FIVER – Robust Verification of Countermeasures against Fault Injections

Jan Richter-Brockmann[1], Aein Rezaei Shahmirzadi[1], Pascal Sasdrich[1], Amir Moradi[1] and Tim Güneysu[1,2]

[1] Ruhr University Bochum, Horst Görtz Institute for IT Security, Bochum, Germany
[2] DFKI, Bremen, Germany
firstname.lastname@rub.de

Abstract. Fault Injection Analysis is seen as a powerful attack against implementations of cryptographic algorithms. Over the last two decades, researchers proposed a plethora of countermeasures to secure such implementations. However, the design process and implementation are still error-prone, complex, and manual tasks which require long-standing experience in hardware design and physical security. Moreover, the validation of the claimed security is often only done by empirical testing in a very late stage of the design process. To prevent such empirical testing strategies, approaches based on formal verification are applied instead providing the designer early feedback.

In this work, we present a fault verification framework to validate the security of countermeasures against fault-injection attacks designed for ICs. The verification framework works on netlist-level, parses the given digital circuit into a model based on Binary Decision Diagrams, and performs symbolic fault injections. This verification approach constitutes a novel strategy to evaluate protected hardware designs against fault injections offering new opportunities as performing full analyses under a given fault models.

Eventually, we apply the proposed verification framework to real-world implementations of well-established countermeasures against fault-injection attacks. Here, we consider protected designs of the lightweight ciphers CRAFT and LED-64 as well as AES. Due to several optimization strategies, our tool is able to perform more than 90 million fault injections in a single-round CRAFT design and evaluate the security in under 50 min while the symbolic simulation approach considers all 2^{128} primary inputs.

Keywords: FIA · Fault Verification · Formal Verification · BDD · Symbolic Simulation

1 Introduction

In 1997, Biham and Shamir proposed Differential Fault Analysis (DFA) as powerful attack against implementations of cryptographic algorithms [BS97]. In the aftermath, this seminal work sparked a new branch of research dealing with cryptographic Fault Injection Analysis (FIA). Over the last two decades, researchers mainly pursued three sub-topics, covering the development of new *analysis techniques*, the enhancement of *fault injection methods* in hardware devices, and the design of effective *countermeasures*.

With respect to analysis techniques, the DFA from Biham and Shamir [BS97] continuously was improved and applied to several symmetric block ciphers [BS03, Gir04, AMT13]. Additionally, many different attacks have been proposed, ranging from Ineffective Fault Attacks (IFAs) [Cla07], over Statistical Fault Attacks (SFAs) [FJLT13], to the recently proposed Statistical Ineffective Fault Analysis (SIFA) [DEG+18].

The second research direction addresses the physical disturbance of an ongoing operation in a target cryptographic algorithm or device. Therefore, several methods have been presented over the last years, including clock glitches [DEG+18], voltage glitches [ZDCT13], electromagnetic pulses [DDRT12, DLM19], focused laser beams [SA02], and several others [MTOL12, O'F20, CML+11, HS13, ABC+17].

Ultimately, researchers from academia and industry proposed a plethora of countermeasures to secure cryptographic operations against FIA. More precisely, modern countermeasures leverage redundancy in time, area, or information and can be classified as detection-based, correction-based, and infection-based techniques While detection-based countermeasures were intensively investigated in [AMR+20], using linear error codes, originally known from coding theory, to protect symmetric block ciphers, this approach was extended in [SRM20] such that linear error codes also were deployed to correct occurring faults. The last class – infection-based countermeasures – was investigated in [GST12] and infects the state of a cryptographic algorithm with random bits in case a fault occurred intending to generate useless outputs for an attacker.

However, despite extensive theoretical research on efficient and effective countermeasure, in particular the process of designing and implementing such methods in practice is still an error-prone, complex, and manual task, requiring longstanding experience and expertise in hardware design and physical security. Furthermore, the correctness and security of implemented designs and countermeasures are predominantly evaluated through empirical testing of prototypes or final products, making it difficult to correct or adjust design deficiencies and security flaws. To counteract this issue, formal verification can support the designer during the design process and provide an early indication of deficiencies and flaws.

As a consequence, the approach of empirical testing is replaced by security proofs which, however, require an appropriate definition of the adversary model and an abstraction of fault injection methods. Recently, the authors of [RBSG21] proposed a new and generic fault model which allows to precisely define and describe an attacker. While the work from Arribas et *al.* [AWMN20] already presents a fault verification tool called VerFI, directly working on a gate-level netlist of a cryptographic circuit, it works with a limited set of fault models (i.e., bit-flip and stuck-at) and exposes some open challenges, particularly with respect to the reliability of the reported results. More precisely, as the tool is simulation-based, the user has to select dedicated test vectors, which can lead to undetected corner-cases and false-positive results. To this end, in this work, we close this gap by proposing a verification approach that inherently prevents misleading evaluation results and reports.

Contributions. We propose a formal verification approach and corresponding tool for countermeasures against FIA for cryptographic algorithms implemented on Integrated Circuits (ICs). Hence, similar to VerFI [AWMN20], our approach works on a given gate-level netlist serving as starting point to create a model of the underlying digital logic circuit. However, instead of relying on empirical testing methods, we present a formal verification approach that is based on Binary Decision Diagrams (BDDs). This data structure inherently provides the possibility to observe the output of a Boolean function considering all combinations of the input variables. Hence, we avoid false-positives that could be created by selecting inauspicious test vectors. Additionally, we propose a symbolic fault injection approach allowing to cover all possible fault events that can occur in digital logic circuit under a given fault model while avoiding to detect any undiscovered corner cases that may lead to successful fault injection attacks.

Furthermore, instead of fixing the applied fault model to a predefined subset as in VerFI, we incorporate the generic fault model from [RBSG21]. This permits a precise definition and description of the adversary while analyzing the countermeasure's claims.

In order to achieve reasonable performance with respect to the evaluation time and circuit size, we present different kinds of optimization strategies. On the one hand, these strategies address the reduction of the complexity of the number of fault combinations that can occur in a digital logic circuit. On the other hand, we propose several approaches to increase the performance of our fault verification tool. The tool is publicly available and can be accessed via https://github.com/Chair-for-Security-Engineering/FIVER.

Outline. In Section 2 we briefly summarize notations and BDDs, introduce our circuit model, and provide background to FIA. Based on these preliminaries, we present our fault verification concept in detail in Section 3. Afterwards, in Section 4, we introduce our fault verification tool FIVER by providing more details about the applied BDD library, about the tool flow and optimization strategies. In Section 5, we present practical evaluations and experiments applying our tool to common ciphers equipped with detection-based and correction-based countermeasures. Eventually, we conclude our work in Section 6.

2 Background

In this section, we briefly state our notations used throughout this work. Afterwards, we provide essential background of BDDs and introduce our circuit model. We conclude this section by discussing fault injection analysis and the fault verification tool VerFI.

2.1 Notation

We denote function by using sans-serif fonts, i.e., f. While we express single-bit values by x_i, the corresponding multi-bit variable is denoted by $\mathbf{x}$. Upper-case Greek letters are used to denote sets, e.g., Λ.

2.2 Binary Decision Diagrams

BDDs have been introduced by Akers [Ake78] and refined by Bryant [Bry86] (introducing variable ordering), providing a compact and concise data structure to represent Boolean functions in discrete mathematics and computer science. These days, many applications in Electronic Design Automation (EDA) and computer-aided IC design and verification rely on (reduced and ordered) BDDs[1]

Given any Boolean function $\mathsf{f} : \mathbb{F}_2^i \mapsto \mathbb{F}_2$, BDDs provide a concise and canonical (for a given variable ordering) graph-based representation, with a single root node and at most two terminal nodes (leaves) $\{\mathbf{0}, \mathbf{1}\}$. The formal definition of BDDs is given as follows, divided into a purely *syntactical* description and a *semantical* interpretation.

2.2.1 Syntactical Definition of BDDs

Each BDD can be represented as a finite Direct Acyclic Graph (DAG) according to the following syntactical definition.

Definition 1 (BDD Syntax)**.** *A* Reduced Ordered Binaray Decision Diagram *is a pair $(\pi, \mathcal{D})$, with π denoting the variable ordering and $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ describing a finite DAG with vertices $\mathcal{V}$, edges $\mathcal{E}$, and the following properties:*

(1) Given a single root note, each node $v \in \mathcal{V}$ is either a non-terminal node or one of the terminal nodes $\{\mathbf{0}, \mathbf{1}\}$.

[1] In this work we deal with Reduced and Ordered BDD (ROBDD), to which we refer by BDD for the sake of simplicity.

(2) Each non-terminal node v is labeled with a variable in $\mathbf{X}$, with $|\mathbf{X}| = n$, denoted as $\mathsf{var}(v)$ *and has exactly two child nodes in $\mathcal{V}$, denoted as* $\mathsf{then}(v)$ *and* $\mathsf{else}(v)$*.*

(3) For each path from root to a terminal node, the variables in $\mathbf{X}$ are encountered at most once and in the order defined by the variable ordering π. More specifically, the ordering π is a bijection $\pi : \{1, 2, \ldots, n\} \mapsto \mathbf{X}$.

(4) For each node $v \in \mathcal{V}$, it holds that $\mathsf{then}(v) \neq \mathsf{else}(v)$*.*

(5) There are no duplicate nodes, i.e., for each pair of nodes $\{v, v'\} \in \mathcal{V}$, it holds that either $\mathsf{var}(v) \neq \mathsf{var}(v')$*,* $\mathsf{then}(v) \neq \mathsf{then}(v')$*, or* $\mathsf{else}(v) \neq \mathsf{else}(v')$*.*

2.2.2 Semantical Definition of BDDs

Each BDD, with root node $v \in \mathcal{V}$, recursively represents a Boolean function $\mathsf{f} : \mathbb{F}_2^i \mapsto \mathbb{F}_2$ using the *Shannon decomposition* of f according to the following definition.

Definition 2 (BDD Semantics)**.** *A Boolean function f over $\mathbf{X}$ is defined recursively by a BDD, carried out at each node according to the following rules:*

(1) If v is the terminal node $\mathbf{1}$, then $\mathsf{f}_v|_x = 1$, otherwise, if v is the terminal node $\mathbf{0}$, then $\mathsf{f}_v|_x = 0$.

(2) For each non-terminal node v with $\mathsf{var}(v) = x_i$*, f_v is defined according to the* Shannon decomposition $\mathsf{f} = x_i \cdot \mathsf{f}_{\mathsf{then}(v)}|_{x_i=1} + \overline{x_i} \cdot \mathsf{f}_{\mathsf{then}(v)}|_{x_i=0}$.

2.2.3 Boolean Operations over BDDs

Given the syntactical and semantical definitions of BDDs, any arbitrary Boolean operation $\circ$ over two Boolean functions f_{v_1} and f_{v_2}, given as BDDs with root nodes v_1 and v_2, can be defined recursively, such that:

$$\begin{aligned}\mathsf{f} &= x_i \cdot \mathsf{f}|_{x_i=1} + \overline{x_i} \cdot \mathsf{f}|_{x_i=0} \\ &= x_i \cdot (\mathsf{f}_{v_1} \circ \mathsf{f}_{v_2})|_{x_i=1} + \overline{x_i} \cdot (\mathsf{f}_{v_1} \circ \mathsf{f}_{v_2})|_{x_i=0} \\ &= x_i \cdot (\mathsf{f}_{v_1}|_{x_i=1} \circ \mathsf{f}_{v_2}|_{x_i=1}) + \overline{x_i} \cdot (\mathsf{f}_{v_1}|_{x_i=0} \circ \mathsf{f}_{v_2}|_{x_i=0})\end{aligned}$$

2.3 Circuit Model

As our goal is to verify hardware implementations protected against fault injection attacks, we introduce an abstract model to describe the underlying circuits. To this end, we construct such a model based on a gate-level netlist describing a digital logic circuit $\mathbf{C}$. Naturally, it is assumed that the circuit $\mathbf{C}$ realizes an arbitrary (vectorial) Boolean function $\mathsf{f} : \mathbb{F}_2^i \to \mathbb{F}_2^o$ with input size $i \geq 1$ and output size $o \geq 1$. At the lowest level, we decompose the circuit $\mathbf{C}$ into atomic components, called *gates*, which can be further divided into purely *combinational gates* and *sequential gates*.

Definition 3 (Combinational Gate)**.** *A* combinational gate *g_c is a physical component in a digital logic circuit that evaluates its output as a pure (Boolean) function of the present inputs only (without any dependency on the history of inputs).*

In this work we limit the set of Boolean functions implemented as combinational gates by $\mathcal{G}_c = \{\mathsf{not}, \mathsf{and}, \mathsf{nand}, \mathsf{or}, \mathsf{nor}, \mathsf{xor}, \mathsf{xnor}\}$. We further distinguish between gates with fan-in size one (unary gates) and size two (binary gates) leading to the sets $\mathcal{G}_u = \{\mathsf{not}\}$ and $\mathcal{G}_b = \{\mathsf{and}, \mathsf{nand}, \mathsf{or}, \mathsf{nor}, \mathsf{xor}, \mathsf{xnor}\}$ such that $\mathcal{G}_c = \mathcal{G}_u \cup \mathcal{G}_b$.

Definition 4 (Sequential Gate)**.** *A* sequential gate *$g_s \in \mathcal{G}_s$ is a physical, clock-synchronized, memory component in a digital logic circuit for which the output depends not only on the present inputs but also on the history of previous inputs.*

Sequential (memory) gates store a single Boolean variable $x \in \mathbb{F}_2$ while we model them as clock-dependent synchronization points. We suppose that a sequential gate has only one output. Some standard libraries make use of flip-flops with two complementary outputs. In such cases, the gate is further decomposed to a sequential gate followed by a combinatorial gate not. Analogously to the definition of combinational gates, we define the set $\mathcal{G}_s = \{\mathsf{reg}\}$. Given that, we unite all valid gates in one set $\mathcal{G} = \mathcal{G}_c \cup \mathcal{G}_s$ to properly model a digital logic circuit $\mathbf{C}$ as defined in Definition 5.

Definition 5 (Circuit Representation)**.** *A digital logic circuit* $\mathbf{C}$ *is modeled by a DAG formally described by* $\mathcal{D} = \{\mathcal{V}, \mathcal{E}\}$*, with* $\mathcal{V}$ *the set of vertices and* $\mathcal{E}$ *the set of edges. A single vertex* $v \in \mathcal{V}$ *represents a combinational or sequential gate* $g \in \mathcal{G}$ *and a single edge* $e \in \mathcal{E}$ *represents a wire connecting two vertices* $v_1, v_2 \in \mathcal{V}$ *and carrying a digital signal, modeled as an element from the finite field* $\mathbb{F}_2$*.*

Based on this definition, the model cannot handle circuits with loops, which is a common practice in digital logic circuit designs[2]. Hence, to allow our model to handle such cases, the circuit needs to be unrolled before it is translated to a DAG. By unrolling we describe the process of removing any cyclic loops and replacing them by acyclic structures. For example, a cryptographic round-based implementation of an arbitrary block-cipher can be unrolled by instantiating the round logic r-times connecting them separated by registers where r denotes the number of rounds.

2.4 Fault Injection Analysis

Physical Fault Injection Techniques. In general, Fault Attacks (FAs), as a particular branch of physical attacks, aim at disturbing the regular execution of a physical device by forcing it to operate under non-specified conditions. For this, common approaches attempt to maliciously induce faults during operation in order to violate the timing requirements of a digital logic circuit, where the most prominent techniques use *clock glitches* or *voltage glitches* [SGD08, ADN+10, GSD+08]. More specifically in the former approach, the period of the clock is tightened for one (or a couple of) cycles while in the latter, the power supply of the target device is altered for a short moment to disrupt its normal execution.

Besides timing violations, it has been shown that varying the ambient operating conditions, such as the temperature, can also lead to a faulty behavior [GA03, HS13]. However, while all aforementioned techniques are non-invasive as they do not require a modification of the targeted device, semi-invasive attacks allow injecting localized faults with high precision. Particularly, decapsulation of the chip package enables an attacker to induce faults using electromagnetic pulses [ORJ+13, OGT+14, BKH+19] or an intense light source like laser beams [SA02, SFG+16].

Consolidated Fault Models. Fault models are widely used as an abstraction of an undesired physical event, enabling hardware designers to predict the circuit behavior in the presence of the faults. For this, *stuck-at* and *toggle* (bit-flip) fault models are commonly adduced and consolidated models, often considered in IC test and verification processes [RU96], but also in FIA.

For the *stuck-at*-0 (resp. *stuck-at*-1) model, it is assumed that an individual signal, e.g., the output of a binary logic gate, is tied to logical **0** (resp. **1**). However, while the actual value of the signal is not relevant in the stuck-at models, the toggle model relies on the original value. More precisely, for toggle model, the signal is turned to logical **0** if the actual value is **1**, and vice versa.

[2] By this, we do not refer to combinatorial loops which are not considered as a usual synchronous design architecture.

Recently, the authors of [RBSG21] proposed a new consolidated fault model which is described by a function $\zeta(n, t, l)$. Here, n defines the total number of fault events that can occur at the same time in one logic stage. The parameter t describes the fault type. More precisely, the authors proposed a set of fault mappings τ_j which then describe the actual behavior of the fault. To model a fault injection in a target gate, the associated Boolean function is replaced by another function which is defined in τ_j. In their work, they specify common fault models like *stuck-at* and *bit-flip* but also more advanced mappings that describe laser fault injections in the 15 nm Open-Cell Library. However, the last parameter l defines the valid fault locations in a given digital logic circuit. Particularly, the fault location limits fault injections to combinational (c), sequential (s) or both gate types (cs). In the remainder of this work, we also follow this notation and define an attacker by $\zeta(n, t, l)$.

Fault Analysis Techniques and Countermeasures. In the context of cryptographic fault analysis, several techniques have been introduced to recover a secret key after successful fault injection. Examples include DFA [BS97], SFA [FJLT13], Differential Fault Intensity Analysis (DFIA) [GYTS14], IFA [Cla07], and SIFA [DEK$^+$18].

Depending on the situation and the scenario, various factors, such as access to the faulty results, the precision of the fault injection, or the underlying cryptographic algorithm affect the final choice of the analysis technique. However, due to the efficiency of such powerful attacks, the research community has also dedicated a considerable body of research to propose methodologies counteracting fault injections. For this, all approaches and countermeasures commonly rely on redundancy in terms of area, time, information, or any combination of them.

For instance, an encryption function can be instantiated twice (or multiple times) to form a basic detection scheme (based on spatial redundancy) allowing to check the consistency of the outputs through a simple comparison [MSY06]. Another trivial way to detect the presence of a fault is *recomputation* (as temporal redundancy), i.e., the construction recomputes the output multiple times using the same dedicated function and compares the results [MSY06]. In [AMR$^+$20], a code-based approach based on Concurrent Error Detection (CED) schemes, i.e., information redundancy, has been proposed, where fault propagation in hardware implementations has been taken into account. More precisely, the authors guarantee the detection of any induced fault in any location of the design, including data path, Finite State Machine (FSM), control signals, and consistency check modules at any clock cycle. However, since the proposed technique is vulnerable against advanced attacks such as IFA and SIFA, the detection facility of [AMR$^+$20] was extended to fault correction in [SRM20] to also protect implementations against IFA and SIFA.

Additionally, the authors proposed important properties and guidelines to design resilient hardware countermeasures against fault injection attacks. The most significant criteria, called *Independence Property*, was introduced in [AMR$^+$20] and demands that a digital circuit is separated into independent parts such that each computes exactly one output bit. Then, a checkpoint is placed at the output of each separate part ensuring to detect or correct any fault within the capabilities of the underlying countermeasure. To this end, introducing a checkpoint after each non-linear function ensures to stop fault propagation as early as possible and prevents unnecessary complexity when designing large circuits fulfilling the independence property over several non-linear functions. Hence, in the context of designing countermeasures for block ciphers, a checkpoint should be introduced after each substitution layer which ensures to detect occurring faults in each round.

A couple of more techniques have been proposed to protect against SIFA [BKHL20, SJR$^+$20, RAD20]. A combined countermeasure against Side-Channel Analysis (SCA) and SIFA has been introduced in [DDE$^+$20]. In this approach, the non-linear layer should be implemented by Toffoli gates and the whole design should be masked. Then, the authors claimed that a simple duplication can prevent a single-fault SIFA. In [BBB+20], a randomized duplication-based approach is presented, where no masking is needed.

State-of-the-Art Fault Verification. Practical evaluation of countermeasures against fault attacks on physical devices and real products is a complex and time-consuming task and needs considerable expertise and experience. Hence, this certainly highlights the necessity of verification tools and automated analysis techniques to accelerate evaluation and assist designers in analysis of countermeasures. Moreover, it can help to reduce the cost of the fabrication process while maintaining the desired level of security.

In 2017, the authors of [BGE+17] presented a tool for automatic construction of algebraic fault attacks called *AutoFault*. AutoFault works on gate-level netlists and uses a SAT solver to detect possible vulnerabilities in a given design without deeper knowledge of the cipher construction. However, the user has to define a list of fault locations which limits evaluations to the corresponding areas of a target design. Hence, if AutoFault is used to verify countermeasures against fault attacks, the tool could report false-positive results since a full coverage of all possible fault events is impossible.

In [KRH17], a framework, called XFC, for fault characterization in block ciphers was presented. It receives the block cipher specification and a fault model in order to determine locations for fault injection during the execution of the encryption. By tracing the fault propagation and its effects on the ciphertext, the tool evaluates the exploitability of a fault in terms of DFA and returns the computational complexity of the recoverable part of the (round) key. However, while XFC is mostly limited to a specific class of DFAs, ExpFault [SMD18] is designed to cover even more fault analysis techniques. Unfortunately, even though both tools can help adversaries to find the optimal location to inject faults and facilitate fault attacks, they are not able to assist designers in assessing the security of implementations equipped with fault attack countermeasures. As a consequence, this issue was addressed in a framework called *SAFARI* [RRHB]. More precisely, this framework uses XFC to automatically identify locations that can be exploited by fault injection attacks, given a description of the (unprotected) target block cipher in a dedicated block cipher specification language. Then, based on user defined security levels, SAFARI automatically equips the given block cipher with a parity or redundant-based countermeasure and returns HDL or C code accordingly. Recently, another work that focuses on the exploitability of fault injection attacks on microcontrollers was presented at CHES in 2019 [HBZL19]. The authors propose a tool called *TADA* which automatically detects vulnerabilities of a block cipher software implementation on assembly level and returns exploitable faults with the help of an SMT solver.

Unfortunately, all aforementioned works aim to detect exploitable fault injection attacks on hardware or software implementations of block ciphers, hence taking an adversarial perspective. Only [RRHB] additionally applies protection mechanisms to susceptible areas. However, none of these works take the perspective of the designer in targeting the assessment and formal verification of countermeasures against fault injection attacks implemented for hardware devices. More specifically, all those approaches are not able to verify a given design considering all possible fault events that could occur under all valid input combinations.

There are a few works addressing this topic by proposing open-source fault simulators for fault diagnosis [NCP92, LH96, BN08]. However, their application to cryptographic fault analysis is quite limited as they can simulate only a single-bit fault injection. As a result, VerFI [AWMN20] is the first automated open-source cryptographic fault diagnosis tool designed to evaluate fault-protected cryptographic implementations. For this, the tool directly operates on the gate-level netlist of a hardware design and is able to assess detection, infection, and correction-based countermeasures. Moreover, the user is able to define a fault model, a bounded adversary model, the location of the faults, the desired clock cycles for fault injections, and some input test vectors for simulation. Then, the tool simulates the circuit considering the parameterized fault injection and provides the coverage for every test vector and a final overall result including the total number of faults, all the non-detected faults per input test vector, reporting the location and type of faults, and the corresponding faulty outputs, which may assist the designer to identify the design failures.

Limitations of VerFI. Although VerFI facilitates the verification of fault-protected implementations, the result of the analysis depends on the selected input test vector(s). Hence, it is possible that the tool indicates the security of a design under a certain set of test vectors, while different test vectors would lead to observable or exploitable faults.

For this, let us consider a simple PRESENT S-box implementation [BKL+07], protected by a single bit of parity, as depicted in Figure 1. More precisely, the S function receives a 4-bit input $S_{in} = \langle a, b, c, d \rangle$ and provides the 4-bit S-box output $S_{out} = \langle x, y, z, t \rangle$, where a and x are most significant bits. Simultaneously, the redundant part S' operates on the 4-bit input, independent of S, and estimates the parity bit of the S-box output. Eventually, the consistency check module verifies the correctness of the S output given the estimated parity bit and indicates a fault in case of inconsistency. However, such a design is not necessarily secure against single-bit fault injection in case fault propagation occurs in the S function. In other words, for some test vectors, an attacker can inject a single-bit fault in such a way that an even number of faulty bits appear at the output, hence no opportunity to be detected by the consistency check module. One of such cases is shown in Figure 1, where a single-bit fault propagates to two different output bits (x and y) depending of input test vector, i.e., $S_{in} \in \{\texttt{0x1,0x2,0x3,0x9,0xA,0xB}\}$. More precisely, the tool reports all single-bit faults are detected when $S_{in} \in \{\texttt{0x0,0x4,0x5,0xD,0xE,0xF}\}$ and there is at least one non-detected fault in the rest test vectors due to fault propagation. To mitigate this issue *independence property* has been defined in [AMR+20] to guarantee an n-bit induced fault affect at most n-bit output bits. To this end, no cell should be involved in the computation of multiple output wires. As one can see, this property is not fulfilled in the given example, leading to insecure implementation for some test vectors in the underlying adversary model.

Hence, VerFI confirms the security of the design if the evaluation is only based on a limited number of test vectors, while additional test vectors could reveal the flaw. This behavior becomes even more challenging with increasing circuit size and number of inputs, as using VerFI it is almost impossible to check all input combinations for such a fault-protected cryptographic design. As a consequence, this highlights the importance of an automated tool that does not rely on simulation of test vectors but symbolically checks all possible cases under given fault models.

3 Verification Concept

Before we present our verification approach in more detail, we introduce appropriate models for fault events and describe important terms required for diagnosing fault effects. The proposed verification approach covers the generation of circuit models, symbolic fault injection, and the corresponding fault diagnosis.

3.1 Fundamental Terminology

Formal verification of security requires formal descriptions and definitions of adversary models and security properties. More precisely, given capabilities and limitations of an adversary model, formal verification can prove security properties of any design under

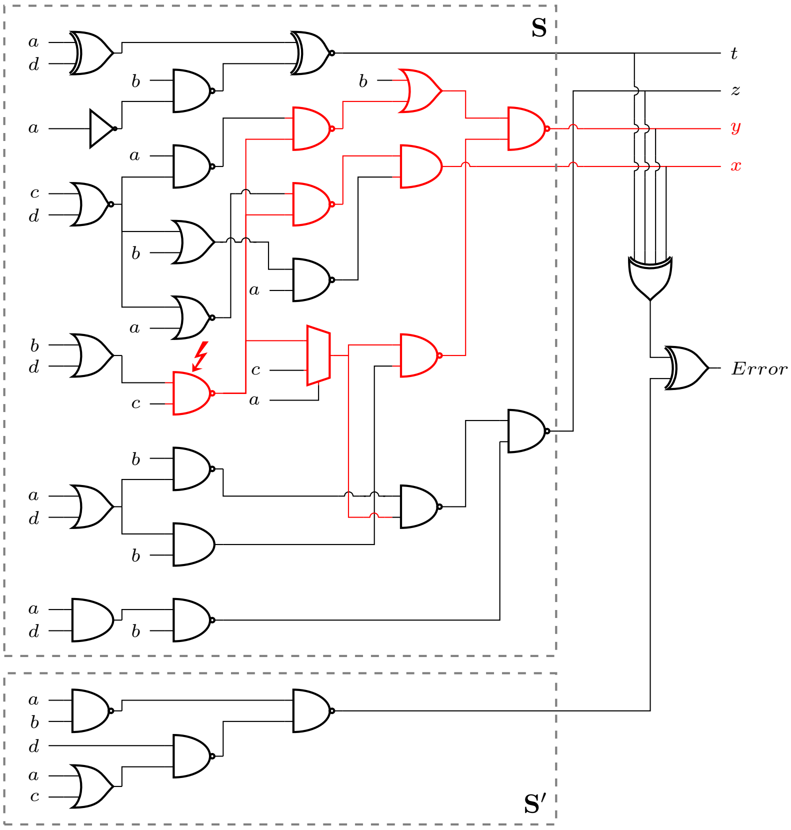

Figure 1: A simple PRESENT S-box implementation protected by a single-bit parity. The shown fault propagation happens when $\langle a, b, c, d \rangle \in \{\texttt{0x1,0x2,0x3,0x9,0xA,0xB}\}$.

verification in the presence of the given adversary models. To this end, we briefly outline fundamentals of our basic fault injection models as follows.

Fault Events. Any accidental condition that results in a malfunction or misbehavior of a digital logic circuit is considered as a fault event. However, while environmental faults have an erratic and random nature, adversarial faults are often precisely located and purposely injected into the circuit. Depending on their retention time, fault events can be classified as *transient*, *persistent*, or *permanent*. While transient fault events have a dynamic nature and volatilize after certain periods or changes in the circuit, elimination of persistent fault events requires a full reset of the circuit or system, whereas permanent faults are of static nature and will remain permanently. However, when modeling fault events, considering only transient fault events is sufficient, as any persistent or permanent fault event can be modeled as a repetitive transient fault event.

Observing that digital logic circuits are used to implement the computation of arbitrary Boolean functions $\mathsf{f} : \mathbb{F}_2^i \mapsto \mathbb{F}_2^o$, any fault event in such a digital logic circuit can be precisely modeled by another Boolean function $\mathsf{f}' : \mathbb{F}_2^i \mapsto \mathbb{F}_2^o$ as the authors of [RBSG21] proposed. More precisely, we model fault events at the structural level of logic circuits, assuming logic gates as atomic components, while the misbehavior of a single logic gate is considered as a fault event. As a consequence, changing the functionality of the misbehaving logic gate in the context of the entire circuit results in a clear specification of the *faulty* function f' that can be compared to the *golden*, i.e., fault-free, function f.

Fault Positions. Given our adversary model, as outlined in Section 2.4, the adversarial capabilities are mostly determined and limited by the number of fault events that can be purposely injected into a single evaluation of a digital logic circuit[3]. More specifically, any fault injection might be limited and constrained in *spatial* or *temporal* dimension. For the spatial dimension, we mostly distinguish between *combinational* and *sequential* logic gates that are considered as source for transient fault events. Further, in addition to the spatial locations, each adversary is also limited in the number of fault injections, i.e., the number of fault events that can be caused simultaneously within the same clock cycle. In fact, for the *temporal* dimension, we distinguish between *univariate* and *multivariate* fault injections. In that sense, univariate fault injections only consider fault events occurring in the same clock cycle, while for multivariate fault injections, fault events can occur in different clock cycles.

As a result, the total number of possible fault events, ultimately describing and limiting the adversarial capabilities, is derived as the product of the spatial and temporal limitations. This means, given n fault events in spatial dimension, and v fault events in temporal dimension, the total number of fault events that is injected into a single circuit evaluation is yielded by $n \times v$. Further, depending on the adversary model and if necessary, the distribution of fault events can be adjusted, e.g., according to a *uniform* or *biased* distribution. However, whenever possible, we opt for an exhaustive fault verification, hence, allowing to cover any possible fault distribution.

Classifying Fault Effects. As indicated before, diagnosis of fault events and effects requires knowledge of the expected behavior. As a consequence, comparing the faulty behavior to the expected behavior of a golden circuit allows to evaluate and examine the *fault effectivity.*

In the context of pure *fault-detection* countermeasures, fault handling is delegated and escalated to the system. More precisely, in such a context, the circuit under diagnosis might expose a misbehavior that is detected and clearly communicated as such to the system level. As a consequence, for fault-detection countermeasures, we usually distinguish between *ineffective*, *detected*, and *effective* faults. For this, each fault event that does not lead to an observable misbehavior is classified as *ineffective*, while all fault events that clearly lead to a misbehavior that is not detected by the circuit are marked as *effective* faults, leaving the remaining events in the class of *detected* fault events. In contrast to this, *fault-correction* countermeasures attempt to correct any detected misbehavior immediately such that only *ineffective* or *effective* faults can be observed.

3.2 Verification Approach

In the following, we present our verification approach for fault injection countermeasures on hardware devices. More precisely, we explain how we use a Verilog gate-level netlist of a digital logic circuit to create an appropriate model. This model is used as a foundation to introduce techniques using BDDs to perform efficient evaluations of fault events. Eventually, we provide more insights of optimization strategies that allow supporting larger circuits.

Requirements for Cryptographic Fault Verification. There are some simulation tools [NCP92, LH96, BN08] in the field of integrated circuits testing, also known as reliability analysis, that examine the working environment stress, e.g., thermal cycling and vibration, or the potential manufacturing failures in a chip. However, they are not suitable for cryptographic fault analysis and verification of fault-protected implementations as they are commonly limited to single-bit faults. Moreover, often the user cannot set different fault

[3] In the context of this work, we focus on analysis and verification of clock-synchronous digital logic circuits only.

models or specify desired locations for fault injection. It becomes even more challenging if the evaluated design incorporates dedicated countermeasures against fault attacks. In particular for detection- and infection-based countermeasures, the design returns a fixed value or a random value completely unrelated to the secret key if a fault event was recognized. Hence, the evaluation tool must be able to anticipate the behavior of the design and the integrated countermeasure for correct evaluation. While all these facts are supported by the recently-introduced fault-diagnostics tool VerFI [AWMN20], the result of such an evaluation is based on the given test vector(s). This may lead to a false-positive result. Namely, some faults may appear at the output only for certain input values and might be not detected by every test vector, like the example provided in Section 2.4. In this work, we mainly focus on cryptographic fault analysis that naturally covers every possible test vector, avoiding false positives.

Abstraction Levels. The definitions introduced in Section 2.3 allow us to introduce two abstraction levels *structural* and *functional*. The structural level incorporates the edges and vertices of the DAG, i.e., the wires in the circuit $\mathbf{C}$ connecting the Boolean gates from $\mathcal{G}$. In a verification environment, the structural level is used to define and distinguish different areas of the original circuit, e.g., the register stages or dedicated modules that should be considered in an analysis. Furthermore, the structural level gives us the opportunity to develop special optimization strategies as we describe later in this section. However, the actual faults are injected at the functional level – directly in the combinational or sequential gates. At this level, we can precisely and generically cover several known fault models summarized in Section 2.4.

From Netlist to Direct Acyclic Graph. Figure 2 depicts the verification approach which we follow in this work. As already mentioned above, we analyze hardware circuits based on their (Verilog) gate-level netlist. In the first step, the netlist is transformed into the circuit model introduced in Section 2.3 and therefore converted into a DAG $\mathbf{D}$. The underlying data structure allows us to perform several preprocessing steps at the structural level, as follows.

- First, each node $d \in \mathbf{D}$ is attached with an information holding the gate type. It is accessed by the function `type` (d) and returns one of the following values from $\mathcal{G}_t$.

$$\mathcal{G}_t = \{\mathsf{in}, \mathsf{out}, \mathsf{not}, \mathsf{buf}, \mathsf{reg}, \mathsf{and}, \mathsf{nand}, \mathsf{or}, \mathsf{nor}, \mathsf{xor}, \mathsf{xnor}\}$$

- Second, dependencies between the existing nodes in $\mathbf{D}$ are identified. To be more precise, each node $d \in \mathbf{D}$ is equipped with its *propagation path*, i.e., with a list of nodes that are influenced by the output of d.
- Third, all nodes in $\mathbf{D}$ are separated into two classes depending on whether the corresponding logic gate g is from $\mathcal{G}_r$ or from $\mathcal{G}_c$, i.e., g is whether a sequential or combinatorial gate, respectively. We access this information for a given node d by the function `location` (d).
- Forth, the structural level is perfectly suited to extract topological characteristics of the underlying circuit. This includes the assignment of each node $d \in \mathbf{D}$ to its logic stage.

A single logic stage consists of all combinatorial gates between two successive register stages. Special cases are 1) the first logic stage where the combinatorial gates are between the primary inputs and the first register stages, and 2) the last logic stage where the combinatorial gates are between the last register stages and the circuit's primary outputs. We use the function `stage` (d) to refer to the logic stage of node d.

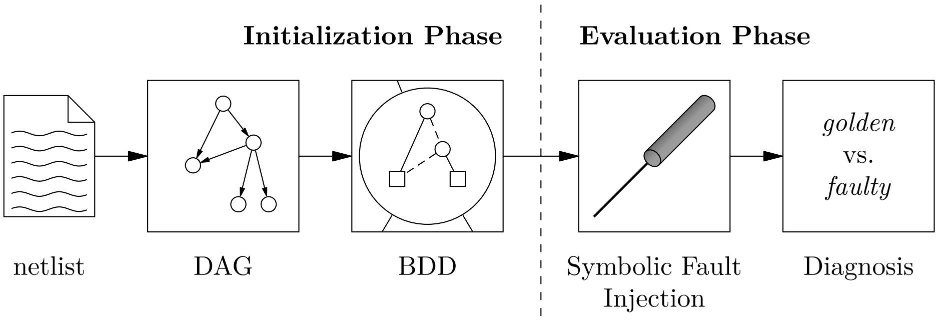

Figure 2: Flow of the proposed verification approach.

Symbolic Simulation using BDDs. The next step in our verification approach consists of mapping the Boolean function associated with each node $d \in \mathbf{D}$ to a BDD which includes the entire subgraph spanned by the node d. Therefore, the DAG is topologically sorted and each node d is evaluated starting from the primary inputs. For each primary input, i.e., $\texttt{type}\,(d) = \mathsf{in}$, a new BDD variable is introduced. For all remaining nodes $d \in \mathbf{D}$ a BDD is constructed from the fan-in BDDs, based on the Boolean function associated with the node d. As an example, let us assume that a node d is associated with an and-gate. Therefore, d has two input edges connected to two previous nodes which have already been evaluated (due to the topological sorting) and the corresponding BDDs have been constructed. Then, the BDD for d is constructed by computing the logical *and* of both fan-in BDDs based on the concept given in Section 2.2.3.

We decided to select BDDs as the underlying data structure to model a digital logic circuit since they offer many advantages. First, BDDs were originally proposed for defining, analyzing, testing, and implementing digital Very Large Scale Integration (VLSI) circuits [Ake78]. Therefore, they seem to be a natural choice for our application, i.e., verifying hardware countermeasures against fault injection attacks. Second, one core idea of BDDs is to work with symbolic simulations which inherently consider all possible states of the BDD variables. Hence, the verification of a digital logic circuit is not limited to a predefined set of test vectors (inputs) but rather all valid inputs are tested and evaluated. This procedure avoids false positives as discussed in Section 2.4. Third, since executions of Boolean functions over BDDs are elementary operations (cf. Section 2.2), injecting faults by exchanging the associated Boolean function of the target node in the DAG $\mathbf{D}$ with a faulty one is a straightforward task that results in simple re-computation of the corresponding BDD.

Symbolic Fault Injection. Our concept of symbolic fault injection is based on the very generic approach presented in [RBSG21], i.e., modeling faults by replacing the original Boolean operation f of a target gate $g \in \mathbf{C}$ with another Boolean operation f' chosen from the same set of functions according to a predefined mapping τ.

In the context of our verification approach, when analyzing the effect of a fault injection in a target logic gate, the corresponding graph node $d \in \mathbf{D}$ is replaced with another graph node d', according to a fault mapping τ. However, since each graph node is explicitly associated with a Boolean operation, the replacement of the graph node not only changes the structural description of the circuit node but also affects the functional behavior. More precisely, while d is associated with a Boolean operation f, the replaced graph node d' is associated with a different Boolean operation f', necessitating a re-generation of the BDD for all subsequent graph nodes (affected by the fault injection). Note that for the remainder of this work, we denote the structurally modified DAG $\mathbf{D}'$ as the *faulty model*, while we refer to the original, fault-free DAG $\mathbf{D}$ as the *golden model*.

In fact, our verification approach is designed to analyze the golden model $\mathbf{D}$ under all possible fault events that can occur for a given fault model $\zeta(n, t, l)$. More specifically, the fault model ζ determines the fault verification process in terms of the number of graph nodes n that should be faulted, the fault mapping τ that is considered to replace the target nodes, and which circuit location l should be considered (i.e., combinational logic, sequential logic, or both).

Therefore, in the first step, our verification approach creates a set of nodes Λ. Particularly, the nodes in Λ are extracted from the golden model $\mathbf{D}$ according to the considered location parameter l, such that

$$\Lambda = \{\, d \in \mathbf{D} \mid \texttt{location}\,(d) = l \,\}\,.$$

In a next step, the nodes in Λ are separated into s subsets θ_i (s denotes the total number of logic stages) holding all nodes belonging to the same logic stage, i.e., each subset θ_i is defined as $\theta_i = \{\, \lambda \in \Lambda \mid \texttt{stage}\,(\lambda) = i \,\}$ for $0 \le i < s$. Therefore, the set of all valid gates $\lambda \in \Lambda$ categorized based on logic stages is noted by $\Theta = \{\theta_0, \theta_1, ..., \theta_{s-1}\}$. In particular, such a categorization allows to distinguish between univariate and multivariate fault injections, i.e., different fault injections with respect to the temporal dimension.

Besides considering the location parameter l and the separation in the temporal dimension to create valid sets of nodes that should be faulted, we further consider the number of fault events n that should be injected simultaneously in a single subset θ_i. Therefore, we introduce the sets Γ_i for $0 \le i < s$ which hold all combinations of n nodes that are available in a single subset θ_i, formally defined as

$$\Gamma_i = \{\, \gamma \mid \gamma = \{d \in \theta_i\}, |\gamma| = n \,\}\,.$$

Note, however, that the cardinality of each Γ_i, i.e., the number of valid combinations of nodes that need to be evaluated in each logic stages, drastically increases in $|\theta_i|$ and n since $|\Gamma_i| = \binom{|\theta_i|}{n}$.

Next, given a valid set of target nodes $\gamma \in \Gamma_i$, each node in $\gamma = \{d_0, \dots, d_{n-1}\}$ is associated with a Boolean operation $\mathtt{f}$ which is replaced by faulty operations according to the fault type t defined in $\zeta(n, t, l)$. In particular, the fault type t (e.g., bit-flip, set, or reset) is defined and described by a fault mapping τ (cf. [RBSG21]). For example, a fault mapping for the gate type $\texttt{and}$ could be defined by $\tau : \{\texttt{and}\} \mapsto \{\texttt{set}, \texttt{reset}, \texttt{nand}\}$. Hence, the number of different fault mappings that can occur for one γ depends on the cardinality of the corresponding fault mapping and is determined by

$$\prod_{j=0}^{n-1} \Big|\tau\big(\texttt{type}\,(d_j)\,\big)\Big|,\ d_j \in \gamma.$$

Eventually, each of these valid combinations leads to a new faulty model $\mathbf{D'}$ which should be compared to the golden model $\mathbf{D}$ to determine the effect of the injected fault (more details are provided in the subsequent paragraph).

As already mentioned above, our verification approach considers univariate and multivariate fault injections. For univariate fault injections, faults are injected in only a single set Γ_i. In contrast, for multivariate fault injections, v different sets Γ_i are selected, where v denotes the number of different logic stages that can be faulty at the same time (e.g., setting $v = 2$ would describe a bivariate fault injection). Note, that in each logic stage (temporal dimension), n nodes can be faulted such that $v \times n$ nodes of the golden model $\mathbf{D}$ are affected. Therefore, analyzing multivariate fault injections drastically increases the combinations of nodes that need to be evaluated. More precisely, each selection of v different sets creates

$$\prod_{v} |\Gamma_i|$$

valid combinations.

Fault Diagnosis. The ultimate goal of our fault verification approach is the diagnosis of fault effectiveness and severity (cf. Figure 2). For this, given a golden model $\mathbf{D}$ and a faulty model $\mathbf{D}'$, the effects of fault injection in $\mathbf{D}$ resulting in $\mathbf{D}'$, are evaluated by analyzing and comparing both models. Particularly, the output nodes of both models are combined to new BDDs (commonly by an exclusive-or which we highlight in more detail in Section 4.2) in order to detect any differences in the outputs considering all valid assignments of the primary input variables. This strategy is especially beneficial for the data-structure of BDDs since counting the number of satisfying variable assignments, i.e., leading to a logical $\mathbf{1}$, can be accomplished efficiently. Hence, determining and counting the satisfiability of the BDDs combining the outputs of the golden and faulty models, directly yields the number of input combinations leading to a difference in both models.

More precisely, based on the analyzed fault-injection countermeasure, incorporated in design under test, the combined BDDs of the golden and faulty models are used to determine the number of *effective*, *ineffective*, and *detected* faults, as well as the total number of fault events as introduced in Section 3.1. Note, however, that the exact fault diagnosis procedure depends on the underlying countermeasure which we discuss in more detail in Section 4.2.

Optimizations. As already indicated before, this verification approach poses some challenges with respect to the complexity when analyzing large circuits or when the number of fault injections n increases. Therefore, we further propose two optimization strategies which both rely on the structural analysis of the circuit model while being independent of the functional behavior of the circuit, i.e., the realized logical function.

The first strategy benefits from the identification of fault propagation paths, which are determined in the initialization phase. More precisely, the propagation paths are determined by a backwards-iterating algorithm given a topologically sorting of the DAG $\mathbf{D}$. Hence, in a breath-first search the algorithm considers each node $d \in \mathbf{D}$ and adds the propagation paths of all nodes d_i connected to the output edges of d along with the node d_i itself. This procedure generates topologically sorted lists of propagation paths since the nodes from the deepest logic levels are added first. Then, assuming a target node $\lambda \in \Lambda$ is faulted, i.e., the associated Boolean function is replaced, we can observe that not all BDDs associated with nodes $d \in \mathbf{D}$ need to be re-evaluated. In particular, evaluating only the nodes that are located on the fault propagation path is sufficient and reduces computational overhead especially when injecting faults on gates in deeper logic levels.

The second optimization strategy reduces the number of nodes that are tested in the evaluation phase (cf. Figure 2). This is achieved by creating a subset $\Lambda_{\text{red}} \subset \Lambda$ which is used instead to generate the valid combinations of target nodes in Θ, using the following ideas and observations. First, registers form synchronization points in a digital logic circuit $\mathbf{C}$ (cf. Definition 4) and occurring fault events will eventually manifest in such register stages. Hence, it is straightforward that all nodes in Λ associated with a register in $\mathbf{C}$ also need to be included in Λ_{red}. Second, all nodes $d \in \mathbf{D}$ associated with gates that are directly connected to registers (i.e., whose output edges are connected to a register input) are added to Λ_{red} as well since they influence the behavior of the synchronization points immediately. Third, we observed that most digital logic circuits have *sensitive gates* directing faults from several locations through the circuit, eventually manifesting in registers. More precisely, we cluster gates to Boolean functions $\tilde{\mathsf{f}} : \mathbb{F}_2^i \mapsto \mathbb{F}_2^1$ with $i > 1$, i.e., to Boolean functions that providing only single-bit outputs. The output gates of such clusters symbolize sensitive gates and fault propagations within such clusters are local and always pass them. Hence, fault injections in these clusters can be modeled by considering the sensitive gates only. Therefore, we add all nodes associated with sensitive gates to the reduced set of nodes Λ_{red}.

Note, this approach selects cluster of gates in a conservative fashion, i.e., each gate

Algorithm 1: Complexity Reduction

```
   Input  : Golden circuit model $\mathbf{D}$, set of valid fault location (nodes) $\Lambda$
   Output : Set of reduced fault locations $\Lambda_{\text{red}}$
1  $\Sigma \leftarrow \emptyset$, $\Lambda_{\text{red}} \leftarrow \emptyset$
2  for $\forall d \in \mathbf{D}$ do
3      if type $(d)$ = reg or type $(d)$ = out then
4          $\Sigma \leftarrow \Sigma \cup d$
5          if $d \in \Lambda$ then
6              $\Lambda_{\text{red}} \leftarrow \Lambda_{\text{red}} \cup d$
7          end
8      end
9  end
10 for $\sigma \in \Sigma$ do
11     $\Lambda_{\text{red}} \leftarrow \Lambda_{\text{red}} \cup$ node_in$(\sigma)$
12     $\Phi \leftarrow \sigma$
13     while $\Phi \neq \emptyset$ do
14         $\alpha \leftarrow \Phi[0]$, delete$(\Phi[0])$
15         for $\forall n \in$ node_in$(\sigma)$ do
16             if type $(n) \neq$ reg and type $(n) \neq$ in then
17                 $\Phi \leftarrow \Phi \cup n$
18             end
19             if out_degree$(\alpha) > 1$ and $\alpha \in \Lambda$ and $\alpha \notin \Lambda_{\text{red}}$ then
20                 $\Lambda_{\text{red}} \leftarrow \Lambda_{\text{red}} \cup \alpha$
21             end
22         end
23     end
24 end
```

with fan-out greater than one is treated as a sensitive gate regardless of the fact that output signals are may re-combined by another gate such that they only influence a single wire. Additionally, analyses using the reduced set of nodes Λ_{red} should only be performed in the bit-flip model, i.e., $\zeta(n, \tau_{bf}, l)$. Again, this conservative approach models a worst-case scenario and ensures that a fault event is definitely effective. Hence, both restrictions guarantee full coverage of all possible fault events that otherwise may occur in a non-reduced set Λ.

However, switching to the introduced circuit model $\mathbf{D}$, nodes associated with registers and nodes directly connected to registers can be extracted from $\mathbf{D}$ in a straightforward way. All nodes $d \in \mathbf{D}$ associated with sensitive gates are identified by iterating over all nodes and extract each node d with more than one output edge. All three steps are formally defined in Algorithm 1 describing the complete process of generating the reduced set of nodes Λ_{red}. Here, line 6 adds all nodes associated with registers to the reduced subset Λ_{red} while line 11 considers the input nodes directly connected to the registers. Eventually, line 20 adds all nodes associated with sensitive gates to Λ_{red}.

Finally, we visualize the determination of sensitive gates and corresponding clusters of gates by an exemplary circuit depicted in Figure 3. All together, the exemplary circuit consists of five clusters denoted by $c_0, ..., c_4$. While gates g_5, g_6, and g_7 influence the existing registers directly, they also represent sensitive gates since faults always propagate through them. However, cluster c_4 consists of gate g_4 and g_2 where only g_4 is considered in a verification approach where Algorithm 1 is applied. To summarize, with the presented reduction approach it is enough to cover $\Lambda_{\text{red}} = \{\text{regs}, g_1, g_4, g_5, g_6, g_7\}$

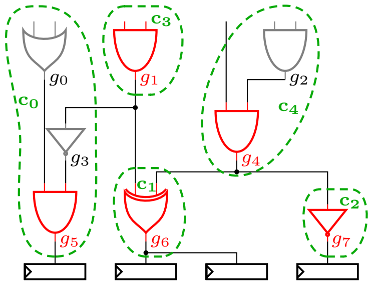

Figure 3: Clustering gates of a given digital logic circuit to reduce the verification complexity.

instead of $\Lambda = \{\mathsf{regs}, g_0, \dots, g_7\}$.

4 The Tool

In this section, we present our fault verification tool FIVER (Fault Injection VERification) which realizes the approaches and concepts from Section 3. For this, we briefly introduce the applied BDD library and explain the general tool flow. Finally, we present some optimization strategies to improve the overall performance of our tool.

4.1 Colorado University Decision Diagram (CUDD) Package

The Colorado University Decision Diagram (CUDD) package is a BDD library developed by Fabio Somenzi at the University of Colorado [Som18]. The library is written in C but provides an interface to C++, used by our tool. Besides a large set of BDD operations offered by CUDD, it provides a large assortment of variable reordering methods. These methods allow reordering BDD variables such that the size of the underlying BDD is optimized. This is especially beneficial when the size of the evaluated circuit increases.

4.2 Tool Flow

In this section, we introduce our fault verification tool in more detail. To start the analysis of a target design, a configuration file needs to be provided first. Afterwards, the internal tool chain is evoked and executed. At the end of the tool chain, an evaluation function is called in order to determine the number of effective, ineffective, and detected faults and generate the final evaluation report.

Configuration. Our fault verification tool uses a configuration file to specify and execute the desired analysis. This configuration file includes and sets parameters controlling the execution environment and host resources (e.g., Central Processing Unit (CPU) cores or memory) as well as the fault model parameters, including the number of fault injections n, the number of simultaneous fault injections v in temporal dimension (e.g., univariate, bivariate, etc.), the location parameter l, and whether the complexity reduction approach should be applied or not. Furthermore, the definition of the fault mapping τ needs to be provided allowing the software to consider custom fault mappings for evaluation and diagnosis. However, along with the software on GitHub[4], we provide template definitions for common fault mappings (e.g., bit-flip, set, reset). Finally, a reference to a blacklist of entity names can be provided, excluding all matching modules from the fault injection process during the evaluation phase.

[4] https://github.com/Chair-for-Security-Engineering/FIVER

Tool Chain. The tool chain is guided by the verification approach introduced in Figure 2. Hence, the Verilog netlist of the target design is parsed first. The outcome is an intermediate representation of the circuit containing the gate type, the list of input nodes, and additional annotations. Based on this intermediate representation, the DAG $\mathbf{D}$ of the golden model is generated. Besides, as this function already processes the intermediate representation, the annotations are used to identify the blacklisted entities. Afterwards, the CUDD library is used to process a topologically sorted representation of the DAG $\mathbf{D}$ and creates a BDD for each node $d \in \mathbf{D}$ based on the associated type. Further, if the configuration file enables the complexity reduction, Algorithm 1 is evoked, while the initialization phase is concluded by extracting all related graph properties including the number of logic stages, propagation paths, and nodes that need to be considered during the analysis.

During the subsequent evaluation phase, a fault verification function is called by passing the fault model parameters, the list of valid nodes, the golden model, and a BDD manager required for the CUDD library. All together, this function handles the most workload by iterating over four nested loops considering the number of fault injections n, the distinction in the temporal dimension, over all valid nodes, and finally on the lowest level over the defined fault mappings in τ. Note that n only determines the upper bound for the number of simultaneous fault injections, i.e., fault injections smaller than n are considered in the analysis as well. This procedure is very common in evaluation of countermeasures against fault injections and is done in the same way in related works [SMG16, RSBG20]. However, on the lowest level, the tool performs the actual fault injection by replacing the types of the target nodes resulting in the faulty model $\mathbf{D}'$.

Evaluation. During evaluation and fault diagnosis, the faulty model $\mathbf{D}'$ is compared to the golden model $\mathbf{D}$. More precisely, the verification tool creates a new BDD for each output node in $\mathbf{D}$ and $\mathbf{D}'$. Specifically, let us denote the associated BDDs by $\mathcal{B}_i^{\mathbf{D}}$ and $\mathcal{B}_i^{\mathbf{D}'}$ for $0 \leq i < o$ and for the golden and faulty model, respectively.

Then, the evaluation BDDs $\mathcal{B}_i$ are generated such that $\mathcal{B}_i = \mathcal{B}_i^{\mathbf{D}} \oplus \mathcal{B}_i^{\mathbf{D}'}$, i.e., all output BDD pairs of the golden and faulty model are combined by an exclusive-or. This procedure allows to identify all input assignments under which the BDDs of the output nodes differ, i.e., a fault occurred and is visible at the output. In order to track occurring faults over the entire model, all $\mathcal{B}_i$ are further combined by an OR-operation leading to a single BDD $\mathcal{B}$. However, as in most detection-based countermeasures, an additional error flag indicates if a fault was detected, this information can be used to distinguish effective and detected faults. Particularly, if the BDD $\mathcal{E}$ of the error flag produces a zero while $\mathcal{B}$ generates a one, a fault injection leads to an effective (undetected) fault. Consequently, in case $\mathcal{E}$ and $\mathcal{B}$ leading both to a one, a fault was successfully detected by the design. As already introduced in Section 3, the data-structure of BDDs naturally covers all combinations for the given BDD variables. The number of combinations leading to a true assignment in a given BDD can efficiently be determined by a function counting the minterms which is also provided as part of the CUDD library. Hence, the number of effective faults is determined as `countMinterms`$(\mathcal{B} \cdot \overline{\mathcal{E}})$ while the number of detected fault is obtained by `countMinterms`$(\mathcal{B} \cdot \mathcal{E})$. Knowing the total number of fault events, the number of ineffective faults can be easily calculated by subtracting the number of effective and detected faults.

Note, that if countermeasures without an error flag should be analyzed, the evaluation function, i.e., the function that combines the output BDDs, needs to be adapted. However, this is easily possible without any deeper knowledge of the applied BDD library.

Report. As a final step, the tool reports all verification results in a text file. This includes a summary of the number of effective, ineffective, and detected faults, as well as the total number of fault scenarios that were tested[5].

[5] All numbers are reported on a logarithmic scale to avoid overflows in the counting the statistics.

Besides, for each detected effective fault, a clear description of the fault is added to the report. More precisely, all faulted gates leading to effective faults are listed as well as the function used to model the fault injection. This allows the designer to accurately determine the cause of the effective fault event in order to fix the flaw in the evaluated countermeasure.

4.3 Optimizations

In Section 3.2, we already introduced two optimization strategies based on determining the propagation paths of all nodes in $\mathbf{D}$ and on the reduction of the number of target nodes that need to be faulted, which we called *complexity reduction*. Besides those two approaches, we applied further optimizations which are directly related to the tool.

Incremental Faulting. The first approach optimizes the application of replacing the Boolean functions defined in a given fault mapping τ. It is only effective for analysis with $n > 1$ and for nodes with $|\tau(d)| > 1$, i.e., for nodes that are changed to more than one function modeling a fault injection. Therefore, let us consider a current state of the faulty model $\mathbf{D}'$ where n nodes are faulted. The tool would step on to the next valid set of fault mappings. But instead of just starting from a new golden model, the tool computes the difference between the previous applied fault mappings and the new fault mappings. Hence, if for example only the fault mapping for one single node changes, only the type of this node is adapted (triggering a re-evaluation of related BDDs) and not all BDDs associated with the remaining $n-1$ need to be re-evaluated. This *incremental faulting* approach can notably reduce computational time since the number of evaluations can be reduced and the same fault events are not performed multiple times.

Resetting Faulty Model. The next optimization approach addresses the *resetting* of the faulty circuit. More precisely, after a set of valid nodes $\gamma' \in \Gamma_i$ was faulted and analyzed, the tool proceeds with the next valid set $\gamma'' \in \Gamma_i$. Therefore, the functions from the nodes γ' in $\mathbf{D}'$ need to be restored to the original functions defined in the golden model $\mathbf{D}$. In a straightforward approach, the golden model $\mathbf{D}$ could just be copied to the faulty model $\mathbf{D}'$ such that $\mathbf{D}'$ is fault free and the fault injections into the nodes defined in γ'' could be performed. However, this process can be very time-intensive especially for larger models $\mathbf{D}$ and therefore for larger circuits $\mathbf{C}$. Instead, we only change the types of the nodes defined in γ' to the original types from the golden model $\mathbf{D}$. Even though this procedure triggers a re-evaluation of all BDDs placed in the propagation paths of the nodes in γ', it turns out that this process increases the performance notably.

Multithreading. Finally, we parallelized the execution of our tool by using OpenMP[6]. The given problem is perfectly suited for parallelization since each set of valid nodes in Γ_i can be evaluated independently. Therefore, the loop that iterates over the sets defined in Γ_i is parallelized into the number of threads set up in the configuration file.

5 Case Studies

In this section, we apply the tool proposed in Section 4 to various cryptographic hardware implementations. More precisely, we evaluated detection-based and correction-based countermeasures against fault injection attacks attached to the lightweight ciphers CRAFT and LED as well as the full block cipher Advanced Encryption Standard (AES). All designs

[6] https://www.openmp.org/

were taken from [AMR+20, SRM20][7] while we unrolled the designs and only evaluated one or two rounds of the given circuit (for sake of complexity). Further, to obtain the Verilog gate-level netlists, we used the Synopsys design compiler with version E-2010.12-SP2.

In the next two subsections, we first present the evaluation results of the considered case studies, before discussing the limitations of our tool with respect to the size of a given circuit and the applied fault models.

5.1 Evaluation Results

We start our experiments by evaluating the counterexample from Section 2.4. Due to the symbolic fault injection approach, our tool is able to detect the existing flaws in the design and reports the corresponding gates leading to the effective fault injections. However, we proceed our analyses with the lightweight cipher CRAFT [BLMR19] since it is built upon a simple structure leading to a small hardware footprint (i.e., a reasonable number logic gates). As a next step, we decided to analyze LED-64 which is also a lightweight cipher but has a more complex structure [GPPR11]. Eventually, we challenge our tool by evaluating an AES-128 as it is roughly 14 times larger than the LED design. All results are summarized in Table 1 obtained from a system running Ubuntu 18.04.2 with an Intel Xeon E5-1660 CPU with 3.2 GHz and 128 GB RAM. For all upcoming results, we fixed the number of threads used by our tool to eight while each thread could use up to 8 GB RAM (this is sufficient for the most analysis considered in this work). More details about the performance with respect to the number of used cores and amount of memory can be found in Appendix A in Figure 4 and Figure 5.

CRAFT. For CRAFT, we consider detection-based countermeasures for a single-round design (protected against 1-bit, 2-bit, and 3-bit fault injections), a two-rounds design (with the same protection levels), and a two-rounds design which is protected against multivariate 1-bit and 2-bit attacks. Additionally, we provide evaluation results for correction-based countermeasures for single-round designs protected against 1-bit and 2-bit fault injections. For the analysis of single-round designs protected by a detection-based countermeasure, we instantiate the fault model as $\zeta(n, \tau_{bf}, cs)$, i.e., we consider bit-flip faults in combinational and sequential gates. The number of injected faults n is adjusted to the countermeasure meaning that it is set to the maximum protection level of the considered design. The evaluations for the 1-bit and 2-bit designs is executed within 0.021 s and 1.496 s, respectively. However, the evaluation of the 3-bit design is more challenging because more than 90 million combinations need to be tested. Without any complexity reduction, this evaluation takes roughly 50 min while the application of Algorithm 1 decreases the evaluation time to only six minutes. Note, that all these analyses are performed under all input combinations for plaintext and key, i.e., 2^{128} valid inputs.

To demonstrate the functionality of FIVER, we also analyzed a subset of the provided countermeasures with fault models instantiated such that they describe fault injections exceeding the capabilities of our tool. The corresponding experiments are marked by red crosses in Table 1. As expected, the reports contain detailed lists of gates leading to effective fault injections.

The analysis of the two-rounds design gets more complex because each output depends on more primary inputs. While the BDD generations for the 1-bit and 2-bit design could be accomplished by the CUDD library and an evaluation could be executed without any complications, the structure of the 3-bit protected design is too complex such that the parsing and BDD generation process fails.

Next, we analyze the two-rounds design protected against multivariate attacks which consist of two register stages and therefore three logic stages. For the 1-bit protected

[7] The HDL code can be accessed at https://github.com/emsec/ImpeccableCircuits

Table 1: Evaluation results for various ciphers protected against different levels of fault injections. A red cross in the last column indicates that the tool found effective faults which, however, is expected since the capabilities of the countermeasures were exceeded for these experiments. Experiments with simulation times marked by ∞ were not finished in a reasonable time such that we only report the number of combinations.

Redundancy	**Verification Parameter**			**Design Properties**			**Analysis Results**		
(Capability*) [bits]	$\zeta(n,t,l)$	Variate	Complexity Reduction	Comb. Gates	Seq. Gates	Logic Stages	Combinations	Time [s]	Security
CRAFT – 1 round (detection)									
1 (1)	$\zeta(1,\tau_{bf},cs)$	univariate	no	845	80	2	766	0.021	✓
1 (1)	$\zeta(2,\tau_{bf},cs)$	univariate	no	845	80	2	151 561	0.769	✗
3 (2)	$\zeta(2,\tau_{bf},cs)$	univariate	no	1 410	112	2	329 730	1.496	✓
3 (2)	$\zeta(3,\tau_{bf},cs)$	univariate	no	1 410	112	2	64 320 469	441	✗
4 (3)	$\zeta(3,\tau_{bf},cs)$	univariate	no	1 679	128	2	91 737 144	2 937	✓
			yes	1 679	128	2	4 665 200	360	✓
CRAFT – 2 rounds (detection)									
1 (1)	$\zeta(1,\tau_{bf},cs)$	univariate	no	1 571	160	3	1 491	0.378	✓
1 (1)	$\zeta(2,\tau_{bf},cs)$	univariate	no	1 571	160	3	417 882	62	✗
3 (2)	$\zeta(2,\tau_{bf},cs)$	univariate	no	2 526	224	3	868 500	157	✓
3 (2)	$\zeta(3,\tau_{bf},cs)$	univariate	no	2 526	224	3	250 984 950	∞	–
			yes	2 526	224	3	7 364 279	408	✗
CRAFT – 2 rounds – multivariate (detection)									
1 (1)	$\zeta(1,\tau_{bf},cs)$	bivariate	no	1 720	160	3	682 832	140	✓
1 (1)	$\zeta(1,\tau_{bf},cs)$	trivariate	yes	1 720	160	3	99 542 528	26 955	✓
3 (2)	$\zeta(2,\tau_{sr},s)$	bivariate	no	2 915	224	3	38 651 200	81 897	✓
CRAFT – 1 round (correction)									
3 (1)	$\zeta(1,\tau_{bf},cs)$	univariate	no	2 868	112	2	2 788	0.081	✓
3 (1)	$\zeta(2,\tau_{bf},cs)$	univariate	no	2 868	112	2	3 201 690	22	✗
7 (2)	$\zeta(2,\tau_{bf},cs)$	univariate	no	17 460	176	2	129 651 034	3 543	✓
			yes	17 460	176	2	10 923 888	130	✓
LED-64 – 1 round (detection)									
1 (1)	$\zeta(1,\tau_{bf},cs)$	univariate	no	1 541	0	1	1 301	0.064	✓
1 (1)	$\zeta(2,\tau_{bf},cs)$	univariate	no	1 541	0	1	846 951	9.558	✗
3 (2)	$\zeta(2,\tau_{bf},cs)$	univariate	no	2 435	0	1	1 730 730	27	✓
3 (2)	$\zeta(3,\tau_{bf},cs)$	univariate	no	2 435	0	1	1 072 477 550	12 722	✗
4 (3)	$\zeta(3,\tau_{bf},cs)$	univariate	no	2 916	0	1	1 654 087 449	17 348	✓
			yes	2 916	0	1	3 983 413	94	✓
AES-128 – 1 round (detection)									
1 (1)	$\zeta(1,\tau_{bf},cs)$	univariate	no	24 864	0	1	24 432	22	✓
4 (2)	$\zeta(2,\tau_{bf},cs)$	univariate	no	34 159	0	1	298 473 528	∞	–
			yes	34 159	0	1	56 632 584	471 281	✓

* The capability determines the maximum number of faults that can be detected or corrected by the corresponding countermeasure.

design, we perform a bivariate ($v = 2$) and a multivariate ($v = 3$) evaluation, where the multivariate analysis takes roughly 7.5 h due to the increased number of combinations (even though we enabled the complexity reduction) as explained in Section 3.2. However, a bivariate analysis for the 2-bit protected design with $\zeta(2, \tau_{bf}, cs)$ is out of scope since the number of combination is too large (over 200 billion). Nevertheless, switching to the fault model $\zeta(2, \tau_{sr}, s)$ which can be used to model fault injections caused by electromagnetic pulses (cf. [RBSG21]), could reduce the number of combinations such that the analysis finishes within 23 h. Note that applying Algorithm 1 would not reduce the complexity since $\zeta(2, \tau_{sr}, s)$ only considers nodes associated with register anyways and they would also be added to Λ_{red}.

Eventually, we analyze the single-round CRAFT designs protected by correction-based countermeasures against 1-bit and 2-bit fault injections. To do so, we first slightly adapted the fault diagnosis function such that we count the effective faults by `countMinterms` ($\mathcal{B}$) (cf. Section 4.2) as the correction-based countermeasures does not provide an error flag. Therefore, we only count effective and ineffective faults but we cannot distinguish the number of corrected faults. However, the analysis of the 1-bit protected design is performed without reducing the set of target nodes and within 0.081 s. Switching to a 2-bit protected design requires seven bits of redundancy resulting in an increased number of target gates. Nevertheless, our tool can analyze the 130 million fault combinations in under 1 h and validates the security of the design. Again, applying the proposed approach to reduce the complexity (i.e., Algorithm 1), reduces the number of fault combination to roughly 10 million while the simulation time is decreased to only 130 s.

LED. In our next case study, we analyze a single-round design of LED-64 protected by detection-based countermeasures against 1-bit, 2-bit, and 3-bit fault injections. For all three designs, we selected $\zeta(n, \tau_{bf}, cs)$ as fault model to allow a fair comparison to the CRAFT case study. As for CRAFT, the 1-bit and 2-bit countermeasure can be analyzed in a few seconds although the evaluation time increases compared to the analyses for the CRAFT design. Switching to the 3-bit protected design results in over 1.6 billion combinations that need to be tested. Nevertheless, our tool is able to perform this evaluation in under 5 h without any complexity reduction applied. Enabling the complexity reduction reduces the evaluation time roughly by a factor of 185. We also tried to analyze a two-round design of LED-64 but due to the increased dependencies of the outputs on the primary inputs, we are not able to parse the circuit into BDDs.

AES. In our last case study, we analyzed AES-128 protected by detection-based countermeasures against 1-bit and 2-bit fault injections. While the analysis for the 1-bit protected design can easily be managed by our tool (in only 22.5 s), the 2-bit protected design is more challenging. Hence, due to the enormous amount of gates (over 34 000), the number of combinations drastically increases. Therefore, we are only able to analyze the design by applying Algorithm 1 to reduce the complexity. Even then, the evaluation takes roughly 5.5 d but, nevertheless, it is manageable by our tool.

5.2 Limitations

Given the results of the three case studies, we now identify limitations of our tool with respect to circuit sizes and fault models.

Circuit Size. In two cases (CRAFT two rounds, LED-64 two rounds) our tool is not able to parse the circuit into the proposed data structures, i.e., the construction of the BDDs does not terminate. These problems occur because the outputs of the given circuit and therefore the BDDs of the output nodes in $\mathbf{D}$ depend on too many inputs, i.e., the depth of the BDDs increases. More precisely, the two-round CRAFT design (equipped with 3-bit protection) only consists of 3 739 gates which is clearly not the limiting factor since larger designs (e.g., CRAFT correction, AES) can be processed by our tool. This leads to the conclusion, that the circuit structure (i.e., the realized Boolean function) instead of circuit size prevents the parsing into BDDs. For the two round LED-64 design, each output BDD depends on all 64-bit plaintext variables and on all 64-bit key variables. Hence, circuits with similar structures and dependencies are out of scope for our tool which, however, is expectable since otherwise common block ciphers could probably be broken. More precisely, if our tool could successfully parse a two-round LED-64 design, parsing an entire unrolled implementation of LED-64 would probably also possible since the dependencies in the cipher would not increase. Therefore, the whole cipher could be analyzed over all possible combinations of input variables, i.e., considering all valid plaintexts and keys.

Fault Model. Limitations with respect to the fault model naturally occur when the number of simultaneous fault injections n increases or multivariate fault injections should be analyzed. One of this limitation appears for the multivariate 2-bit protected CRAFT design under the model $\zeta(2, \tau_{bf}, cs)$ for $v = 2$. Evaluating this design without any complexity reduction would require to test more than 200 billion different fault combinations. For the given circuit size (i.e., 3 396 gates), this exceeds the capabilities of our tool. However, as already pointed out in Section 3, such a limitation naturally occurs due to the growth of the binomial coefficient. With the introduction of Algorithm 1, we can counteract this growth, but further improvement still remains an open research challenge.

Circuit Structure. As already indicated in Section 2, our tool is limited to unrolled digital logic circuits. This is mainly due to the underlying data structure of DAGs that does not allow any loops in our model. Therefore, a designer of a countermeasure has to unroll a target design before it can be processed by our verification tool.

Iterative Block Ciphers. Although our tool is mostly limited to the analyses of single-round or two-round implementations, we do not see major obstacles with respect to the verification of common countermeasures and the corresponding assertions. Particularly, when considering countermeasures based on linear error codes [AMR+20, SRM20], the underlying scheme usually protects each round with the same mechanism. Hence, an evaluation of a single round (univariate) or two rounds (multivariate) would be sufficient to verify the correctness of a protection mechanism. Similarly, countermeasures that are based on duplication are often equipped with detection or majority voting modules positioned at the end of a cipher execution. Again, these schemes could be seamlessly verified by our framework, focusing the analysis to the last round of the target scheme.

6 Conclusion

In this work, we present a framework to verify the security of countermeasures against fault-injection attacks designed for ICs. Given a Verilog gate-level netlist, our tool relies on BDDs to model the underlying Boolean function of the digital logic circuit and uses symbolic simulation to avoid false-positive results while covering all possible input combinations. Further, assuming different fault models under consideration, our framework automatically identifies potential fault locations and performs a full analysis under all given models. Since complexity of evaluation of digital logic circuits increases with circuit size, we propose various performance optimization strategies, ranging from algorithmic to programming specific techniques.

Eventually, we conduct several case studies to demonstrate the application on real-world digital logic circuits implementing well-established countermeasures against fault-injection attacks. More precisely, we successfully analyze implementations of the lightweight ciphers CRAFT and LED as well as the widespread AES. In fact, our tool is able to analyze more than 90 million fault injections for a single round of CRAFT in under 50 min while still testing all 2^{128} assignments of the primary inputs.

Acknowledgments

The work described in this paper has been supported in part by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and through the projects 406956718 SuCCESS and VE-HEP (16KIS1345) supported by the German Federal Ministry of Education and Research BMBF.

References

[ABC$^+$17] Stéphanie Anceau, Pierre Bleuet, Jessy Clédière, Laurent Maingault, Jean-Luc Rainard, and Rémi Tucoulou. Nanofocused X-Ray Beam to Reprogram Secure Circuits. In *CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2017.

[ADN$^+$10] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When Clocks Fail: On Critical Paths and Clock Faults. In *CARDIS 2010*, volume 6035 of *Lecture Notes in Computer Science*, pages 182–193, 2010.

[Ake78] Sheldon B. Akers. Binary Decision Diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[AMR$^+$20] Anita Aghaie, Amir Moradi, Shahram Rasoolzadeh, Aein Rezaei Shahmirzadi, Falk Schellenberg, and Tobias Schneider. Impeccable Circuits. *IEEE Trans. Computers*, 69(3):361–376, 2020.

[AMT13] Subidh Ali, Debdeep Mukhopadhyay, and Michael Tunstall. Differential fault analysis of AES: towards reaching its limits. *J. Cryptogr. Eng.*, 3(2):73–97, 2013.

[AWMN20] Victor Arribas, Felix Wegener, Amir Moradi, and Svetla Nikova. Cryptographic Fault Diagnosis using VerFi. In *HOST 2020*, pages 229–240. IEEE, 2020.

[BBB$^+$20] Anubhab Baksi, Shivam Bhasin, Jakub Breier, Anupam Chattopadhyay, and Vinay B. Y. Kumar. Feeding Three Birds With One Scone: A Generic Duplication Based Countermeasure To Fault Attacks (Extended Version). *IACR Cryptol. ePrint Arch.*, 2020:1542, 2020.

[BGE$^+$17] Jan Burchard, Mael Gay, Ange-Salomé Messeng Ekossono, Jan Horácek, Bernd Becker, Tobias Schubert, Martin Kreuzer, and Ilia Polian. AutoFault: Towards Automatic Construction of Algebraic Fault Attacks. In *FDTC 2017*, pages 65–72. IEEE Computer Society, 2017.

[BKH$^+$19] Arthur Beckers, Masahiro Kinugawa, Yu-ichi Hayashi, Daisuke Fujimoto, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Design Considerations for EM Pulse Fault Injection. In *CARDIS 2019*, volume 11833 of *Lecture Notes in Computer Science*, pages 176–192. Springer, 2019.

[BKHL20] Jakub Breier, Mustafa Khairallah, Xiaolu Hou, and Yang Liu. A Countermeasure Against Statistical Ineffective Fault Analysis. *IEEE Trans. Circuits Syst.*, 67-II(12):3322–3326, 2020.

[BKL$^+$07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[BLMR19] Christof Beierle, Gregor Leander, Amir Moradi, and Shahram Rasoolzadeh. CRAFT: Lightweight Tweakable Block Cipher with Efficient Protection against DFA attacks. *IACR Trans. Symmetric Cryptol.*, 2019(1):5–45, 2019.

[BN08] Alberto Bosio and Giorgio Di Natale. LIFTING: A Flexible Open-Source Fault Simulator. In *17th IEEE ATS 2008*, pages 35–40. IEEE Computer Society, 2008.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *CRYPTO 1997*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.

[BS03] Johannes Blömer and Jean-Pierre Seifert. Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In *FC 2003*, volume 2742 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 2003.

[Cla07] Christophe Clavier. Secret External Encodings Do Not Prevent Transient Fault Analysis. In *CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 2007.

[CML$^+$11] Gaetan Canivet, Paolo Maistri, Régis Leveugle, Jessy Clédière, Florent Valette, and Marc Renaudin. Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-based FPGA. *J. Cryptol.*, 24(2):247–268, 2011.

[DDE$^+$20] Joan Daemen, Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Florian Mendel, and Robert Primas. Protecting against Statistical Ineffective Fault Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):508–543, 2020.

[DDRT12] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic Transient Faults Injection on a Hardware and a Software implementations of AES. In *FDTC 2012*, pages 7–15. IEEE Computer Society, 2012.

[DEG$^+$18] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. Statistical Ineffective Fault Attacks on Masked AES with Fault Countermeasures. In *ASIACRYPT 2018*, volume 11273 of *Lecture Notes in Computer Science*, pages 315–342. Springer, 2018.

[DEK$^+$18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):547–572, 2018.

[DLM19] Mathieu Dumont, Mathieu Lisart, and Philippe Maurine. Electromagnetic Fault Injection : How Faults Occur. In *FDTC 2019*, pages 9–16. IEEE, 2019.

[FJLT13] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault Attacks on AES with Faulty Ciphertexts Only. In *FDTC 2013*, pages 108–118. IEEE Computer Society, 2013.

[GA03] Sudhakar Govindavajhala and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. In *S&P 2003*, pages 154–165. IEEE Computer Society, 2003.

[Gir04] Christophe Giraud. DFA on AES. In *AES 2004*, volume 3373 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2004.

[GPPR11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED Block Cipher. In *CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2011.

[GSD+08] Sylvain Guilley, Laurent Sauvage, Jean-Luc Danger, Nidhal Selmane, and Renaud Pacalet. Silicon-level Solutions to Counteract Passive and Active Attacks. In *FDTC 2008*, pages 3–17. IEEE, 2008.

[GST12] Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. Infective Computation and Dummy Rounds: Fault Protection for Block ciphers without check-before-output. In *LATINCRYPT 2012*, volume 7533 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2012.

[GYTS14] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa M. I. Taha, and Patrick Schaumont. Differential Fault Intensity Analysis. In *FDTC 2014*. IEEE Computer Society, 2014.

[HBZL19] Xiaolu Hou, Jakub Breier, Fuyuan Zhang, and Yang Liu. Fully Automated Differential Fault Analysis on Software Implementations of Block Ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):1–29, 2019.

[HS13] Michael Hutter and Jörn-Marc Schmidt. The Temperature Side Channel and Heating Fault Attacks. In *CARDIS 2013*, volume 8419 of *Lecture Notes in Computer Science*, pages 219–235. Springer, 2013.

[KRH17] Punit Khanna, Chester Rebeiro, and Aritra Hazra. XFC: A Framework for eXploitable Fault Characterization in block ciphers. In *DAC 2017*, pages 8:1–8:6. ACM, 2017.

[LH96] Hyung Ki Lee and Dong Sam Ha. HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 15(9):1048–1058, 1996.

[MSY06] Tal Malkin, François-Xavier Standaert, and Moti Yung. A Comparative Cost/Security Analysis of Fault Attack Countermeasures. In *FDTC 2006*, volume 4236 of *Lecture Notes in Computer Science*, pages 159–172. Springer, 2006.

[MTOL12] Philippe Maurine, Karim Tobich, Thomas Ordas, and Pierre Yvan Liardet. Yet Another Fault Injection Technique: by Forward Body Biasing Injection. In *YACC'2012*, 2012.

[NCP92] Thomas M. Niermann, Wu-Tung Cheng, and Janak H. Patel. PROOFS: a fast, memory-efficient sequential circuit fault simulator. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 11(2):198–207, 1992.

[O'F20] Colin O'Flynn. Low-Cost Body Biasing Injection (BBI) Attacks on WLCSP Devices. In *CARDIS 2020*, volume 12609 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2020.

[OGT+14] Sébastien Ordas, Ludovic Guillaume-Sage, Karim Tobich, Jean-Max Dutertre, and Philippe Maurine. Evidence of a Larger EM-Induced Fault Model. In *CARDIS 2014*, volume 8968 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2014.

[ORJ+13] Rachid Omarouayache, Jérémy Raoult, Sylvie Jarrix, Laurent Chusseau, and Philippe Maurine. Magnetic Microprobe Design for EM Fault Attack. In *Symposium on Electromagnetic Compatibility*, pages 949–954. IEEE, 2013.

[RAD20] Keyvan Ramezanpour, Paul Ampadu, and William Diehl. RS-Mask: Random Space Masking as an Integrated Countermeasure against Power and Fault Analysis. In *HOST 2020*, pages 176–187. IEEE, 2020.

[RBSG21] Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. Revisiting Fault Adversary Models - Hardware Faults in Theory and Practice. Cryptology ePrint Archive, Report 2021/296, 2021. https://eprint.iacr.org/2021/296.

[RRHB] Indrani Roy, Chester Rebeiro, Aritra Hazra, and Swarup Bhunia. SAFARI automatic synthesis of fault-attack resistant block cipher implementations.

[RSBG20] Jan Richter-Brockmann, Pascal Sasdrich, Florian Bache, and Tim Güneysu. Concurrent Error Detection Revisited: Hardware Protection against Fault and Side-channel Attacks. In *ARES 2020*, pages 20:1–20:11. ACM, 2020.

[RU96] Teresa Riesgo and Javier Uceda. A fault model for VHDL descriptions at the register transfer level. In *EURO-DAC'96*, pages 462–467. IEEE Computer Society Press, 1996.

[SA02] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In *CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2002.

[SFG+16] Falk Schellenberg, Markus Finkeldey, Nils Gerhardt, Martin Hofmann, Amir Moradi, and Christof Paar. Large Laser Spots and Fault Sensitivity Analysis. In *HOST 2016*, pages 203–208. IEEE Computer Society, 2016.

[SGD08] Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. Practical Setup Time Violation Attacks on AES. In *EDCC-7 2008*, pages 91–96. IEEE Computer Society, 2008.

[SJR+20] Sayandeep Saha, Dirmanto Jap, Debapriya Basu Roy, Avik Chakraborty, Shivam Bhasin, and Debdeep Mukhopadhyay. A Framework to Counter Statistical Ineffective Fault Analysis of Block Ciphers Using Domain Transformation and Error Correction. *IEEE Trans. Inf. Forensics Secur.*, 15:1905–1919, 2020.

[SMD18] Sayandeep Saha, Debdeep Mukhopadhyay, and Pallab Dasgupta. ExpFault: An Automated Framework for Exploitable Fault Characterization in block ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):242–276, 2018.

[SMG16] Tobias Schneider, Amir Moradi, and Tim Güneysu. ParTI - Towards Combined Hardware Countermeasures Against Side-Channel and Fault-Injection Attacks. In *CRYPTO 2016*, volume 9815 of *Lecture Notes in Computer Science*, pages 302–332. Springer, 2016.

[Som18] Fabio Somenzi. CUDD: CU decision diagram package-release 2.7.0. 2018.

[SRM20] Aein Rezaei Shahmirzadi, Shahram Rasoolzadeh, and Amir Moradi. Impeccable Circuits II. In *DAC 2020*, pages 1–6. IEEE, 2020.

[ZDCT13] Loïc Zussa, Jean-Max Dutertre, Jessy Clédière, and Assia Tria. Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism. In *IOLTS 2013*, pages 110–115. IEEE, 2013.

A Performance Results of FIVER

Figure 4 shows the evaluation times for different number of cores used by our tool. The results were obtained for a single-round CRAFT design with four bit redundancy under the fault model $\zeta(3, \tau_{bf}, cs)$ and enabled complexity reduction. The memory limit for each CUDD manager was set to 8 GB.

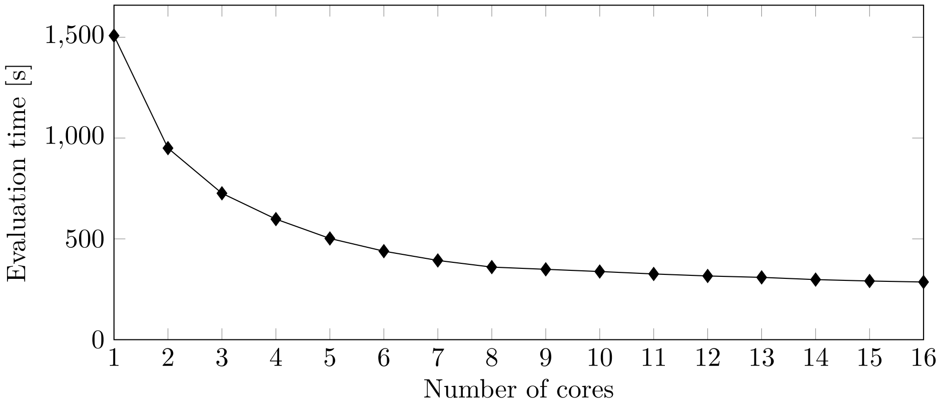

Figure 4: Multithreading performance for a single-round CRAFT design with four bit redundancy.

Figure 5 shows the evaluation performances for different settings of the memory limit for each BDD manager. The results were obtained for the same design and same fault model as in Figure 4 while in this case the number of threads was fixed to eight.

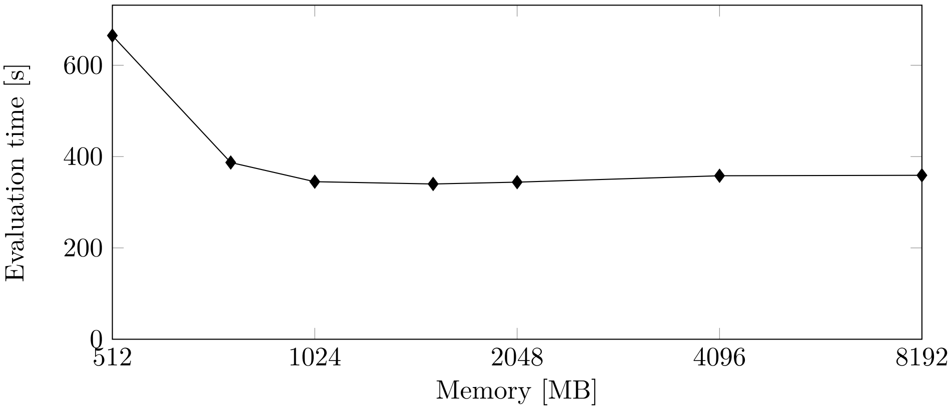

Figure 5: Dependency of the memory limit on the performance for an single-round CRAFT design with four bit redundancy.