitive region* is defined as a target function extended with unique ID tags and extra code to configure a timer that triggers a programmable interruption after a specific delay, which defines the fault injection time. The interruption code is personalized to apply a set of fault effects associated with the target *fault model*. Faults are injected by modifying general and special registers, such as the program counter and stack pointer, or tampering with the interrupted process memory. This allows the user to determine the *mutation order* freely and to implement numerous fault effects such as simple or multiple instructions skip and data/code modifications at bit, byte, or word level. Furthermore, a parameter simulates transient and permanent fault effects (i.e., the fault injection occurs at each target function call).

**Takeaways:** EFS is dedicated to the application domain of smart cards. The main advantage of the approach is to consider the final hardware target for fault simulation. However, fault injections are limited to hardware elements accessible from the software stack. Thus, the impact of faults on hidden microarchitectural structures is not considered. Furthermore, since the approach relies on hardware timers, such a resource must be available to apply the proposed approach. The user must manually define the sensitive region to be evaluated and modify the operating system to manage the fault injection-related interrupts.

Fault-Resistant Partitioning<sup>21</sup> [66] presents a combined approach where, after a first evaluation of the hardware design, fault injection vulnerabilities are then considered to evaluate the resilience of a specific software executable. The authors' main contribution lies in how they verify the hardware design: they introduce the concept of k-fault-resistant partitioning, which formally proves the security of the given circuit. This framework's *fault model* considers a list of logic gates that can be faulted with transient faults where a bit can be set to 0/1 or be flipped. It supports an unlimited *mutation order*. The hardware verification step can be done at the RTL or netlist level. They define a circuit as k-secure if, when injecting up to k-faults, it is impossible to make the circuit misbehave without it being detected.

First, the circuit design is converted to a cycle-and-bit-accurate model. Then, it derives a fault model  $\mathcal{F}$  from the input fault model and the produced circuit model. The circuit's k-security property is analyzed using an inductive approach. This is done by checking an invariant that implies the k-security property. This invariant holds under two conditions: circuit outputs are correct under any k-fault injections that do not raise an alert, the sequential circuit elements can be partitioned such that any k-fault injections are either detected or confined in partitions.

Verifying the k-fault-resistant partitioning starts by iteratively building a partitioning that ensures fault confinement or detection. Once a suitable partitioning is built, it iteratively verifies that this partitioning also ensures the output's integrity. If this verification fails, the faults that lead

<sup>&</sup>lt;sup>21</sup> https://github.com/CEA-LIST/Fault-Resistant-Partitioning

to the outputs' corruption are added to a set  $\mathcal{F}'$ , denoted as a set of exploitable faults, and the partitions whose corruption alters the outputs' integrity are added to the set  $\mathcal{P}'$ , denoted as a set of exploitable partitions. The verification eventually succeeds and outputs the sets  $\mathcal{F}'$  and  $\mathcal{P}'$ . This hardware verification step is independent of the executed software, so it can be executed only once and then used to verify multiple pieces of software. If no exploitable faults are identified, there's no need to perform the software verification step.

Next, it analyzes program executions to verify if an attacker can reach its goal. This phase only considers the faults that the previous step could not prove are detected by hardware countermeasures. This software and hardware co-verification takes the hardware design, a binary program, the mutation order, the attacker goal, and the sets  $\mathcal{F}'$  and  $\mathcal{P}'$  as input. The system modelling process combines all these elements into a unified model mapping the software execution onto the hardware, whose behaviour gets modified by the faults in  $\mathcal{F}'$  and  $\mathcal{P}'$ . The potential fault locations derived from the two sets are used to select the best-suited abstraction level during the modelling step. The verification step tells whether the system is robust against the considered attacker and produces counterexamples in the form of VCD files if any vulnerability is found.

**Takeaway:** The notions of k-security and k-fault-resistant partitioning lead to formal proof of the resilience of a CPU design against faults. Additionally, this tool can verify if the faults not detected by hardware countermeasures can result in faults for a specific software executed by that CPU. So, this tool lets a design team analyze trade-offs between hardening a design at the hardware, software, or a mix of the two. Thus, it is a tool that can paint a complete picture of a device's security against faults and can be used in different design stages. The main limitation is that the existence of a k-resistant partitioning is a sufficient but not a necessary condition for the k-security of a circuit. Building a k-secure partitioning relies on a heuristic. So, while successfully constructing a k-secure partitioning proves k-security, failure to obtain such partitioning does not imply the existence of vulnerabilities.

### 6 Discussion of the current state of the art

The landscape of fault injection testing has evolved significantly, shifting from traditional physical fault injection methods to modern simulator-based approaches. This transition brings numerous benefits, including greater accessibility for non-experts and enhanced cost-effectiveness. Fault simulators make fault injection testing feasible for a broader range of developers. Our findings highlight the significant progress in this domain and underscore unresolved challenges.

Achievements. As we see throughout the paper, different fault simulators adopt diverse approaches toward making an application fault-resilient. Some fault simulators incorporate formal verification; for instance, FIES [28] reports the presence or absence of possible fault attacks for a selected portion of an implementation, thus serving as a guarantee for the security of an application. On the other hand, fault simulators based upon experimental evaluation tend to cover a larger code base, which may lack in providing a guarantee but attempt to inject comparatively more faults. Fault injection simulators also cover the ARM architecture reasonably well. Tools like FiSim [47] make it convenient for the user by providing an environment that takes the target implementation in a high-level language and cross-compiles it to make it ready for fault simulation. Others like ARMORY [26] support numerous fault models and take an innovative approach to reduce emulation time by saving states during fault simulation. ARMORY, SAMVA [21], and CELTIC [69] allow a user to specify an exploitability model. For experienced users, ARCHIE [24] offers the flexibility to use various

architectures and targets user-defined locations for fault simulation, while for targeting hardware designs, FISSA lets the user define the sensitive region and target registers thus offering more control on the setup. SAMVA and CELTIC take on the problem of fault space exploration using two contrasting techniques. While SAMVA focuses on identifying attack paths based on user-defined exploit specifications and the attacker's goals, CELTIC performs fault injection on real hardware targets to obtain target-specific fault models (TSFMs). Introducing TSFMs could help identify the most likely fault models. The various fault simulators aimed at hardware designs, like SoFI and SYNFI, help make informed choices for the later stages. Tools like SimpliFI do not require the user to define the mutation order or the fault model since it relies on timing-based faults, thus exploring fault space in another way.

The state-space problem refers to processing the possible mutations generated by fault instances up to the given mutation order. The number of combinations may lead to a state-space explosion that quickly renders the problem of fault injection intractable. In our view, the complexity of the state space depends on three parameters:

- 1. Selection of the sensitive region determines the target locations where faults are injected. The goal is to reduce the size of the sensitive region to reduce the state space. This delicate task requires experience with fault injection attacks. The two (extreme) options are *manual* selection and *automated* selection. Automated selection is attractive to novice audiences. However, manual or hybrid selection of the sensitive region has advantages. For instance, FiSim supports the automatic selection of the sensitive region by considering the entire implementation, which can be helpful, though it increases runtime. ARMORY [26] hybrid selection gives more control to the user and ensures that libraries or external/referred code-bases are not the targets for faults injection. ARCHIE [24] manual selection is appealing when specific instructions are targeted. While automatic selection could be beneficial during the early stages of the development cycle, hybrid and manual selection could be beneficial during the later design stages. Tools for hardware targets like FISSA [43], SimpliFI [22], Chiffre [18], SYNFI [40] and EFS [48] use manual selection because for hardware circuits that help to restrict the sensitive region.
- 2. The choice of fault model is a pain point for many fault injection vulnerability assessment tools. We see two approaches. The first is *target informed*, where the hardware platform determines the tested fault effects. The second is to consider a library of known fault models. It is expected that the more precision the fault requires, the less likely it will manifest itself. This is strictly connected with the chosen threat model. Most works either do not perform a feasibility evaluation step or err on caution by considering a potent attacker in the threat model. CELTIC [69] approaches this problem of determining the feasibility of fault models by generating target-specific fault models (TSFMs). These TSFMs are obtained by simultaneously executing fault injections and fault characterizations using grid search on the target. Then, depending on a threshold, selected TSFMs are considered further in the experiments. This makes the success of the fault simulations entirely dependent on the generated TSFMs and would only be as reliable. Delving into fault characterization for targets could be beneficial in learning how probable a fault model is for that target and reducing the state space.

For instance, one of the fault models supported by most of the fault simulators, i.e., the single-bit flip fault model, fails to capture effects caused due to voltage glitching (VCC), electromagnetic fault injections (EMFI), body bias injection (BBI) [25] or laser fault injection (LFI). To represent such fault effects, fault models must capture multiple bit flips, timing violations, spatial correlation and differentiation between transient and permanent faults. The other most fre-

quently used fault model, instruction skip falls short when it comes to representing corruption of the program counter (PC), stack pointer (SP) and link register (LR) [7,6,59,64].

3. The mutation order exponentially affects the number of fault instances. Specific simulators let the user consider an unlimited mutation order, like SYNFI [40], but the analysis is restricted to a sub-circuit of the original design. Another possibility is to employ formal methods to explore the search space efficiently. An example of this is Fault-Resistant Partitioning [66], which iteratively builds partitioning by gradually expanding an initial partition. Each time the partition is extended, the proposed tool applies an algorithm to prove an invariant that implies the resistance of the current partitioning up to a given number of faults. This iterative construction bounds the complexity of obtaining a secure partitioning, as one does not have to exhaustively check each possible partitioning to find the biggest one that still satisfies the invariant. Other frameworks put an upper limit on the mutation order or concurrent fault locations to ensure that the state space does not become too big to search. An example of this approach is SOFI [68]. Instead of exploring the fault space by executing fault campaigns, simulators like SAMVA [21] generate attack paths which significantly decrease the search space, depending on the user-defined exploit specifications and attacker's goals. Another approach to tackle it would be to characterize faults instead of injecting all possible fault combinations. CELTIC [69] does precisely so and attempts to explore the fault space by learning possible successful fault configurations using fault characterization. An exciting research direction is to consider further how to reduce the fault space exploration problem effectively.

User Guarantees. Some tools apply formal methods to the problem of determining a circuit's/software's vulnerability to fault injection. This approach gives the strongest guarantees about the accuracy of the tool's analysis. The most common guarantee is the absence of false negatives, as tools that only ensure the elimination of false positives would be inadequate for security applications. Unfortunately, this guarantee usually implies the presence of false positives. However, designers prefer to err on the side of caution, protecting against attacks that may not be feasible in practice rather than risking the possibility of missing an exploitable vulnerability. The best would be the absence of false positives and negatives, but this is not easy without a prototype to test. Part of the reason is that design stages S1 to S4 and H1 and H2 lack information about the hardware's physical layout. However, the layout is crucial in some fault models. In the case of laser fault injection, a single laser spot typically causes faults in multiple locations close to each other. But at the same time, the usual assumption is of an attacker with only one laser available [68], as laser setups are expensive. Suppose a vulnerability requires injecting faults into multiple locations. In that case, it may be feasible only if those locations are close to the final design's layout. With today's computational resources, we believe it would be possible to give formal guarantees about the absence of false negatives and false positives only by reducing the scope of the problem.

In the pre-silicon design phase, there are too many unknowns regarding the physical realization of a circuit. The situation can be different for software tools, assuming the developer knows which device will be used to run the code. To this aim, an area of research could be finding a set of reasonable assumptions about the physical aspects of a hardware design that make it easier to provide formal guarantees without hindering the validity of the analysis results in practice.

Other tools only give empirical guarantees about their results. To guarantee the absence of false negatives, the typical approach considers a worst-case attacker with capabilities that are hard to achieve in practice. However, this approach also implies the likelihood of false positives.

SoFI [68] considers an overly powerful attacker, thus guaranteeing the absence of false negatives in their evaluation at the expense of false positives. Other tools, like SYNFI [40], do not pick a threat model, but rely on the user to specify it. Whether that results in false positives (or false negatives) depends on whether the used threat model accurately models the reality. In other words, such tools can formally guarantee that an application is not subject to fault injection attacks, assuming the user provided a suitable description of the attacker's capabilities. This is a compromise between the guarantees given and the difficulty of delivering such guarantees: by having the user help in defining their threat model, the tool can provide formal guarantees using formal methods. As a bonus, such an approach results in tools that can be used to consider different threat models. In theory, a user could produce a threat model that formally guarantees the absence of false negatives and false positives. However, such a threat model would need to be formally proven by the user, making this an unrealistic approach to the problem. A more realistic approach would be to use a threat model derived from prior experiments to determine an attacker's capabilities.

**Reachability versus Exploitability.** Reachability is the condition that determines whether an injected fault is effective, whereas exploitability is the condition that determines whether an attack is possible. Tools focused on reachability could be used as fault injection simulators, whereas those focused on exploitability could be deemed fault attack simulators. This is because successfully injecting one or multiple faults does not necessarily break the security property of the target application. This can happen for multiple reasons, including unused values and masked faults. In the case of unused values, the fault injection attack successfully modifies a value in memory, but the application does not use that value, resulting in an ineffective fault. In the case of masked fault, the successful injection of multiple faults results in a state where the faults cancel the effects of each other, leading to an outcome equivalent to the normal functioning of the application. Hence, successful faults based on reachability signify that the faults manifested, but only exploitability can guarantee that the effects caused by the faults can be used to weaken the application. AR-MORY [26], SAMVA [21], and CELTIC [69] takes exploitability conditions. Exploring more into simulators that would differentiate between effective/ineffective faults and reachability/exploitability ity would give more clarity into fault manifestation.

**Performance versus Accuracy.** In [72], it is shown that there is a trade-off between performance and accuracy. Targeting higher abstraction layers would result in better performance, while lower abstraction levels would be more focused on achieving accuracy.

### 7 Conclusion

This paper presents the state of the art of pre- and post-silicon fault simulators and provides a novel taxonomy to classify them. We conclude by listing the main takeaways for the community of users who evaluate the resistance of their design against fault injection attacks, the developers who build tools in this domain, and the wide research community working in this field.

**Takeaway for the Community.** As security professionals, we are taught that *security must be built-in and not added as an afterthought.* Fault injection simulators let a designer test the security of an application early on in different design stages. This reduces the cost of building secure applications, as performing security evaluations of the design becomes possible. Different design stages carry different information about the result. Therefore, we would suggest targeting multiple design stages to combine the advantages of these two possible approaches. When developing an integrated solution, i.e., hardware and software, using a combined tool gives a complete picture

of the product's security as a whole. This also lets designers evaluate trade-offs regarding which countermeasures to implement in hardware and which in software. Having more *open source* tools would be helpful. As illustrated in Table 1, for software tools, there is a preference for ARM architectures, whereas for hardware tools, there is an inclination towards RISC-V architectures. While hardware tools preferring RISC-V over ARM due to RISC-V's open-source nature is understandable, having more software tools for RISC-V would help in verifying fault-resistance for RISC-V, too.

**Takeaway for Developers.** Many tools available in the public domain are proof of concepts made for research purposes. We should not underestimate the effort required to transform such artifacts into mature products. The developers' community could provide a practical contribution to the field by refining such PoCs to become usable without too much friction. Additionally, we suggest using the taxonomy presented in section 2 when presenting new tools to the user community. Employing a common terminology to classify fault injection simulators would help users in comparing them and selecting the ones that best suit their needs. Another aspect that would help in comparing different tools would be establishing common benchmarks to evaluate them on so that results can be used in a direct comparison.

There is a benchmark available for software tools and another for hardware tools: https://lazart. gricad-pages.univ-grenoble-alpes.fr/fissc/, and https://opencores.org/projects. We note that these two benchmarks have different quality levels: while the collection of software test cases presents, for each application, a simple implementation that is then gradually hardened against physical attacks by adding a diverse set of countermeasures, the collection of hardware designs is just a list of IPs. To make this more suitable for the testing of fault injection simulators, the community of hardware designers could provide different versions of the same design, starting with a naive implementation and then gradually adding countermeasures. Having the same design with and without countermeasures is necessary to properly test a tool's accuracy in identifying faults.

Takeaway for Researchers. Fault injection is a field relevant to the security industry, with ample research opportunities. We identify the following three research directions: (1) automated selection of the sensitive region, (2) faults' feasibility, and (3) state space explosion. Automatically selecting the sensitive region (1) has the practical advantage of sparing the user a tedious manual analysis, which might be error-prone. Feasibility of faults (2) is a research problem that needs more attention. It is of practical importance, but it poses significant challenges. First, the information needed to evaluate faults' feasibility could depend on the fault technique: e.g., laser fault injection strongly depends upon a circuit's layout, while inducing instruction skips by glitching the power supply has no dependence on the layout. Evaluating the feasibility of faults requires at least some information about the hardware, so it could not be doable for software tools. Nevertheless, progress in this area would significantly advance the field of pre-silicon fault injection.

State space explosion (3) occurs when a framework attempts an exhaustive search over all the possible fault combinations. This approach is not feasible in the case of large circuit designs, voluminous software projects, or when considering higher mutation orders. However, leaving out some fault combinations could lead to false negatives. This problem could be tackled by developing sound heuristics that enable tools to reduce the search space without missing potential faults.

It could be interesting to compare the false positives rate among tools that give formal guarantees on the absence of false negatives and then compare these results with the same question for tools providing empirical guarantees on the absence of false negatives. This would be a cost metric for providing such a guarantee, and could be an important factor to consider when a user is choosing which tool to use.

The field of fault injection needs more FOSS<sup>22</sup> tools to foster progress and innovation. When developing a proof of concept application, making the source code available under a permissive license is necessary. To promote real-world application of the presented ideas, care should be taken in providing adequate documentation that guides the user in setting up and using the PoC. This would help the developers' community make mature products out of PoCs.

### References

- Adhikary, A., Buhan, I.: Sok: Assisted fault simulation existing challenges and opportunities offered by AI. In: Zhou, J., Batina, L., Li, Z., Lin, J., Losiouk, E., Majumdar, S., Mashima, D., Meng, W., Picek, S., Rahman, M.A., Shao, J., Shimaoka, M., Soremekun, E.O., Su, C., Teh, J.S., Udovenko, A., Wang, C., Zhang, L.Y., Zhauniarovich, Y. (eds.) Applied Cryptography and Network Security Workshops -ACNS 2023 Satellite Workshops, ADSC, AIBlock, AIHWS, AIoTS, CIMSS, Cloud S&P, SCI, SecMT, SiMLA, Kyoto, Japan, June 19-22, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13907, pp. 178–195. Springer, Kyoto, Japan (2023)
- Agosta, G., Barenghi, A., Pelosi, G., Scandale, M.: Differential fault analysis for block ciphers: an automated conservative analysis. In: Poet, R., Rajarajan, M. (eds.) Proceedings of the 7th International Conference on Security of Information and Networks, Glasgow, Scotland, UK, September 9-11, 2014. p. 137. ACM, Glasgow, Scotland, UK (2014)
- Aidemark, J., Vinter, J., Folkesson, P., Karlsson, J.: GOOFI: generic object-oriented fault injection tool. In: 2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings. p. 668. IEEE Computer Society, San Francisco, CA, USA (2003)
- Arribas, V., Wegener, F., Moradi, A., Nikova, S.: Cryptographic fault diagnosis using verfi. In: 2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020. pp. 229–240. IEEE, San Jose, CA, USA (2020)
- 5. Baier, C., Katoen, J.: Principles of model checking. MIT Press, Cambridge, MA, USA (2008)
- Balasch, J., Gierlichs, B., Verbauwhede, I.: An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In: Breveglieri, L., Guilley, S., Koren, I., Naccache, D., Takahashi, J. (eds.) 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011. pp. 105–114. IEEE Computer Society (2011)
- Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. Proc. IEEE 100(11), 3056–3076 (2012)
- Becker, M., Baldin, D., Kuznik, C., Joy, M.M., Xie, T., Müller, W.: XEMU: an efficient QEMU based binary mutation testing framework for embedded software. In: Jerraya, A., Carloni, L.P., Maraninchi, F., Regehr, J. (eds.) Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012. pp. 33–42. ACM, Tampere, Finland (2012)
- Bosio, A., Natale, G.D.: LIFTING: A flexible open-source fault simulator. In: 17th IEEE Asian Test Symposium, ATS 2008, Sapporo, Japan, November 24-27, 2008. pp. 35–40. IEEE Computer Society, Sapporo, Japan (2008)
- Breier, J., Hou, X., Liu, Y.: Fault attacks made easy: Differential fault analysis automation on assembly code. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2018(2), 96–122 (2018)
- Burchard, J., Gay, M., Ekossono, A.M., Horácek, J., Becker, B., Schubert, T., Kreuzer, M., Polian, I.: Autofault: Towards automatic construction of algebraic fault attacks. In: 2017 Workshop on Fault

<sup>&</sup>lt;sup>22</sup> Free and Open Source Software. Here "free" is the notorious "free as in freedom".

Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017. pp. 65–72. IEEE Computer Society, Taipei, Taiwan (2017)

- Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association, San Diego, CA, USA (2008)
- Chen, Z., Narayanan, N., Fang, B., Li, G., Pattabiraman, K., DeBardeleben, N.: Tensorfi: A flexible fault injection framework for tensorflow applications. In: Vieira, M., Madeira, H., Antunes, N., Zheng, Z. (eds.) 31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020. pp. 426–435. IEEE, Coimbra, Portugal (2020)
- Civera, P., Macchiarulo, L., Rebaudengo, M., Reorda, M.S., Violante, M.: Exploiting FPGA for accelerating fault injection experiments. In: 7th IEEE International On-Line Testing Workshop (IOLTW 2001), 9-11 July 2001, Taormina, Italy. pp. 9–13. IEEE Computer Society, Taormina, Italy (2001)
- Developer, A.: Gnu toolchain for arm (2024), https://developer.arm.com/Tools%20and%20Software/ GNU%20Toolchain, accessed: 2024-06-29
- Dureuil, L., Petiot, G., Potet, M., Le, T., Crohen, A., de Choudens, P.: FISSC: A fault injection and simulation secure collection. In: Skavhaug, A., Guiochet, J., Bitsch, F. (eds.) Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9922, pp. 3–11. Springer, Trondheim, Norway (2016)
- Dureuil, L., Potet, M., de Choudens, P., Dumas, C., Clédière, J.: From code review to fault injection attacks: Filling the gap using fault model inference. In: Homma, N., Medwed, M. (eds.) Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers. Lecture Notes in Computer Science, vol. 9514, pp. 107–124. Springer, Bochum, Germany (2015)
- Eldridge, S., Buyuktosunoglu, A., Bose, P.: Chiffre: A configurable hardware fault injection framework for risc-v systems (2018)
- Ferraretto, D., Pravadelli, G.: Efficient fault injection in QEMU. In: 16th Latin-American Test Symposium, LATS 2015, Puerto Vallarta, Mexico, March 25-27, 2015. pp. 1–6. IEEE Computer Society, Puerto Vallarta, Mexico (2015)
- Fuchs, E.: An evaluation of the error detection mechanisms in MARS using software-implemented fault injection. In: Hlawiczka, A., Silva, J.G., Simoncini, L. (eds.) Dependable Computing - EDCC-2, Second European Dependable Computing Conference, Taormina, Italy, October 2-4, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1150, pp. 73–90. Springer, Taormina, Italy (1996)
- Gicquel, A., Hardy, D., Heydemann, K., Rohou, E.: SAMVA: static analysis for multi-fault attack paths determination. In: Kavun, E.B., Pehl, M. (eds.) Constructive Side-Channel Analysis and Secure Design - 14th International Workshop, COSADE 2023, Munich, Germany, April 3-4, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13979, pp. 3–22. Springer, Munich, Germany (2023)
- Grycel, J.T., Schaumont, P.: Simplifi: Hardware simulation of embedded software fault attacks. Cryptogr. 5(2), 15 (2021)
- Hari, S.K.S., Tsai, T., Stephenson, M., Keckler, S.W., Emer, J.S.: SASSIFI: an architecture-level fault injection tool for GPU application resilience evaluation. In: 2017 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2017, Santa Rosa, CA, USA, April 24-25, 2017. pp. 249–258. IEEE Computer Society, Santa Rosa, CA, USA (2017)
- Hauschild, F., Garb, K., Auer, L., Selmke, B., Obermaier, J.: ARCHIE: A qemu-based framework for architecture-independent evaluation of faults. In: 18th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2021, Milan, Italy, September 17, 2021. pp. 20–30. IEEE, Milan, Italy (2021)
- Hayashi, Y., Hasegawa, R., Wadatsumi, T., Monta, K., Miki, T., Nagata, M.: Fault injection attacks exploiting high voltage pulsing over si-substrate backside of ic chips. https://fdtc.deib.polimi.it/FDTC24/slides/FDTC2024-talk-1.2.pdf (2024), presented at FDTC 2024, September 4, 2024, Halifax, Canada

23

- Hoffmann, M., Schellenberg, F., Paar, C.: ARMORY: fully automated and exhaustive fault simulation on ARM-M binaries. IEEE Trans. Inf. Forensics Secur. 16, 1058–1073 (2021)
- Höller, A., Krieg, A., Rauter, T., Iber, J., Kreiner, C.: Qemu-based fault injection for a system-level analysis of software countermeasures against fault attacks. In: 2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015. pp. 530–533. IEEE Computer Society, Madeira, Portugal (2015)
- Höller, A., Schonfelder, G., Kajtazovic, N., Rauter, T., Kreiner, C.: FIES: A fault injection framework for the evaluation of self-tests for cots-based safety-critical systems. In: 15th International Microprocessor Test and Verification Workshop, MTV 2014, Austin, TX, USA, December 15-16, 2014. pp. 105–110. IEEE Computer Society, Austin, TX, USA (2014)
- Hou, X., Breier, J., Zhang, F., Liu, Y.: Fully automated differential fault analysis on software implementations of block ciphers. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2019(3), 1–29 (2019)
- Jenn, E., Arlat, J., Rimén, M., Ohlsson, J., Karlsson, J.: Fault injection into VHDL models: The MEFISTO tool. In: Digest of Papers: FTCS/24, The Twenty-Fourth Annual International Symposium on Fault-Tolerant Computing, Austin, Texas, USA, June 15-17, 1994. pp. 66–75. IEEE Computer Society, Austin, Texas, USA (1994)
- Jeong, E., Lee, N., Kim, J., Kang, D., Ha, S.: FIFA: A kernel-level fault injection framework for armbased embedded linux system. In: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017. pp. 23–34. IEEE Computer Society, Tokyo, Japan (2017)
- Joye, M., Tunstall, M. (eds.): Fault Analysis in Cryptography. Information Security and Cryptography, Springer, Berlin, Heidelberg (2012)
- K., K., Rebeiro, C.: Faultmeter: Quantitative fault attack assessment of block cipher software. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2023(2), 212–240 (2023)
- K., K., Roy, I., Rebeiro, C., Hazra, A., Bhunia, S.: FEDS: comprehensive fault attack exploitability detection for software implementations of block ciphers. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2020(2), 272–299 (2020)
- 35. Kanawati, G.A., Kanawati, N.A., Abraham, J.A.: FERRARI: A tool for the validation of system dependability properties. In: Digest of Papers: FTCS-22, The Twenty-Second Annual International Symposium on Fault-Tolerant Computing, Boston, Massachusetts, USA, July 8-10, 1992. pp. 336–344. IEEE Computer Society, Boston, Massachusetts, USA (1992)
- Karlsson, J., Lidén, P., Dahlgren, P., Johansson, R., Gunneflo, U.: Using heavy-ion radiation to validate fault-handling mechanisms. IEEE Micro 14(1), 8–23 (1994)
- 37. Khanna, P., Rebeiro, C., Hazra, A.: XFC: A framework for exploitable fault characterization in block ciphers. In: Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017. pp. 8:1–8:6. ACM, Austin, TX, USA (2017)
- Madeira, H., Rela, M.Z., Moreira, F., Silva, J.G.: RIFLE: A general purpose pin-level fault injector. In: Echtle, K., Hammer, D.K., Powell, D. (eds.) Dependable Computing - EDCC-1, First European Dependable Computing Conference, Berlin, Germany, October 4-6, 1994, Proceedings. Lecture Notes in Computer Science, vol. 852, pp. 199–216. Springer, Berlin, Germany (1994)
- Moro, N., Heydemann, K., Encrenaz, E., Robisson, B.: Formal verification of a software countermeasure against instruction skip attacks. J. Cryptogr. Eng. 4(3), 145–156 (2014)
- Nasahl, P., Osorio, M., Vogel, P., Schaffner, M., Trippel, T., Rizzo, D., Mangard, S.: SYNFI: pre-silicon fault analysis of an open-source secure element. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2022(4), 56–87 (2022)
- 41. O'Flynn, C., van Woudenberg, J.: The Hardware Hacking Handbook: Breaking Embedded Security with Hardware Attacks. No Starch Press, San Francisco, CA (Nov 2021)
- 42. Parasyris, K., Tziantzoulis, G., Antonopoulos, C.D., Bellas, N.: Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates. In: 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014. pp. 622–629. IEEE Computer Society, Atlanta, GA, USA (2014)

- Pensec, W., Lapôtre, V., Gogniat, G.: Scripting the unpredictable: Automate fault injection in caba simulation for vulnerability assessment. In: Euromicro Conference on Digital System Design (DSD 2024). Paris (August 2024)
- Porpodas, V.: ZOFI: zero-overhead fault injection tool for fast transient fault coverage analysis. CoRR abs/1906.09390, n/a (2019)
- Potet, M., Mounier, L., Puys, M., Dureuil, L.: Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA. pp. 213–222. IEEE Computer Society, Cleveland, Ohio, USA (2014)
- Richter-Brockmann, J., Shahmirzadi, A.R., Sasdrich, P., Moradi, A., Güneysu, T.: FIVER robust verification of countermeasures against fault injections. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2021(4), 447–473 (2021)
- 47. Riscure: Fisim: An open-source deterministic fault attack simulator prototype (2024), https://github. com/Riscure/FiSim, accessed: 2024-06-29
- Rivière, L., Bringer, J., Le, T., Chabanne, H.: A novel simulation approach for fault injection resistance evaluation on smart cards. In: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015. pp. 1–8. IEEE Computer Society, Graz, Austria (2015)
- Roy, I., Rebeiro, C., Hazra, A., Bhunia, S.: SAFARI: automatic synthesis of fault-attack resistant block cipher implementations. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 39(4), 752–765 (2020)
- Roy, I., Rebeiro, C., Hazra, A., Bhunia, S.: Faultdroid: An algorithmic approach for fault-induced information leakage analysis. ACM Trans. Design Autom. Electr. Syst. 26(1), 2:1–2:27 (2021)
- Saha, S., Alam, M., Bag, A., Mukhopadhyay, D., Dasgupta, P.: Learn from your faults: Leakage assessment in fault attacks using deep learning. J. Cryptol. 36(3), 19 (2023)
- Saha, S., Jap, D., Patranabis, S., Mukhopadhyay, D., Bhasin, S., Dasgupta, P.: Automatic characterization of exploitable faults: A machine learning approach. IEEE Trans. Inf. Forensics Secur. 14(4), 954–968 (2019)
- 53. Saha, S., Kumar, S.N., Patranabis, S., Mukhopadhyay, D., Dasgupta, P.: ALAFA: automatic leakage assessment for fault attack countermeasures. In: Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019. p. 136. ACM, Las Vegas, NV, USA (2019)
- 54. Saha, S., Kumar, U., Mukhopadhyay, D., Dasgupta, P.: An automated framework for exploitable fault identification in block ciphers - A data mining approach. In: Kühne, U., Danger, J., Guilley, S. (eds.) PROOFS 2017, 6th International Workshop on Security Proofs for Embedded Systems, Taipei, Taiwan, September 29th, 2017. EPiC Series in Computing, vol. 49, pp. 50–67. EasyChair, Taipei, Taiwan (2017)
- Saha, S., Kumar, U., Mukhopadhyay, D., Dasgupta, P.: An automated framework for exploitable fault identification in block ciphers. J. Cryptogr. Eng. 9(3), 203–219 (2019)
- Saha, S., Mukhopadhyay, D., Dasgupta, P.: Expfault: An automated framework for exploitable fault characterization in block ciphers. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2018(2), 242–276 (2018)
- Sakiyama, K., Li, Y., Iwamoto, M., Ohta, K.: Information-theoretic approach to optimal differential fault analysis. IEEE Trans. Inf. Forensics Secur. 7(1), 109–120 (2012)
- Schirmeier, H., Hoffmann, M., Dietrich, C., Lenz, M., Lohmann, D., Spinczyk, O.: Fail\*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In: 11th European Dependable Computing Conference, EDCC 2015, Paris, France, September 7-11, 2015. pp. 245–255. IEEE Computer Society, Paris, France (2015)
- Selmke, B., Heyszl, J., Sigl, G.: Attack on a DFA protected AES by simultaneous laser fault injections. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016. pp. 36–46. IEEE Computer Society (2016)
- 60. Simevski, A., Kraemer, R., Krstic, M.: Automated integration of fault injection into the ASIC design flow. In: 2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFTS 2013, New York City, NY, USA, October 2-4, 2013. pp. 255–260. IEEE Computer Society, New York City, NY, USA (2013)

A. Adhikary et al.

- 61. Srivastava, M., SLPSK, P., Roy, I., Rebeiro, C., Hazra, A., Bhunia, S.: SOLOMON: an automated framework for detecting fault attack vulnerabilities in hardware. In: 2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020. pp. 310–313. IEEE, Grenoble, France (2020)
- Svenningsson, R., Vinter, J., Eriksson, H., Törngren, M.: MODIFI: A model-implemented fault injection tool. In: Schoitsch, E. (ed.) Computer Safety, Reliability, and Security, 29th International Conference, SAFECOMP 2010, Vienna, Austria, September 14-17, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6351, pp. 210–222. Springer, Vienna, Austria (2010)
- Tang, A., Sethumadhavan, S., Stolfo, S.J.: CLKSCREW: exposing the perils of security-oblivious energy management. In: Kirda, E., Ristenpart, T. (eds.) 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 1057–1074. USENIX Association, Vancouver, BC, Canada (2017)
- 64. Timmers, N., Spruyt, A., Witteman, M.: Controlling PC on ARM using fault injection. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016. pp. 25–35. IEEE Computer Society (2016). https://doi.org/10.1109/FDTC.2016.18, https://doi.org/10.1109/FDTC.2016.18
- 65. Tollec, S., Asavoae, M., Couroussé, D., Heydemann, K., Jan, M.: μarchifi: Formal modeling and verification strategies for microarchitectural fault injections. In: Nadel, A., Rozier, K.Y. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023. pp. 101–109. IEEE, Ames, IA, USA (2023)
- 66. Tollec, S., Hadzic, V., Nasahl, P., Asavoae, M., Bloem, R., Couroussé, D., Heydemann, K., Jan, M., Mangard, S.: Fault-resistant partitioning of secure cpus for system co-verification against faults. IACR Cryptol. ePrint Arch. n/a(n/a), 247 (2024)
- 67. van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., Giuffrida, C.: Drammer: Deterministic rowhammer attacks on mobile platforms. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1675–1689. ACM, Vienna, Austria (2016)
- Wang, H., Li, H., Rahman, F., Tehranipoor, M.M., Farahmandi, F.: Sofi: Security property-driven vulnerability assessments of ics against fault-injection attacks. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 41(3), 452–465 (2022)
- Werner, V., Maingault, L., Potet, M.: An end-to-end approach for multi-fault attack vulnerability assessment. In: 17th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2020, Milan, Italy, September 13, 2020. pp. 10–17. IEEE, Milan, Italy (2020)
- Wiersma, N., Pareja, R.: Safety != security: On the resilience of ASIL-D certified microcontrollers against fault injection attacks. In: 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017. pp. 9–16. IEEE Computer Society, Taipei, Taiwan (2017)
- Winter, S., Piper, T., Schwahn, O., Natella, R., Suri, N., Cotroneo, D.: GRINDER: on reusability of fault injection tools. In: Zhu, H., Hao, D., Mariani, L., Subramanyan, R. (eds.) 10th IEEE/ACM International Workshop on Automation of Software Test, AST 2015, Florence, Italy, May 23-24, 2015. pp. 75–79. IEEE Computer Society, Florence, Italy (2015)
- van Woudenberg, J., Velegalati, R., Breunesse, C.B., Vermoen, D.: Improving cpu fault injection simulations: Insights from rtl to instruction-level models. https://fdtc.deib.polimi.it/FDTC24/slides/ FDTC2024-talk-1.3.pdf (2024), accessed: September 30, 2024
- Zhang, F., Guo, S., Zhao, X., Wang, T., Yang, J., Standaert, F., Gu, D.: A framework for the analysis and evaluation of algebraic fault attacks on lightweight block ciphers. IEEE Trans. Inf. Forensics Secur. 11(5), 1039–1054 (2016)

## A Fault Injection Simulators Cheatsheet

Table 1: We use the following conventions: A-Automatic, H-Hybrid, M-Manual;  $\bigcirc$ -formally proven absence of false negatives,  $\bigcirc$ -formally proven absence of false negatives and positives;  $\diamondsuit$ -no guarantees,  $\diamondsuit$ - empirical guarantee of false negatives absence.  $\blacklozenge$ -empirical guarantee of both false negatives and false positives absence; \* this property is not explicitly stated but inferred from context.

	Fault model					Use cases		
Tool	Step	Region	Fault effect	Mutation order	Guarantees	Architecture	Tested on	Code
Lazart	S2	М	Test inversion	Unlimited	<b>♦</b>	Any	Verify function	No
Formal Verif	S3	М	Instruction skip/ replacement	1	•	ARM M3	AES, AES (MiBench), SHA0 (MiBench)	No
FIES	S4	Η	Bit-flip, Stuck-at, Given value	Unlimited	$\diamond$	All QEMU targets	Access control application and control flow errors; memory- related faults	Yes
FiSim	S4	А	Transient instruction skip, bit-flip	1	$\diamond$	ARM32, ARM64	Secure boot implementa- tion	Yes
ARMORY	S4	Η	Permanent/ transient/ until overwrite instruction/ register skip, byte-set, byte-clear, bit-flip, clear, fill, bit-set, bit-clear	Unlimited	$\diamond$	ARMv6-M, ARMv7-M, ARMv7-EM	DFA on AES, secure bootloader	Yes
ARCHIE	S4	М	Permanent/ transient set-0/set1, toggle	Unlimited	$\diamond$	ARM, RISC-V	TinyAES, secure bootloader	Yes
SAMVA	S4	М	Instruction- skip	Unlimited	$\diamond$	ARM	PIN Verify	No

CELTIC	S4	Н	TSFMs	Unlimited	$\diamond$	ARM M4	VerifyPIN, GetChallenge	No
FISSA	H1	М	Single/ multiple bit-flips, set/reset	1 or 2	$\diamond$	N/A	DIFT on D-RISCY	Yes
SoFI	H2	А	Bit-flip, stuck-at	1 or $2^*$	$\diamond$	N/A	OpenCores benchmark	No
SYNFI	H2	М	Bit-flip, stuck-at	Unlimited	lacksquare	N/A	OpenTitan, Ibex (part of OpenTitan)	Yes
SimpliFI	H3	М	Clock glitch	1 glitch but multiple effects	$\diamond$	N/A	RISC-V instructions, AES	No
Chiffre	H4	М	Pseudo- random bit-flip, multiple bit-flips, stuck-at	Unlimited	$\diamond$	N/A	Rocket Chip's CSR registers	Yes
µarchiFI	С	А	Single/ multiple bit-flips, set/reset	Limited by number of transitions	•*	Any	VerifyPIN, Key schedule AES; Both on RISC-V	Yes
EFS	С	М	Modify CPU/mem- ory state (PC, registers, stack)	Unlimited	$\diamond$	N/A (smart cards)	VerifyPIN, AES	No
Fault- Resistant Partitioning	С	Η	Bit-flip, stuck-at	Unlimited*		Any*	OpenTitan, Ibex (part of OpenTitan), VerifyPIN, tiny AES	Yes

SoK: The apprentice guide to automated fault injection simulation

27