Cryptographic Fault Diagnosis using VerFI

Victor Arribas*, Felix Wegener†, Amir Moradi† and Svetla Nikova*
* KU Leuven, imec-COSIC, Belgium {firstname.lastname}@esat.kuleuven.be
† Ruhr University Bochum, Horst Görtz Institute for IT Security, Germany {firstname.lastname}@rub.de

Abstract—**Historically, fault diagnosis for integrated circuits has singularly dealt with reliability concerns. In contrast, a cryptographic circuit needs to be primarily evaluated concerning information leakage in the presence of maliciously crafted faults. While Differential Fault Attacks (DFAs) on symmetric ciphers have been known for over 20 years, recent developments have tried to structurally classify the attackers' capabilities as well as the properties of countermeasures. Correct realization of countermeasures should still be manually verified, which is error-prone and infeasible for even moderate-size real-world designs. Here, we introduce the concept of Cryptographic Fault Diagnosis, which revises and shapes the notions of fault diagnosis in reliability testing to the needs of evaluating cryptographic implementations. Additionally, we present VerFI, which materializes the idea of Cryptographic Fault Diagnosis. It is a fully automated, open-source fault detection tool processing the gate-level representation of arbitrary cryptographic implementations. By adjusting the bounds of the underlying adversary model, VerFI allows us to rapidly examine the desired fault detection/correction capabilities of the given implementation. Among several case studies, we demonstrate its application on an implementation of LED cipher with combined countermeasures against side-channel analysis and fault-injection attacks (published at CRYPTO 2016). This experiment revealed general implementation flaws and undetectable faults leading to successful DFA on the protected design with full-key recovery.**

Index Terms—**Fault Diagnosis, Fault Simulation, Verification, Tool, Differential Fault Attack, DFA, ParTI**

I. Introduction

Current technological trends, e.g., the Internet of Things and Edge Computing, lead to the deployment of many distributed computing devices in need of strong cryptography. In this new scenario of ubiquitous computing, a real-world attacker can easily obtain physical access to a device. Hence, the need to secure implementations against both passive physical attacks (side-channels) and active physical attacks (fault injections) arises. While many tools for verification of side-channel resistant implementations exist, ranging from purely formal verification to more practical evaluations [1]–[6], no automated procedure has yet been suggested to evaluate the countermeasures against fault-injection attacks in the literature.

Fault injection countermeasures started by proposing redundancy techniques, either consecutively or in parallel, together with opportunities to detect (or correct) certain faults [7], [8]. While this is usually referred to as Concurrent Error Detection (CED), the vast majority of the works focused on Error Detecting Codes (EDCs) with parity as the most straightforward case [9] and linear codes as more sophisticated solutions [10], [11]. Besides the fact that most of the aforementioned works provide only ad-hoc solutions, almost none of them consider a formal description of the adversary's capabilities. In contrast, ATP Security [12] provides a formal model and investigates algorithmic aspects for taper-proof security. Later, Private Circuits II [13], CAPA [14], M&M [15], and an extension of a glitch-free multiparty protocol [16] have defined a boundary for the adversary in order to guarantee the fault detection. A more precise formalization of the adversary model for fault injection has only recently been introduced in [17].

Outside the cryptography community, the field of integrated circuits testing has developed plenty of simulation tools and analysis techniques to examine the effect of possible manufacturing failures in a product, also known as reliability analysis. Testing through simulation has the advantage that it can be performed in great detail and is potentially very accurate since it gives realistic emulation of faults and detailed monitoring of their consequences on the system [18]. Moreover, practical evaluations of a real product need enormous amount of time as well as considerable expertise. In addition to this, a preliminary evaluation is advantageous in order to reduce costs of the fabrication process. While many fault simulation tools for reliability analysis of integrated circuits exist, they are commonly limited to one-bit faults [19], [20], which severely limit their applicability for cryptographic fault analysis.

Several previous works can assist the automated fault analysis of a formal cipher description to identify and facilitate potential fault attacks. Zhang *et al.* [21] propose a method to automate Algebraic Faults Attacks (AFA) using SMT solvers, while more recent works of Khana *et al.* [22] and Saha *et al.* [23] provide frameworks for fault characterization in cryptographic algorithms tailored to Differential Fault Analysis (DFA) attacks. These tools can surely facilitate an active attack. However, assuming a countermeasure to fault-injection attacks where the adversary's boundary is defined, no public tools are currently known to (automatically) examine the security of such an implementation under the defined adversary model. Apart from the difficulty of practical evaluations, it is highly challenging to limit the injected faults based on the assumed fault model.

For this reason, only a reduced number of works proposing countermeasures against Fault Analysis evaluate their scheme [9], [15], [24]. To this end, the authors had to undergo an extremely tedious process of manually modifying the HDL code or the netlist to be able to perform fault injection in simulation. Hence, the importance of a unified and automated framework to facilitate the evaluation of fault resistant implementations is highlighted.

Our Contribution. In this work we try to close this gap by introducing the concepts of Cryptographic Fault Diagnosis and providing the corresponding missing tool to automate it. With bit-level granularity, VerFI operates on the netlist of a given hardware implementation. It is a fault simulator, where the user adjusts the specifications and capabilities of the adversary. Then, utilizing a set of test vectors, the tool simulates the design while injecting the faults fitting to the adjusted bounds. The tool reports the fault coverage (rate of detected cases), and details of the undetected cases, including location and type of fault, input test vector, and clock cycle(s). We demonstrate the capabilities of VerFI by assessing the security of multiple exemplary hardware designs equipped with fault-injection countermeasures known through public literature. Most notably, considering ParTI [25] as a combination of Threshold Implementations (TI) [26] and Binary EDC, using VerFI we exhibit a design flaw and the cases where bounded faults are left undetected, which enable a successful full-key-recovery DFA attack. Additionally, we verify the security of the schemes CAPA [14], M&M [15], and Impeccable Circuits [17] by analyzing several implementations, confirming the theoretical claims of these schemes.

II. Preliminaries

In this section, we give an introduction to reliability testing in digital electronics and fault simulation, and introduce the most common fault models used in cryptographic implementations.

A. Integrated Circuits Testing

The role of *reliability testing* in digital circuits is to evaluate the correct functionality of a particular design, thereby examining whether something went wrong during the design or the manufacturing process. The role of *diagnosis* is to ascertain the underlying reason for a malfunction [27]. Reliability testing is a crucial part of the circuit design flow, done along different stages of the design to ensure correct functionality. Multiple factors can affect the manufacturing process, leading to a wrong result. Simulation techniques help to anticipate and determine what failed during the process. Fault simulation has been extensively used in fault diagnosis to achieve this goal.

B. Fault Simulation

In fault simulation, the fault-free circuit is simulated to get the correct output given a specific input test vector. Then, the faulty circuit is simulated with the same input. If the corresponding outputs are different, it is said that the given input test vector detects such a fault. Instead, when the effect of the fault is not propagated to the outputs, it is concluded that the fault is not observed and hence not detected by the given test vector. Different fault simulation techniques exist in the literature, but the most common ones are serial, parallel, deductive, and concurrent [27]. The advantage of the last two techniques is that in one input-output pass, it is possible to asses the effect of every fault, making them highly efficient. However, in the presence of a significant number of faults, the memory demand can be immensely high. Fault diagnosis of digital circuits is most often addressed with single-fault simulation models due to the impossibility of covering all possible combinations of multiple faults [19], [20].

There are multiple commercial Electronic Design Automation (EDA) tools featuring powerful fault simulators, tailored to address IC testability. The typical workflow of these tools is to generate a scan circuit by replacing the functional flip-flops with scan flip-flops and then run an Automatic Test Pattern Generation (ATPG) tool which only handles the combinational part of the circuit provided the full scan design. There are also several works on open-source fault simulators for fault diagnosis. The first one, known as PROOFS [28], implements a single stuck-at fault simulator. Subsequent work [29], optimizes the fault simulation without changes in the fault model. Later, the work [30] extends this to multiple and single stuck-at faults, and a subset of Single Event Upset (SEU) limited to registers, hampering fault injection inside combinatorial circuits.

C. Fault Models in Cryptographic Implementations

Multiple fault models have been described in the literature concerning fault injection in cryptographic algorithms [17], [31]–[33]. All of them commonly concentrate on how many faults are allowed, when those faults should be injected or their duration, and the type of the fault. Depending on how many faults are allowed, some fault models distinguish between single- and multiple-bit fault injections. Regarding the number of faults, we encounter two different approaches. On the one hand, countermeasures like duplication or EDCs [25] limit the number of faults to be injected (usually per cycle) based on the level of redundancy or the distance of the underlying code. On the other hand, countermeasures utilizing Message Authentication Code (MAC) tags, *e.g.*, CAPA [14] or M&M [15] restrict areas of the circuit to be faulted instead of limiting the number of injected faults. With respect to the timing of fault injection, the authors of Impeccable Circuits [17] introduce the Univariate ($\mathcal{M}$) and Multivariate ($\mathcal{M}^*$) concepts, in which stuck-at or SEU faults can be injected in a single clock cycle or in multiple clock cycles respectively.

Usually, previous models either contemplate a uniform or a biased fault model: in the uniform model the faults follow a uniform distribution, where each of them has the same probability of occurring. In contrast, a biased fault model is characterized by an increased chance of occurrence for particular faults (specific areas only).

III. Cryptographic Fault Diagnosis

In this work, we aim to bring the fault diagnosis knowledge from reliability testing to the cryptographic setting for the evaluation of fault countermeasures. We adapt these concepts to the needs of cryptography to define the notion of Cryptographic Fault Diagnosis. We compare the goal of testing in IC design versus the goal of evaluating fault-resistant implementations of cryptographic algorithms. Furthermore, we introduce a new fault model to complement the state of the art and provide a consolidated way of characterizing any fault model.

A. Conventional vs. Cryptographic Fault Diagnosis

The process of reliability testing and evaluation of fault-resistant cryptographic implementations differ in multiple concepts:

- The **primary goal** is the principal difference between both procedures. While the first one aims at ascertaining whether a defect occurred during the manufacturing process, the second one's goal is to evaluate the performance of a countermeasure against fault-injection attacks.
- **Detecting** a fault in reliability testing means that the fault propagates to the outputs of the circuit. On the other hand, in cryptography, a fault is detected when the integrated countermeasure reacts to a faulty execution, *e.g.*, by raising an abort or status signal, or by propagating an error randomizing the internal state.
- The **origin** of faults in the first scenario comes from small defects during the manufacturing process, which makes these faults purely random and unpredictable. On the contrary, cryptographic implementations face malicious users injecting intentional faults, specially crafted to bypass the security of the system.
- The **number** of simulated faults also differs. Circuits are tested with single-bit faults due to the high coverage this model provides and the impossibility of simulating all combinations of multiple faults. Conversely, the methods used to inject malicious faults often target multiple faults at the same time, *e.g.*, laser fault injection.
- **Coverage** in testing (associated with a test set) is defined as the ratio of faults that the test set can detect over the total number of possible faults in the circuit. The coverage in a cryptographic implementation is associated with the underlying countermeasure. It is defined as the ratio of faults detected over the total number of faults injected.

Given the aforementioned differences, a simulator with a different testing approach is needed to conduct a successful cryptographic fault diagnosis using particular fault simulation methodologies. The *result* of the test shall give an accurate assessment of the performance of the countermeasure. The simulator needs to *identify* the detection mechanism and to recognize whether the circuit reacts to the injected fault. Probably as the most challenging task, the simulator needs to be highly flexible to be able to emulate all different possible *origins* (fault model defined by the user), which represent the hypothetical attack methodology. Due to the need to simulate *multiple*-fault injections, the use of deductive or concurrent fault simulation becomes impractical when considering a high number of possible combinations. Hence, we identify more suitable methodologies, such as serial or parallel fault simulation. As a final result, meaningful *coverage* has to be reported following the type of implemented countermeasure.

B. Consolidating Fault Models

The most extensively used model in fault diagnosis for digital circuits is the stuck-at-0/1 fault model. These are faults that occur in the interconnections of Boolean gates, *i.e.*, wires or nets. Additionally, bit-flip faults (SEUs) occurring in the memory elements, among others, are also used.

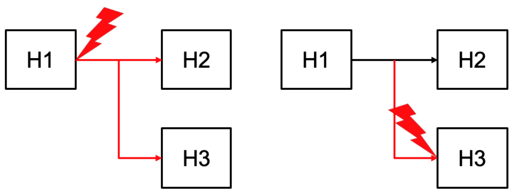


Fig. 1: Injecting fault on a hierarchical module (H3) from the inputs in the gate-output fault model $\mathcal{M}_{\mathcal{O}}$ (left) and in the gate-input fault model $\mathcal{M}_{\mathcal{I}}$ (right).

To define a unified fault model for cryptographic implementations, we benefit from the extensive theory on how faults are modeled in reliability testing, in addition to the fault models already presented in previous section. To encapsulate all possible fault models, we propose a characterization given at four different levels, going from more general notions to narrower properties of the model. The proposed fault model is structured as follows:

1) Gate-input & -output fault models: We model faults to occur within the Boolean components (gates), rather than on the wires since most of the malicious attacks result in a faulty operation on these components. Given the great variety of fault attacks, with very different effects and accuracy, we model faults in Boolean components as gate-output faults ($\mathcal{M}_{\mathcal{O}}$) where faults occur solely on the output wires, or as gate-input faults ($\mathcal{M}_{\mathcal{I}}$) where faults may occur on any input and the output. The gate-input fault model (analogous to the wire model used in IC testing) is a stronger model due to the finer granularity in the injection. The crucial difference between the two fault models is that with $\mathcal{M}_{\mathcal{I}}$, the attacker is given the ability to hit particular branches of fan-outs going into different components, directly affecting their inputs. On the other hand, in $\mathcal{M}_{\mathcal{O}}$, the gate driving the stem of such fan-out should be targeted, propagating the fault to all fan-out branches. Fig. 1 illustrates this concept.

Conventional fault injection mechanisms used against cryptographic devices, *e.g.*, clock or voltage glitches, are best modeled as gate-output faults. Furthermore, most of the scenarios consider the faults to happen solely at the output of the gates.

Nevertheless, more advanced techniques could be used to inject more precise faults, and thus, they are best modeled as gate-input faults. These techniques include Electromigration [34], EM glitch, or very accurate laser beams (laser cutter), which can affect input wires of a gate [35]. With highly accurate tools such as Scanning Electron Microscope (SEM) or Focused Ion Beam (FIB), the attacker might be able to inject intra-gate faults, getting a CMOS network to *stuck-at open/close* state. As reported in [36], [37], the most accurate way of modeling these faults is gate-input faults.

2) Uniform & Biased fault models: We represent a uniform fault model as $\mathcal{M}_{\mathcal{U}}$ and a biased fault model as $\mathcal{M}_{\mathcal{B}}$. In the case of cryptographic implementations, $\mathcal{M}_{\mathcal{U}}$ is less often utilized. The faults are maliciously injected by an attacker, who

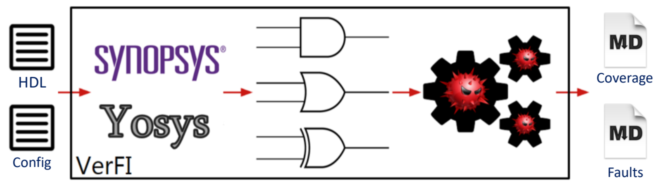


Fig. 2: VerFI's framework.

targets the (most) vulnerable parts of the design, *e.g.*, utilizing a localized laser beam.

3) Univariate & Multivariate models and number of faults: $\mathcal{M}_t$ represents a univariate fault model limiting the number of faults to t in a single clock cycle. $\mathcal{M}_t^*$ represents a multivariate fault model allowing at most t number of faults per cycle.

4) Type: The type of faults are typically SEU or SA0/1:

- **Single Event Upset (SEU)** is a non-permanent error that affects memory elements (registers) in a digital circuit, causing a bit to flip. Single Event Transient (SET) faults are similar errors that affect the combinational logic, which could turn into an SEU if propagated until a sequential logic (memory) unit. We refer to a non-propagated SET as an ineffective fault. For the sake of clarity, throughout the remaining of this work, we treat every bit-flip as a SEU. This type of fault is the one that most realistically models the effect of DFA [38], [39].
- **Stuck-at-0 (SA0)** and **Stuck-at-1 (SA1)** are the most common type of faults used in fault modeling theory. They represent a wire stuck to ground or V_{dd}, respectively. Contrary to this, in the cryptography settings, these kinds of faults are only used to model different fault attacks, like Safe-Error Attacks (SEAs) [40], or Statistical Ineffective Fault Attack (SIFA) [41].

IV. VerFI

In this section, we present the details of our Verification Tool for Fault Injection (VerFI[1]). We first introduce the framework and its functionalities. Then, we explain the fault simulator in detail, the relevance of the input test vectors, and the evaluation results. Finally, we explain how the tool can be used to address SIFA resistance and to evaluate infective countermeasures.

A. Framework

VerFI is an open-source tool designed to analyze the coverage of implementations of fault-injection countermeasures at the gate level, achieving an early assessment in the design flow. The tool receives an RTL-level design (Verilog or VHDL) or directly a Verilog netlist and generates the analysis result. Fig. 2 illustrates VerFI's framework. The execution of the program takes place in two steps, starting with a preprocessing of the RTL files to continue with the actual evaluation of the design.

[1]Our tool can be found in: https://github.com/vmarribas/VerMFi

a) Preprocessing: In the first stage, the tool automates the synthesis of the given RTL to create a netlist, which is parsed to produce the fault configuration files. These files allow the user to define a personalized fault injection. The tool can work with two different synthesizers, either Synopsys or Yosys [42], offering the option of a full open-source framework. Both Synopsys and Yosys are fully automated within VerFI's framework, which builds the corresponding *.tcl* file for each design, specifying the right constraints to make sure no components are optimized away by the synthesizer. This is crucial for fault countermeasures, which often use redundancy for protection. If needed, the integration of different synthesizers, *e.g.*, Cadence Genus, is straight forward as long as the synthesis result is a Verilog netlist.

b) Evaluation: The second part of the program receives one of the fault configuration files describing the injection, together with an optional file with the input test vectors provided by the user. The tool employs an event-driven technique to simulate the circuit and determines the values of all primary outputs (including the *status* signals, *e.g.*, a signal indicating the detection of a fault). During the experiment, the same fault simulations are performed for every input. The tool reports the coverage for every test vector and a final overall result with the total number of faults and the average coverage for the whole set of inputs. The tool reports all the non-detected faults per input test vector with the corresponding faulty output, as well as the ineffective faults.

c) VerFI's functionalities: The tool is designed to have high flexibility, allowing the users to define their fault model and evaluate their designs accordingly. The principal purpose of VerFI is to evaluate fault-injection countermeasures, but its functionality is not limited to this task. The user can mount simulated fault attacks over the actual netlist using the outputs generated by the fault simulations without the need to deploy the design on FPGA or ASIC. Nonetheless, the tool does not automate any attack; it only provides the means to perform them. Attacks like DFA, AFA [43], SEA [44] or SIFA, can make use of the results of VerFI to facilitate the key-recovery process. The user has full freedom to emulate different kinds of attacks and fault models.

B. VerFI's Fault Simulator

We describe the main parts of the fault simulator, including how faults are modeled, the fault injection mechanisms that allow the user to emulate any combination of the fault models presented in Sect. III-B, and how simulation is performed.

1) State-of-the-art EDA tools: There are several commercial tools already featuring powerful fault simulators, tailored to address IC testability. Apart from their expensive licenses, these tools require extensive training. Furthermore, the methodology they use (ATPG on the scan version of the circuit) does not provide the complete simulation of the design, which would be necessary for several countermeasures to decide whether a malicious fault is detected. Moreover, their reliability models are not suitable to analyze maliciously crafted faults.

Additionally, we aim to design a fully open-source tool that is accessible to everyone willing to have an initial assessment of their design. VerFI extends the faults covered in [30], accepting SEUs in every component of the circuit and allowing the user to specify when exactly the fault(s) should be injected, to fully comply with the fault models discussed in Sect. III-B.

2) Fault: As already mentioned before, we model errors in the circuit as gate-faults. We define a class `fault`, which has three attributes: active, cycle, and type. These three attributes respectively define whether the fault should be injected or not, at which cycle, and the type of the fault.

A `fault` object is appended to every cell (comb. or seq.), which is triggered when the cell is evaluated, modifying its output (or not) according to the fault type. We noticed that several fault attacks need a fault to be injected at the input of a non-linear function of the cipher. Hence, at the inputs of hierarchical sub-modules, we insert buffer gates, to each of which a `fault` is appended as well. These buffers enable the possibility of injecting faults at such input signals, thereby optionally activating the gate-input fault model (see Sect. III-B).

Since most of the countermeasures need to keep a strict hierarchy to avoid undesirable circuit optimizations when synthesized, it suffices to consider faults at inputs of sub-modules to deal with the gate-input fault model. The current version of the tool offers the possibility of injecting gate-input faults only at inputs of sub-modules, not at the inputs of every single cell. This is advantageous due to the lower number of faults that have to be considered. In future work, we will add an option to consider the gate-input fault model at every single cell of the circuit. This would lead to a significant increase in analysis complexity as many more faults should be simulated.

3) Injection Mechanism: The fault injection mechanism is the most delicate feature as it determines the fault model used for the evaluation. It has to be flexible to adapt to as many scenarios as possible, and allow the users to test their designs with the fault model of their choice. To this end, the execution of the tool is split into two steps, preprocessing and evaluation.

The user is provided with two fault configuration files in which the applicable options can be specified. Each file defines a different kind of fault injection mechanism, either *hierarchical* or *component-wise*. In both of them, the user defines the number of fault simulations N_s that the tool has to perform, and the number of faults N_f to inject per simulation. In the hierarchical injection, the user should additionally define the maximum number of faults per clock cycle N_t. For every module, the user should decide whether or not certain module should be faulted, whether to use the gate-input model, when to inject the fault, and which kind (SEU or SA0/1).

a) Hierarchical Injection: The first file drafts the hierarchical structure of the design with the respective module names, from which the tool stores all the cells from the modules enabled to be faulty and in which `cycle(s)`. A set of faults is created as $\boldsymbol{\mathcal{F}} = \{f_1, \ldots, f_{N_e}\}$, where f_i is the identifier of the ith cell enabled to be faulty, and N_e the number of such cells. The tool selects up to N_f random faults of $\boldsymbol{\mathcal{F}}$ per injection $\mathcal{I}$ and repeats this for up to N_s times. The user can perfectly delimit the areas where the faults should be injected, with equal probability. This delimitation allows either a uniform or biased fault model. Moreover, this also allows the evaluation of designs where just certain parts of the circuit are protected.

b) Component-wise Injection: The second file details every module's gates and (optionally) its inputs, allowing the user to precisely decide where to fault. This functionality is meant to test very particular faults, usually to be used after a more generic test with the hierarchical injection, ideal for launching an actual attack, or just debugging. Similarly to the previous modality, the tool collects and stores all faults into $\boldsymbol{\mathcal{F}}$. On the one hand, if $N_f = N_e$, the tool performs a single simulation activating all such faults. On the other hand, if $N_e > N_f$, then the tool simulates all possible combinations of N_f faults in the set $\boldsymbol{\mathcal{F}}$. If they exceed a user-defined threshold, the injection is done at random performing a max. of N_s fault simulations. Otherwise, N_s is ignored.

c) Word-wise Injection: The user can also define the faults to be injected in a word-wise fashion, *e.g.*, byte- or nibble-wise faults. The idea is to enable fault injection into the components driving all wires of the targeted bus. In this case, the tool can inject a random offset Δ into the targeted bus (SEU), or randomly set/reset some bits of the selected word (SA0/1). Depending on how the hierarchy of the circuit is designed, it would suffice to enable the `fault` flag of the blocks treating such buses, or, otherwise, a more careful selection can be made with the component-wise injection.

4) Simulation: The synthesized netlist is parsed to build a structural representation of the circuit, with wires, pins, and gates modeled as objects interconnected with each other. With this structural representation, we implement an *event-driven* simulator similar to the one described in algorithm 2 of [30]. To emulate the sequential behavior of the circuit and the concept of time provided by the clock, a simple scheduler in the form of a while-loop is implemented, where each iteration is a clock cycle. The simulation finishes when a particular *done* signal (as a primary output of the circuit) is set to '1/0', or when the loop exceeds the number of iterations requested by the user.

Faults are processed at the right iteration (cycle), activated or deactivated according to the specified clock cycle(s). The tool can simulate either round-based and serial implementations, or merely isolated pipelines.

Our goal is not to design the most efficient general purpose fault simulator, but to design a robust fault evaluation framework tailored to cryptographic implementations, which blends the concepts of fault diagnosis and fault evaluation in cryptographic algorithms. Thus, we start with a serial fault simulator, which simulates one by one the given fault injections, *i.e.*, one injection per simulation.

a) Performance optimizations: It is possible to significantly enhance the performance of the current version by implementing parallel fault simulation. The idea is to transform the current bit-wise operations to operate on 256-bit words by utilizing AVX2, where each bit of the word would correspond to a different fault simulation, allowing us to perform 256 fault simulations simultaneously (default to 64-bit words). This is already planned and devised as future work. Note that the extension of the current version of VerFI to support parallel fault simulation would only enhance it performance-wise. Its applicability and the supported models remain unchanged.

C. Input Test Vectors

The inputs to the circuit is another variable that influences the total space of possible fault simulations, together with the fault location and the time. The impact of different inputs on the coverage is considerable, since, for example, a single-bit fault could be classified as non-detected for a particular input, while for a different input the fault might not propagate and thus classified as ineffective. Hence, in order to have a meaningful result, the more inputs provided to the evaluation, the higher the confidence. Since it is impractical to evaluate all possible inputs exhaustively, VerFI gives the user two options to specify the inputs for the experiment:

- The user can specify the number of different inputs for simulation, and the tool would choose them at random.
- The user can provide a file with a list of input test vectors. For every input to simulate, the tool performs the same experiment, *i.e.*, the same fault injections.

D. Evaluation

The primary goal of designing VerFI is to verify countermeasures against fault attacks on cryptographic implementations. The tool accepts two forms of detection mechanisms: if the circuit detects a fault, it either triggers a *abort* signal or sets the full output to zero to avoid revealing any information about the faulty output (methodology proposed in [17]). Hence, the faults are classified into three different groups, based on whether the cipheterxt output (*ct*) is faulty ($\mathcal{O}_f$) or not ($\mathcal{O}$), and on the abort signal $\mathcal{A}$:

1) *Detected ($\mathcal{F}_D$):* the design detects the injected fault using any of the aforementioned detection mechanisms, *i.e.*, $ct = ?$ & $\mathcal{A} = 1$ or $ct = 0$.
2) *Non-Detected ($\mathcal{F}_N$):* the countermeasure does not detect the injected fault, and a faulty output is provided, *i.e.*, $ct = \mathcal{O}_f$ & $\mathcal{A} = 0$.
3) *Ineffective ($\mathcal{F}_I$):* the injected fault is not propagated to the output. In this case, the design does not detect the fault, but the output does not differ from the correct one, *i.e.*, $ct = \mathcal{O}$ & $\mathcal{A} = 0$.

The tool provides separated results for the coverage of effective faults and ineffective faults since the user could choose to protect against the first kind of faults but not the second. Thus, the detection coverage, given in Eqn. (1) does not take into account the ineffective faults. The subsequent section describes a verification procedure in case of SIFA attacks.

$$\mathcal{C} = \frac{\mathcal{F}_D}{\mathcal{F}_D + \mathcal{F}_N} \tag{1}$$

E. Statistical Ineffective Fault Attack

Statistical Ineffective Fault Attack (SIFA) [41] exploits ineffective faults, which lead to a non-uniform distribution on intermediate values enabling a key-recovery attack.

SIFA is fundamentally based on the fact that both effective (detected and non-detected) and ineffective faults appear. Hence, an algorithmic countermeasure to thwart SIFA could be an error *correction* mechanism, which prevents effective faults from appearing given a particular attacker model. Thereby, error correction can prevent the root cause of SIFA.

In order to check for full protection against SIFA, we can compute a coverage that depends on the effective and ineffective faults (see in SIFA paper, page 16). To provide full protection against SIFA, 100% of the injected faults would have to be ineffective. The fact of having 100% ineffective faults would break the requirement: "require that there is some dependency between the observation of an ineffective fault induction and the faulted intermediate value x" [41]. As a metric to address how possible is a SIFA attack, we use the *Fault Ineffectivity Rate (π)*, defined in [41] as follows:

$$\pi = \frac{\mathcal{F}_I}{\mathcal{F}_D + \mathcal{F}_N}. \tag{2}$$

The ineffectivity rate refers to the probability of an ineffective fault to occur. An ideal SIFA would have a high ineffectivity rate. Nevertheless, if $\pi \to \infty$ ($\mathcal{F}_D + \mathcal{F}_N = 0$) we can say that a SIFA attack is not possible.[2] Additionally, if $\pi = 0$ ($\mathcal{F}_I = 0$), it means SIFA is also prevented. In this case, the countermeasure is designed to ensure that every fault is effective [45].

Countermeasures at the protocol level are also a viable alternative. Although VerFI has been primarily designed to analyze algorithmic countermeasures, a protocol based on limiting the number of queries (N) can be evaluated with our tool by computing the likelihood of a successful SIFA attack given π and N.

F. Infective Countermeasures

During the simulation of any particular fault, VerFI treats the design as a black box, only processing inputs and outputs of the top entity to control the simulation and check the fault detection, respectively. Despite parsing and emulating the actual netlist, the tool does not know what the functionality of every component or wire is - it has no semantic information.

Regular infective countermeasures do not provide any *status* signal for fault detection; they merely propagate the error *infecting* the state of the computation. Hence, the attacker only sees garbage output. The problem of most infective countermeasures is that a rightful user is unaware if the output is correct or random junk. Our tool suffers from the same problem since it is difficult to decide if the output is an exploitable faulty output or merely random.

This issue can be resolved with little intervention by the user, who may annotate a signal in the netlist by just adding the comment //Check_bits:<signalname> to assert whether the infection was triggered or not. Thus, if the output is incorrect and the assertion flag was triggered, the tool knows that the fault was detected, and the output is just a random string.

[2]The designer should not insert any kind of detection mechanism that may give the attacker information between effective and ineffective faults.

V. Case Studies

In order to test the functionality of the tool, we evaluate multiple implementations secured against either combined or fault attacks from the literature. This includes implementations of CAPA [14], M&M [15], the original ParTI LED implementation [25], and different Impeccable Circuits [17] implementations of Midori [46]. For all of them, VerFI could find interesting results.

A. *CAPA and M&M*

CAPA is an algorithm-level combined countermeasure against SCA and fault-injection attacks in the *tile-probe-and-fault* model. It evolves from SPDZ [47], a Multi-Party Computation (MPC) protocol providing protection in the presence of both passive and active adversaries.

Additionally, we evaluate a design implementing infective countermeasures. As most of these countermeasures in the literature are broken, we chose the recently published M&M scheme. It evolves from CAPA, relaxing the attacker model to obtain a more practical scheme, where masking and information theoretic mac tags are combined. We perform the experiments on the original code for the full AES M&M V2 implementation ($k = 8$, $m = 1$), provided by the authors, and emulate a similar evaluation as performed in [15]. This experiment demonstrates the applicability of VerFI to analyze infective countermeasures.

B. *ParTI*

LED [48] is an AES-like block cipher. To achieve fault tolerance ParTI [25] applies an Error-Detection Code to each nibble of the state. More precisely, each nibble is encoded with eight bits allowing two completely separated computations to take place, which we call the data path, illustrated on the right, and the parity path, illustrated on the left in Fig. 3. The error check can detect any error that is local to only one path of the computation and any error of Hamming weight of up to 3 overall 8 bits of each codeword. This property comes from the underlying extended Hamming code applied in ParTI [25].

Both the parity path on the left and the data path on the right are split into a three-share TI to ensure first-order SCA resistance. This requires the cubic SBox to be split into quadratic functions G and F. Similarly, the parity path contains corresponding quadratic functions R and Q.

We had access to the original implementation of ParTI LED benchmarked in [25]. An initial evaluation with VerFI allowed us to spot an implementation mistake. The majority of the faults injected at the last two clock cycles of the computation were non-detected since the check resolution happened one or two cycles after the output was released. The checking mechanism presented in [25] needs a careful unsharing that takes two cycles to complete, as shown in Fig. 4. This mechanism does not entail a problem during the computation. However, if the output is issued (triggering a *done* signal) right after the computation of the last round (as it was the case), then the check for this round does not have enough time to finish. The authors have been notified and have provided an updated implementation.

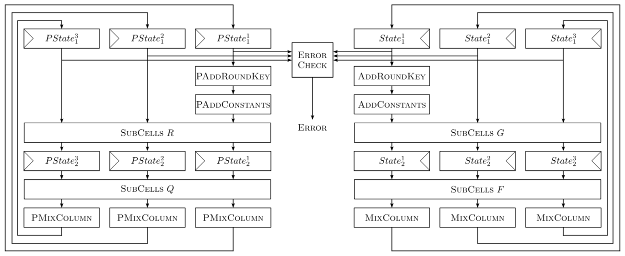


Fig. 3: Structure of LED protected with ParTI, taken from [25].

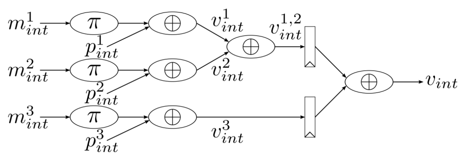


Fig. 4: Computation and unmasking of the error check vector in ParTI for first-order security, taken from [25].

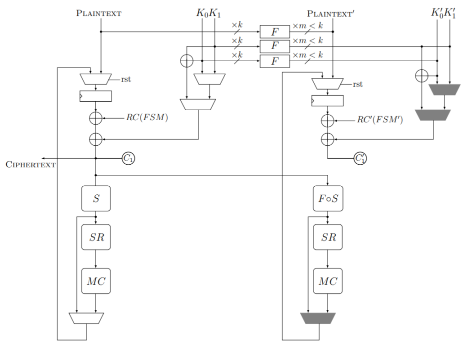


Fig. 5: Implementation of round-based Midori for a small degree of redundancy, taken from [17].

We conducted further analysis on the corrected implementation, finding several non-detected faults within the fault model mentioned above. We manage to bypass the checking mechanism with 2-bit faults, i.e., of Hamming weight 2. They are injected either at the same clock cycle (univariate $\mathcal{M}_{t=2}$) or in subsequent clock cycles (multivariate $\mathcal{M}_{t=1}^{*}$). The non-linearity of the SBox further propagates the faults, affecting more bits than the initially faulted, and producing a valid codeword that does not trigger the error detection. These findings confirm the suspicions raised in [17] since the given implementation of ParTI does not fulfill any of the properties proposed in that same work. With such non-detected faults, we can launch a DFA attack on the protected implementation. The details of the attack are given in Sect. VI.

C. *Impeccable Circuits*

A methodology that enables the secure and practical implementation of code-based CED schemes in the presence of fault propagation is suggested in [17].

In order to evaluate the soundness of the fault models considered in [17], we target Impeccable Circuits implementations of Midori [46] and a toy cipher. The authors have considered the gate-output fault model $\mathcal{M}_{\mathcal{O}}$ in their analyses, and designed the schemes accordingly. Utilizing VerFI, we subject the implementation to a stronger adversary model, *i.e.*, gate-input fault model $\mathcal{M}_{\mathcal{I}}$. Using this model, we were able to bypass the checking mechanism of several implementations. Fig. 5 presents the scheme of a round-based Midori protected with a small degree of redundancy. The implementation is carefully designed by using the independence property and taking care of the fault propagation, as well as placing the checks accordingly. In this case, the only block susceptible to propagate a fault is the SBox. No-fault injected in any of the gates inside the SBox would propagate further to create a valid codeword. The check is placed right before, so no previously injected fault would reach the inputs of the SBox. This investigation confirms the consistency of the claim of Impeccable Circuits based on its underlying $\mathcal{M}_{\mathcal{O}}$ model. Instead, by injecting a fault at the input of the SBox ($\mathcal{M}_{\mathcal{I}}$), the faults are further propagated to get a valid codeword, bypassing the checks, similarly to what happened in the case of the ParTI LED. This experiment practically illustrates how, by injecting faults at the gate inputs, we achieve a more powerful adversary.

VI. DFA on ParTI LED

We recall DFA and apply it to an implementation of LED protected by ParTI to achieve full key recovery.

A. Differential Fault Analysis

In a DFA [39] for a given plaintext and a specific fault model, the attacker can observe both the correct and faulty ciphertexts ct_c and ct_f respectively. Given the pair of both outputs, the attacker guesses parts of the key and reverses the cipher computation up to the point of fault injection to check whether the difference computed corresponds to the considered fault model. Thereby the attacker can reduce the total number of key candidates with each (ct_c, ct_f) pair until very few or even one unique key candidate remains. Due to the diffusion properties of secure block ciphers, the attack is usually limited to the penultimate or last round to avoid guessing the entire key which would be computationally infeasible. In the following, we limit ourselves to adversaries who can inject toggle (SEU) faults in a bounded number of individual bits.

B. Attack

We formulate criteria for faults to remain undetected by ParTI and subsequently describe two different DFA attacks leading to a full key recovery by exploiting undetected faults.

a) Undetected Faults: Due to the non-linearity of the SBox stages, a fault limited to few bits at their input may be propagated into a fault affecting more bits at the time of the check. Note that we stay within the gate-output fault model. Based on Fig. 3, faults injected into the AddRoundKey, can represent faults at the input of G. Similarly, the faults injected either at the G module or the registers placed between F and G would translate in faults at the input of F. The masking countermeasure within ParTI separates both data and parity paths into three shares. This neither helps nor hinders the DFA as the fault can just be injected into an arbitrary share. Since

TABLE I: All undetected Two Bit Faults in each Nibble of ParTI-LED: Δ_J denotes the input fault of operation J.

Δ_G	Δ_F	Δ_R	Δ_Q	# non-detected	# ineffective
2	0	0	2	2	14
4	0	0	2	2	14
0	8	8	0	8	8
1	0	0	4	8	8
8	0	0	4	8	8
1	0	0	8	8	8

the AddRoundKey blocks are only connected to the first share of G (resp. R), the adversary is limited to inject faults on this share in order to achieve a fault at the input of G. In Table I, we summarize all faults of Hamming weight two that lead to undetected faults at the time of the check. All presented fault locations have been found with our tool VerFI.

Once a faulty nibble passes the error check, there is no further possibility of detection in a later round as the EDC operates on each nibble individually. Hence, the described faults can be injected in an arbitrary round without detection.

b) Attacking Individual Nibbles: As LED is structurally very similar to AES, we can adapt the DFA demonstrated by Giraud [49] for key recovery. Since LED performs the MixColumns (MC) operation in the final round as well, we assume that the attacker computes equivalent keys k' corresponding to the actual cipher key via the relation

$$MC^{-1}(ct \oplus k) = MC^{-1}(ct) \oplus k', \text{ with } k = MC(k'),$$

where ct stands for ciphertext. For the sake of brevity, from here on, we refer to the elements before the last MC operation as ciphertexts while retaining the prime for clarity

$$ct'_c = MC^{-1}(ct_c) \text{ and } ct'_f = MC^{-1}(ct_f),$$

with ct_c and ct_f the correct and faulty ciphertexts, respectively.

We describe the attack corresponding to the third row of Table I. For each nibble i, the attacker injects a non-detected fault (Δ_F, Δ_R) into the input of F and R and determines the output. If the ciphertext is faulty, he guesses one nibble of the key and computes back to the time of injection, thereby obtaining a distinguisher along the data path:

$$\Delta_F \stackrel{?}{=} F^{-1}(ct'_{f,i} \oplus kg'_i) \oplus F^{-1}(ct'_{c,i} \oplus kg'_i).$$

Similarly, he can obtain a distinguisher along the parity path. However, both distinguishers cannot be used jointly, as the MC operation differs from the PMC operation, thereby affecting the reference of all variables denoted with a prime.

The attacker must obtain at least two pairs to reduce the number of key guesses kg for one nibble to a unique one. Note that obtaining all 16 nibbles only derives the last half of all LED key bits (for 128-bit key). The same attack can be performed one step earlier to recover the other half of the key. Thus, the attacker needs to obtain a minimum of 64 correct and faulty ciphertext pairs to achieve a full key recovery (for simplicity, we restrict our analysis to an attacker who does not use computational power to obtain remaining key parts).

c) Attacking Column-wise: In the following, we demonstrate how to reduce the number of pairs to perform the attack. Mounting the previous attack one round earlier within the final step, it affects an entire column of the ciphertext. Hence, the attacker can gain information about four key nibbles at once. For the sake of simplicity, we ignore the SHIFTROWS operation and refer to the nibbles in one column with the indices 0,...,3. The adversary computes back an entire round and some part of the non-linear operation:

$$yd_{0,1,2,3}(ct') = MC_4^{-1}(S_4^{-1}(ct'_{0,1,2,3}))$$
$$zd_{0,1,2,3}(ct') = F_4^{-1}(yd_{0,1,2,3}(ct')),$$

where MC_4 corresponds to mixing one column, and F_4 to the parallel application of the non-linear function to four nibbles.

Without loss of generality, we assume an injection of fault Δ_F into the first nibble, then the distinguisher is

$$(\Delta_F, 0, 0, 0) \overset{?}{=} zd_{0,1,2,3}(ct'_c) \oplus zd_{0,1,2,3}(ct'_f).$$

Further, it is even possible to find a distinguisher without knowledge of F_4 and exact knowledge of the injected fault by considering the equations

$$0 \overset{?}{\neq} yd_0(ct'_c) \oplus yd_0(ct'_f), \quad 0 \overset{?}{=} yd_i(ct'_c) \oplus yd_i(ct'_f),$$

for $i = 1, 2, 3$. This needs, on average, two pairs of (ct_c, ct_f) to determine a unique key guess for four nibbles. Hence, the adversary can extract each half of the key with only eight such pairs (resp. 16 pairs in total to extract the full key). We verified the feasibility of this attack by using VerFI to generate the 16 necessary pairs and determining the correct key accordingly.

VII. RESULTS

In this section, we present the analysis results reported by VerFI for several state-of-the-art fault-resistant hardware implementations, and the tool's performance. All implementations are provided by the respective authors of the analyzed works.

A. Evaluation

We report the practical coverage obtained using VerFI for implementations of ciphers protected by ParTI, CAPA, Impeccable Circuits, and M&M (cf. Tab. II). We analyzed all designs within the fault model proposed in their original work, except for the examples in rows 11 and 12, which we investigated in a stronger fault model ($\mathcal{M}_{\mathcal{I}}$ instead of $\mathcal{M}_{\mathcal{O}}$).

We simulated the CAPA designs within the gate-input fault model $\mathcal{M}_{\mathcal{I}}$, and without a restriction on the places to inject the fault ($\mathcal{M}_{\mathcal{U}}$). We inject 1, 8, and 24 faults in separate experiments (row 1, 2, and 3, respectively, Tab. II). In the first case, the observed coverage is very close to the theoretical claim (exhaustive search unfeasible). When a higher number of faults are injected, the coverage increases due to a higher probability of the detection mechanism being triggered. It is interesting to see how the injection of a single fault in this model is more harmful than the injection of multiple faults. We perform an additional experiment (row 4), this time within the model $\mathcal{M}_{\mathcal{I}} \cap \mathcal{M}_{\mathcal{B}}$, where we targeted exclusively one variable (ϵ) of the multiplication and its respective tag (τ^{ϵ}). This analysis entails a total of 16 locations, which we exhaustively check for combinations of a different number of faults, ranging from 3 to 10. The number of simulations increases from 560 to 12870. The result is the average of the eight experiments, achieving a coverage extremely close to the theoretical one. Since we target value and tag of a single variable, the resulting coverage is the probability of forging a correct tag.

Designs protected with ParTI are analyzed within the model $\mathcal{M}_{\mathcal{O}} \cap \mathcal{M}_{\mathcal{B}}$, with up to two-bit faults injected in each case, excluding the control datapath and the error check. ParTI fails to provide the promised full coverage. Nevertheless, the coverage is still comparably high, ranging between 0.994 and 0.9977. In the first experiment (row 5), the attack focuses on a single share of the first stage of the SBox, for both datapaths. In the second experiment (row 6), we additionally allowed the second stage of the SBox to be faulty in the subsequent clock cycle.

We evaluated several implementations protected with Impeccable Circuits, including a full Midori encryption function and a toy cipher. The experiments are conducted under the model $\mathcal{M}_{\mathcal{O}} \cap \mathcal{M}_{\mathcal{B}}$, restricting the number of faults per clock cycle to t (rows 7 - 14). The tool reports full coverage for all of them, as claimed by the respective authors. Note that this time, the faults could be injected at data processing, the control logic, and error check modules. Additionally, we investigated two implementations further within the $\mathcal{M}_{\mathcal{I}}$ model (rows 11 and 12). Although a high level of coverage is achieved, the designs do not provide full coverage against such an adversary model. This result does not contradict their security guarantees since protection under this model is never claimed.

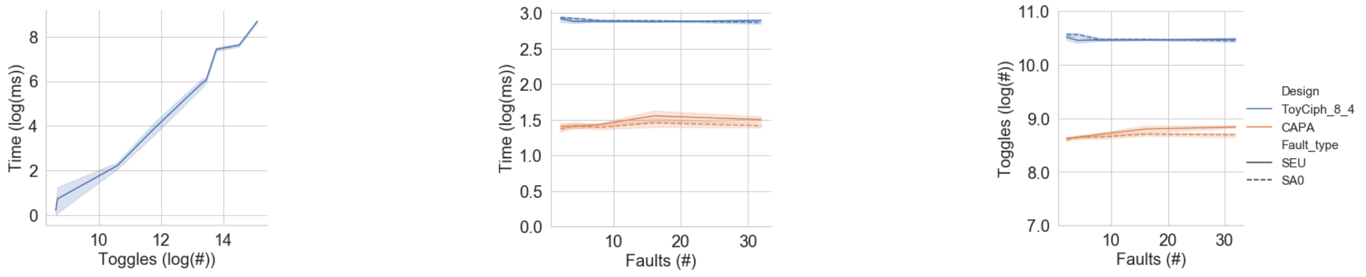
Finally, we present the experiments for M&M AES (rows 15, 16, 17 Tab. II). The first two experiments inject respectively one and eight faults, focused on the most sensitive modules, and the last two rounds of the computation. In the last experiment, we perform a component-wise injection, selecting similar components as in [15], and performing different injections of 1, 2, 4, 8, and 12 number of faults. The resulting coverage of this experiment is the average of the result of every injection. Since the security claims of this scheme include that alpha is never repeated for more than one computation, every experiment is performed with a single fault simulation for each input (# Simuls = 1) to comply with this requirement. We observe that all non-detected faults for the first experiment (row 15) occur when $\alpha = 0$, which happens with probability 2^{-8}. The coverage obtained with VerFI is exceptionally close to this probability. We obtain full coverage in the second experiment (row 16) since injecting multiple random faults increases the chance of one triggering the infection mechanism. The final experiment (row 17) also reports a coverage very close to the theoretical one. Even though this attacker is more potent since specific points of attack are chosen, it is still not better than guessing alpha.

B. Performance

VerFI is fully implemented in C++ for high performance. As it is based on the propagation of signal changes, the execution

TABLE II: Performance evaluation of different fault and combined countermeasures from the literature, including fault model, number of simulations, inputs and faults, injections cycles, and coverage.

Design	Fault model				# Simuls	# Inputs	# Faults	Cycle(s)	$\mathcal{C}_{theory}$	$\mathcal{C}_{tool}$
	$\mathcal{I}/\mathcal{O}$	$\mathcal{U}/\mathcal{B}$	$\mathcal{M}_t/\mathcal{M}_t^*$	Type				(Total)		
CAPA [14] 3 shares 8-bit multiplier (k=1)										
1 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{U}$	$\mathcal{M}_1$	SEU	100	500k	1	1,2,3(3)	0.99609	0.9973
2 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{U}$	$\mathcal{M}_8^*$	SEU	100	500k	8	1,2,3(3)	0.99609	0.9997
3 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{U}$	$\mathcal{M}_{24}^*$	SEU	100	500k	24	1,2,3(3)	0.99609	1
4 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_{3\rightarrow10}^*$	SEU	$560 \rightarrow 12870$	20	$3 \rightarrow 10$	1,2,3(3)	0.99609	0.99625
ParTI [25]										
5 Idem	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_2$	SEU	1k	4	2	93(98)	1	0.994
6 Idem	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_1^*$	SEU	10k	4	2	93.94(98)	1	0.9977
Impeccable Circuits [17] with code of length n, dimension k, and distance d as $[n,k,d]$										
7 Toy Cipher [5,4,2]	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_1$	SEU	120	10	1	14(16)	1	1
8 Toy Cipher [6,4,2]	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_1^*$	SEU	42k	10	2	>9(16)	1	1
9 Toy Cipher [7,4,3]	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_2^*$	SEU	100k	10	4	>9(16)	1	1
10 Toy Cipher [8,4,4]	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_3^*$	SEU	150k	10	6	>9(16)	1	1
11 Toy Cipher [5,4,2]	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_1$	SEU	20	3	1	13(16)	-	0.79
12 Toy Cipher [8,4,4]	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_2^*$	SEU	1.4M	3	2	>9(16)	-	0.999979
13 Midori Enc [8,4,4]	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_3$	SEU	200k	10	3	14(18)	1	1
14 Midori Enc [8,4,4]	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_3^*$	SEU	400k	10	6	>12(18)	1	1
AES M&M V2 [15] 2 shares (d=1), and single alpha (k=1)										
15 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_1$	SEU	1	10k	1	>199(254)	0.99609	0.9957
16 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_8^*$	SEU	1	10k	8	>199(254)	0.99609	1
17 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_{1,2,4,8,12}$	SEU	1	5k	1,2,4,8,12	225(254)	0.99609	0.9959



(a) Toggles vs. Time (b) Faults vs. Time (c) Faults vs. Toggles

Fig. 6: Evolution of VerFI's execution time with respect to the number of toggles and the number of faults per injection.

time is independent of a designs area and proportional to the number of toggles during simulation (Fig. 6a). Noticeably, the number of total simulations proportionally increases execution time, while a different number of faults injected per simulation (N_f) does not significantly alter the execution time for SEU and SA0 faults (Fig. 6b). Similarly, the number of toggles remains indifferent to the total number of faults (Fig. 6c).

Finally, we address the number of simulations for exhaustive checking. Given N_e as the total number of cells enabled to be faulty in a particular cycle and N_f as the number of faults per injection, the number of simulations to exhaustively check all combinations is given by the rapidly growing term $\binom{N_e}{N_f}$.

VIII. Conclusion

In this work, we have introduced the concept of Cryptographic Fault Diagnosis, a structured revision of fault diagnosis concepts known from reliability testing according to their relevance to cryptographic implementations, and a consolidated characterization of cryptographic fault models. To facilitate the practical application of our theory, we introduced VerFI, the first evaluation tool for cryptographic fault analysis working directly on the gate-level netlist of hardware designs. Our C++ tool is open-source, easy to use, and extensible. It is applicable to evaluate both detection- and infection-based countermeasures. VerFI can even asses an implementation's susceptibility to ineffective fault attacks, e.g., SIFA. It automates the evaluation of fault-protected implementations in the presence of a wide range of adversaries. We demonstrated the value and usability of VerFI by identifying implementation flaws in an instance of LED protected by ParTI [25] and performing a practical attack on a flawless implementation leading to a full-key recovery in the presence of adversaries covered by ParTI. Furthermore, we gave an accurate security indication for CAPA [14], designs protected with Impeccable Circuits [17], and M&M [15] according to their theoretical claims.

VerFI can be used jointly with state-of-the-art fault analysis tools [23] to first identify potentially-exploitable parts of a cipher, and then check the corresponding components in relevant clock cycles. Finally, we argue that our contribution is an essential tool for the verification of fault-protected implementations as it replaces manual changes of HDL-files with a fully-automated, high-speed evaluation workflow.

Acknowledgments

The work described in this paper has been supported in part by the NIST Research Grant 60NANB15D34, by the Research Council KU Leuven, C16/18/004, by the German Federal Ministry of Education and Research BMBF (grant nr. 16KIS0666 SysKit_HW), and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and the project 406956718 SuCCESS.

References

[1] G. T. Becker, J. Cooper, E. D. Mulder, G. Goodwill, J. Jaffe, G. Kenworthy, T. Kouzminov, A. Leiserson, M. Marson, P. Rohatgi, and S. Saab, "Test vector leakage assessment (TVLA) methodology in practice," International Cryptographic Module Conference, 2013, http://icmc-2013.org/wp/wp-content/uploads/2013/09/goodwillkenworthtestvector.pdf.

[2] O. Reparaz, "Detecting flawed masking schemes with leakage detection tests," in *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, ser. Lecture Notes in Computer Science, T. Peyrin, Ed., vol. 9783. Springer, 2016, pp. 204–222. [Online]. Available: https://doi.org/10.1007/978-3-662-52993-5_11

[3] R. Bloem, H. Groß, R. Iusupov, B. Könighofer, S. Mangard, and J. Winter, "Formal verification of masked hardware implementations in the presence of glitches," in *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, ser. Lecture Notes in Computer Science, J. B. Nielsen and V. Rijmen, Eds., vol. 10821. Springer, 2018, pp. 321–353. [Online]. Available: https://doi.org/10.1007/978-3-319-78375-8_11

[4] V. Arribas, S. Nikova, and V. Rijmen, "VerMI: Verification Tool for Masked Implementations," in *25th IEEE International Conference on Electronics, Circuits, and Systems*. Bordeaux,FR: IEEE, 2018, p. 4.

[5] D. Sijacic, J. Balasch, B. Yang, S. Ghosh, and I. Verbauwhede, "Towards efficient and automated side channel evaluations at design time," in *PROOFS 2018, 7th International Workshop on Security Proofs for Embedded Systems, colocated with CHES 2018, Amsterdam, The Netherlands, September 13, 2018*, ser. Kalpa Publications in Computing, L. Batina, U. Kühne, and N. Mentens, Eds., vol. 7. EasyChair, 2018, pp. 16–31. [Online]. Available: http://www.easychair.org/publications/paper/xPnF

[6] G. Barthe, S. Belaïd, P. Fouque, and B. Grégoire, "maskverif: a formal tool for analyzing software and hardware masked implementations," *IACR Cryptology ePrint Archive*, vol. 2018, p. 562, 2018. [Online]. Available: https://eprint.iacr.org/2018/562

[7] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[8] X. Guo, D. Mukhopadhyay, C. Jin, and R. Karri, "Security analysis of concurrent error detection against differential fault analysis," *J. Cryptographic Eng.*, vol. 5, no. 3, pp. 153–169, 2015.

[9] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "Error analysis and detection procedures for a hardware implementation of the advanced encryption standard," *IEEE Trans. Computers*, vol. 52, no. 4, pp. 492–505, 2003. [Online]. Available: https://doi.org/10.1109/TC.2003.1190590

[10] C. Carlet and S. Guilley, "Complementary dual codes for countermeasures to side-channel attacks," *Adv. in Math. of Comm.*, vol. 10, no. 1, pp. 131–150, 2016.

[11] S. Azzi, B. Barras, M. Christofi, and D. Vigilant, "Using linear codes as a fault countermeasure for nonlinear operations: application to AES and formal verification," *J. Cryptographic Engineering*, vol. 7, no. 1, pp. 75–85, 2017.

[12] R. Gennaro, A. Lysyanskaya, T. Malkin, S. Micali, and T. Rabin, "Algorithmic tamper-proof (ATP) security: Theoretical foundations for security against hardware tampering," in *TCC*, ser. Lecture Notes in Computer Science, vol. 2951. Springer, 2004, pp. 258–277.

[13] Y. Ishai, M. Prabhakaran, A. Sahai, and D. A. Wagner, "Private circuits II: keeping secrets in tamperable circuits," in *EUROCRYPT 2006*, ser. Lecture Notes in Computer Science, vol. 4004. Springer, 2006, pp. 308–327.

[14] O. Reparaz, L. De Meyer, B. Bilgin, V. Arribas, S. Nikova, V. Nikov, and N. P. Smart, "CAPA: the spirit of beaver against physical attacks," in *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Shacham and A. Boldyreva, Eds., vol. 10991. Springer, 2018, pp. 121–151. [Online]. Available: https://doi.org/10.1007/978-3-319-96884-1_5

[15] L. De Meyer, V. Arribas, S. Nikova, V. Nikov, and V. Rijmen, "M&M: Masks and Macs against physical attacks," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2019, no. 1, pp. 25–50, 2019. [Online]. Available: https://doi.org/10.13154/tches.v2019.i1.25-50

[16] O. Seker, A. Fernandez-Rubio, T. Eisenbarth, and R. Steinwandt, "Extending glitch-free multiparty protocols to resist fault injection attacks," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 3, pp. 394–430, 2018.

[17] A. Aghaie, A. Moradi, S. Rasoolzadeh, F. Schellenberg, and T. Schneider, "Impeccable circuits," *IACR Cryptology ePrint Archive*, vol. 2018, p. 203, 2018. [Online]. Available: http://eprint.iacr.org/2018/203

[18] E. Touloupis, J. A. Flint, V. A. Chouliaras, and D. D. Ward, "Study of the effects of seu-induced faults on a pipeline protected microprocessor," *IEEE Trans. Computers*, vol. 56, no. 12, pp. 1585–1596, 2007. [Online]. Available: https://doi.org/10.1109/TC.2007.70766

[19] J. L. A. Hughes and E. J. McCluskey, "Multiple stuck-at fault coverage of single stuck-at fault test sets," in *Proceedings International Test Conference 1986, Washington, D.C., USA, September 1986*. IEEE Computer Society, 1986, pp. 368–374.

[20] P. Camurati, P. Prinetto, M. Rebaudengo, and M. S. Reorda, "Improved techniques for multiple stuck-at fault analysis using single stuck-at fault test sets," in *[Proceedings] 1992 IEEE International Symposium on Circuits and Systems*. IEEE. [Online]. Available: https://doi.org/10.1109%2Fiscas.1992.229933

[21] F. Zhang, S. Guo, X. Zhao, T. Wang, J. Yang, F. Standaert, and D. Gu, "A framework for the analysis and evaluation of algebraic fault attacks on lightweight block ciphers," *IEEE Trans. Information Forensics and Security*, vol. 11, no. 5, pp. 1039–1054, 2016. [Online]. Available: https://doi.org/10.1109/TIFS.2016.2516905

[22] P. Khanna, C. Rebeiro, and A. Hazra, "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*. ACM, 2017, pp. 8:1–8:6. [Online]. Available: https://doi.org/10.1145/3061639.3062340

[23] S. Saha, D. Mukhopadhyay, and P. Dasgupta, "Expfault: An automated framework for exploitable fault characterization in block ciphers," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 2, pp. 242–276, 2018. [Online]. Available: https://doi.org/10.13154/tches.v2018.i2.242-276

[24] C. Beierle, G. Leander, A. Moradi, and S. Rasoolzadeh, "CRAFT: lightweight tweakable block cipher with efficient protection against DFA attacks," *IACR Trans. Symmetric Cryptol.*, vol. 2019, no. 1, pp. 5–45, 2019.

[25] T. Schneider, A. Moradi, and T. Güneysu, "Parti - towards combined hardware countermeasures against side-channel and fault-injection attacks," in *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, M. Robshaw and J. Katz, Eds., vol. 9815. Springer, 2016, pp. 302–332. [Online]. Available: https://doi.org/10.1007/978-3-662-53008-5_11

[26] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold implementations against side-channel attacks and glitches," in *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, ser. Lecture Notes in Computer Science, P. Ning, S. Qing, and N. Li, Eds., vol. 4307. Springer, 2006, pp. 529–545. [Online]. Available: https://doi.org/10.1007/11935308_38

[27] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Incorporated, 2013.

[28] T. M. Niermann, W. Cheng, and J. H. Patel, "PROOFS: a fast, memory-efficient sequential circuit fault simulator," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 11, no. 2, pp. 198–207, 1992. [Online]. Available: https://doi.org/10.1109/43.124398

[29] H. K. Lee and D. S. Ha, "HOPE: an efficient parallel fault simulator for synchronous sequential circuits," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 15, no. 9, pp. 1048–1058, 1996. [Online]. Available: https://doi.org/10.1109/43.536711

[30] A. Bosio and G. D. Natale, "LIFTING: A flexible open-source fault simulator," in *17th IEEE Asian Test Symposium, ATS 2008, Sapporo, Japan, November 24-27, 2008*. IEEE Computer Society, 2008, pp. 35–40. [Online]. Available: https://doi.org/10.1109/ATS.2008.17

[31] M. Otto, "Fault attacks and countermeasures," Ph.D. dissertation, University of Paderborn, Germany, 2005. [Online]. Available: http://ubdata.uni-paderborn.de/ediss/17/2004/otto/disserta.pdf

[32] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, Feb 2006.

[33] D. Karaklajić, J.-M. Schmidt, and I. Verbauwhede, "Hardware designer's guide to fault attacks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 12, pp. 2295–2306, Dec 2013.

[34] J. R. Black, "Electromigration—a brief survey and some recent results," *IEEE Transactions on Electron Devices*, vol. 16, no. 4, pp. 338–347, April 1969.

[35] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Characterizing the impact of intermittent hardware faults on programs," *IEEE Trans. Reliability*, vol. 64, no. 1, pp. 297–310, 2015. [Online]. Available: https://doi.org/10.1109/TR.2014.2363152

[36] X. Fan, W. R. Moore, C. Hora, and G. Gronthoud, "Stuck-open fault diagnosis with stuck-at model," in *10th European Test Symposium, ETS 2005, Tallinn, Estonia, May 22-25, 2005*. IEEE Computer Society, 2005, pp. 182–187. [Online]. Available: https://doi.org/10.1109/ETS.2005.35

[37] ——, "Extending gate-level diagnosis tools to CMOS intra-gate faults," *IET Computers & Digital Techniques*, vol. 1, no. 6, pp. 685–693, 2007. [Online]. Available: https://doi.org/10.1049/iet-cdt:20060206

[38] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults (extended abstract)," in *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, ser. Lecture Notes in Computer Science, W. Fumy, Ed., vol. 1233. Springer, 1997, pp. 37–51. [Online]. Available: https://doi.org/10.1007/3-540-69053-0_4

[39] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, ser. Lecture Notes in Computer Science, B. S. K. Jr., Ed., vol. 1294. Springer, 1997, pp. 513–525. [Online]. Available: https://doi.org/10.1007/BFb0052259

[40] S. Yen and M. Joye, "Checking before output may not be enough against fault-based cryptanalysis," *IEEE Trans. Computers*, vol. 49, no. 9, pp. 967–970, 2000.

[41] C. Dobraunig, M. Eichlseder, T. Korak, S. Mangard, F. Mendel, and R. Primas, "SIFA: exploiting ineffective fault inductions on symmetric cryptography," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 3, pp. 547–572, 2018. [Online]. Available: https://doi.org/10.13154/tches.v2018.i3.547-572

[42] C. Wolf and J. Glaser, "Yosys - a free verilog synthesis suite." *Proceedings of Austrochip*, 2013.

[43] L. R. Knudsen and C. V. Miolane, "Counting equations in algebraic attacks on block ciphers," *Int. J. Inf. Sec.*, vol. 9, no. 2, pp. 127–135, 2010. [Online]. Available: https://doi.org/10.1007/s10207-009-0099-9

[44] S. Yen, S. Kim, S. Lim, and S. Moon, "A countermeasure against one physical cryptanalysis may benefit another attack," in *Information Security and Cryptology - ICISC 2001, 4th International Conference Seoul, Korea, December 6-7, 2001, Proceedings*, ser. Lecture Notes in Computer Science, K. Kim, Ed., vol. 2288. Springer, 2001, pp. 414–427. [Online]. Available: https://doi.org/10.1007/3-540-45861-1_31

[45] J. Daemen, C. Dobraunig, M. Eichlseder, H. Groß, F. Mendel, and R. Primas, "Protecting against statistical ineffective fault attacks," *IACR Cryptology ePrint Archive*, vol. 2019, p. 536, 2019.

[46] S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita, and F. Regazzoni, "Midori: A block cipher for low energy," in *ASIACRYPT 2015*, ser. Lecture Notes in Computer Science, vol. 9453. Springer, 2015, pp. 411–436.

[47] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, 2012, pp. 643–662. [Online]. Available: https://doi.org/10.1007/978-3-642-32009-5_38

[48] J. Guo, T. Peyrin, A. Poschmann, and M. J. B. Robshaw, "The LED block cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, ser. Lecture Notes in Computer Science, B. Preneel and T. Takagi, Eds., vol. 6917. Springer, 2011, pp. 326–341. [Online]. Available: https://doi.org/10.1007/978-3-642-23951-9_22

[49] C. Giraud, "DFA on AES," in *Advanced Encryption Standard - AES, 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers*, ser. Lecture Notes in Computer Science, H. Dobbertin, V. Rijmen, and A. Sowa, Eds., vol. 3373. Springer, 2004, pp. 27–41. [Online]. Available: https://doi.org/10.1007/11506447_4