Fault-Resistant Partitioning of Secure CPUs for System Co-Verification against Faults

Simon Tollec[1], Vedad Hadžić[2], Pascal Nasahl[2,3], Mihail Asavoae[1], Roderick Bloem[2], Damien Couroussé[4], Karine Heydemann[5,6], Mathieu Jan[1] and Stefan Mangard[2]

[1] Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France, firstname.lastname@cea.fr
[2] Graz University of Technology, Graz, Austria, firstname.lastname@iaik.tugraz.at
[3] lowRISC C.I.C., Cambridge, United Kingdom, nasahlpa@lowrisc.org
[4] Univ. Grenoble Alpes, CEA, List, F-38000, Grenoble, France, firstname.lastname@cea.fr
[5] Thales DIS, France, firstname.lastname@thalesgroup.com
[6] Sorbonne Univ., CNRS, LIP6, F-75005, Paris, France

Abstract. Fault injection attacks are a serious threat to system security, enabling attackers to bypass protection mechanisms or access sensitive information. To evaluate the robustness of CPU-based systems against these attacks, it is essential to analyze the consequences of the fault propagation resulting from the complex interplay between the software and the processor. However, current formal methodologies combining hardware and software face scalability issues due to the monolithic approach used. To address this challenge, this work formalizes the *k-fault-resistant partitioning* notion to solve the fault propagation problem when assessing redundancy-based hardware countermeasures in a first step. Proven security guarantees can then reduce the remaining hardware attack surface when introducing the software in a second step. First, we validate our approach against previous work by reproducing known results on cryptographic circuits. In particular, we outperform state-of-the-art tools for evaluating AES under a three-fault-injection attack. Then, we apply our methodology to the OpenTitan secure element and formally prove the security of its CPU's hardware countermeasure to single bit-flip injections. Besides that, we demonstrate that previously intractable problems, such as analyzing the robustness of OpenTitan running a secure boot process, can now be solved by a co-verification methodology that leverages a k-fault-resistant partitioning. We also report a potential exploitation of the register file vulnerability in two other software use cases. Finally, we provide a security fix for the register file, prove its robustness, and integrate it into the OpenTitan project.

Keywords: Physical Attacks · OpenTitan · Secure Boot · Hardware · Software

1 Introduction

Fault Attacks and Countermeasures. Fault injection (FI) attacks aim to trigger an abnormal execution behavior inside a chip by manipulating the operational conditions of the target device [BCN+06]. Faults are injected by glitching the external clock or voltage supply or by shooting with a laser or an electromagnetic probe into the die [KSV13]. These perturbations corrupt the computations performed by the circuit, leading to the propagation of incorrect values in the microarchitecture and wrong behavior of the system [YSW18]. An adversary exploiting this faulty behavior can attack cryptographic primitives [BDL97, BS97, TMA11, DEK+18], bypass secure boot [VTM+17, dHOGT21], or gain full malicious code execution on a device [NT19].

To ensure robustness against fault injection, security-critical devices such as secure elements implement hardware- and software-based fault countermeasures [JRR+18]. Mitigating fault attacks often relies on spatial or temporal redundancy. Countermeasures like Concurrent Error Detection (CED) schemes are deployed at the hardware level, while software countermeasures implement protections like control-flow integrity [BEMP20, NSL+23] or instruction duplication [BBK+10]. However, software- or hardware-only countermeasures are limited in their ability to protect against fault attacks [YGS+16] or come with a large overhead [AMR+20]. Consequently, recent works combine hardware and software aspects in proposed countermeasures [CCH23, NM23]. Since the fault security of a chip is based on these countermeasures, the correctness and effectiveness of the combination must be ensured. Security evaluation is also crucial, as either conception flaws or the tooling, e.g., the hardware synthesis or the software compiling, could reduce countermeasure security [NOV+22].

Evaluation of System Security. Common security evaluation approaches include penetration testing, which requires a physical chip sample, is costly, time-consuming, and whose results highly depend on the fault injection setup. Simulation or formal verification tools are used to evaluate pre-silicon security and improve fault coverage. Most often, the hardware and the software are analyzed separately. On the one hand, pre-silicon frameworks at the circuit level, such as FIVER [RSS+21] and SYNFI [NOV+22], analyze the resilience of a design's gate-level netlist against fault attacks. These tools rely on bounded verification techniques as they consider cryptographic circuits that have a fixed number of clock cycles. However, they are unable to analyze CPU-based systems or determine under which software conditions identified hardware vulnerabilities can be exploited. On the other hand, software-oriented fault injection frameworks [PMPD14, HSP21, DBP23, KR23] focus on efficiently evaluating the robustness of software countermeasures. They perform their analysis using architectural models instead of actual implementations. Consequently, these frameworks cannot assess the security of combinations of hardware- and software-based countermeasures. Besides, analysis results ignore subtle microarchitectural effects that can lead to vulnerabilities under specific software conditions [LBD+18, TAC+22].

Hardware/Software Co-Verification. Recent research motivates the need to consider both hardware and software to analyze the security of CPU-based circuits [YGS+16, LDPB21]. Although a first formal hardware and software co-verification approach exists [TAC+23], the proposed methodology suffers from scalability issues. For processors, the propagation of the fault effects requires a computationally complex in-depth analysis over multiple clock cycles, whose bound is unknown [TAC+22]. The used bounded verification thus fails to provide generic security guarantees against faults and leads to the classical state explosion problem. Software-related optimizations in the verification, such as constraining some program (faulty) execution paths, show small improvements that significantly depend on the use case and are thus difficult to generalize. Consequently, only up to 100 instructions executed over a microcontroller-like processor can be analyzed for a single fault injection.

Contribution. In this paper, we introduce and formalize the notion of *k-fault-resistant partitioning* to formally prove, at the gate level, whether hardware redundancy-based countermeasures can capture up to k faults injected by an attacker. A k-fault-resistant partitioning is an inductive invariant that implies the robustness of hardware countermeasures of processors, labeled as *k-fault secure*, independently of the program being executed. It thus extends state-of-the-art hardware verification techniques with *unbounded guarantees* for such circuits. We also propose an algorithm to find and prove such k-fault partitions. The outputs of this hardware analysis step are areas of the studied processor with its countermeasures where the invariant does not hold. These verification results allow to

restrict the fault injections we consider when the software is introduced to analyze whether remaining hardware fault locations can lead to software vulnerabilities. The problem of fault propagation and the associated state explosion is drastically reduced, thus enabling a hardware/software co-verification methodology to fully analyze the robustness of systems against fault injection. The k-fault-resistant partitioning notion considers separable spatial and informational redundancy-based protection schemes, that are today the most widely deployed countermeasures in secure elements.

First, we validate our approach by replicating known results on Skinny and AES circuits protected with a code-based CED from the Impeccable Circuits [AMR+20], as no similar work exists on CPUs. Further, we show that our methodology outperforms related work, i.e., proves the 2-fault security of AES in less than 4 h compared to 130 h for FIVER [RSS+21]. In addition, we demonstrate the scalability of k-fault-resistant partitioning by analyzing the 3-fault security of AES, which was not conducted by related work.

Then, to demonstrate the capabilities of our fault injection analysis methodology, we analyze the k-fault security of a development version of the fault-hardened Ibex processor [IBE] used in the OpenTitan secure element [JRR+18]. We first verify two hardware countermeasures, namely its Dual-Core LockStep (DCLS) and the Error Detection Code (EDC) of its register file. Our analysis reveals that DCLS correctly detects any single bit-flip in one of the two cores or in its internal comparison logic, i.e., it is labeled 1-fault secure. However, some single bit-flips injected in the Ibex's register file are not captured by the EDC protection, thus leading to potential software exploitations. The hardware/software co-verification step showcases that an adversary can exploit this vulnerability to manipulate the control flow of the VerifyPIN authentication program [DPP+16] or to perform a differential fault analysis on an AES software implementation [kok]. Nevertheless, we verify the robustness of the OpenTitan secure element running the first step of a secure boot process, as its software countermeasures prevent the register file vulnerability from being exploited. Performance-wise, k-fault-resistant partitioning allows us to analyze a secure processor with a 130 kGE circuit. The hardware/software co-verification step can then address previously intractable software verification of thousands of instructions. All the code and experimental artifacts are publicly available[1].

We disclosed the fault vulnerability of the register file to the OpenTitan project, which acknowledged our findings. In this paper, we provide a fix for the vulnerability and formally prove that the register file is then 1-fault secure. Our fix was integrated[2] into the OpenTitan project.

Outline. This paper is structured as follows. First, we introduce the notations and background in Section 2. Our hardware/software co-verification methodology is described in Section 3, and Section 4 details the fault-resistant partitioning property at the root of our contributions. Section 5 presents the implementation, and Section 6 validates our approach against prior work on cryptographic circuits. Section 7 leverages our full co-verification methodology to evaluate the fault resistance of OpenTitan's secure processor. Finally, Section 8 compares our approach against related work, and Section 9 concludes this work. Proof of our methodology and replicable attack scenarios are given in Appendices A and B.

2 Background

This section first provides a formal description of hardware circuits and then summarizes background on fault attacks as well as hardware-based fault countermeasures.

[1] https://github.com/CEA-LIST/Fault-Resistant-Partitioning
[2] https://github.com/lowRISC/ibex/pull/2117

2.1 Sequential Circuit Model

Definition 1 (Circuit Model)**.** A sequential hardware circuit is modeled as a directed graph $\mathcal{C} = (G, W)$, where G is a set of bit-level circuit elements (gates), and $W \subseteq G \times G$ is the set of wires connecting the gates. Furthermore, each gate $g \in G$ has a type, and belongs to one of the disjoint sets representing inputs I, outputs O, register gates R, and combinational gates C such that $G = I \cup O \cup R \cup C$. Additionally, every loop in the circuit must contain at least one register $r \in R$ to prevent combinational loops.

In the rest of this work, we assume that all registers $r \in R$ are synchronized on the same clock signal. Consequently, we use the clock cycle as the timing unit of the circuit.

Definition 2 (Circuit State)**.** Let $\mathcal{C} = (G, W)$ be a circuit, $I = \{x_1, \ldots, x_{|I|}\} \subseteq G$ be its inputs, and $R = \{r_1, \ldots, r_{|R|}\} \subseteq G$ be its registers, where $|\cdot|$ is the cardinality operator. The state of circuit $\mathcal{C}$ at clock cycle i is the value tuple $\sigma_i^{\mathcal{C}} = (x_1, \ldots, x_{|I|}, r_1, \ldots, r_{|R|})$ containing its inputs I and registers R at the given clock cycle. In the following, we write σ_i and leave out the superscript when the circuit is obvious.

Combinational gates C and outputs O are not part of the state, as their values are entirely determined by the registers R and inputs I at a given clock cycle i. Furthermore, the value of every gate $g \in G$ in the current clock cycle i can be thought of as a function of the current circuit state σ_i, which we write as $g(\sigma_i)$. Assuming the gates G are topologically sorted, we define the notation $S(\sigma_i)$, with $S \subseteq G$, to be the value tuple of all gates $g \in S$ in the state σ_i. As an example, this notation will be used in the following to refer to the circuit's output values $O(\sigma_i)$ at state σ_i, or to assume equalities over output values between two different states, e.g., $O(\sigma_i) = O(\sigma_j)$.

Since circuits execute through time, it is useful to define sequences of consecutive circuit states called *execution traces* where each state depends on its predecessor.

Definition 3 (Execution Trace)**.** Let $\mathcal{C} = (G, W)$ be a circuit with inputs $I \subseteq G$ and registers $R \subseteq G$, and let $\sigma_i = (x_1, \ldots, x_{|I|}, r_1, \ldots, r_{|R|})$ be the current circuit state. The next circuit state at clock cycle $i+1$ is $\sigma_{i+1} = \left(x'_1, \ldots, x'_{|I|}, r'_1, \ldots, r'_{|R|}\right)$, where x'_j are circuit inputs freely chosen by the circuit's environment, and $r'_j = g(\sigma_i)$ are the current state values of the register inputs with $(g, r_j) \in W$. Furthermore, we call a sequence of n such circuit states an *execution trace* $(\sigma_i)_{i=1}^n = (\sigma_1, \ldots, \sigma_n)$.

The theory and methods we introduce in the rest of this work rely on special ways of partitioning a circuit's registers. Here, we give a general definition of a *circuit partitioning*.

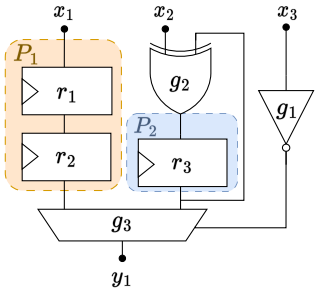
Definition 4 (Circuit Partitioning)**.** Let $\mathcal{C} = (G, W)$ be a circuit. We define a circuit partitioning $\mathcal{P} = \{P_j\}_{j=1}^m$ as a complete partitioning of R such that P_j are disjoint sets of registers, i.e., $P_j \subseteq R$, with $\forall j \neq j' : P_j \cap P_{j'} = \emptyset$ and $R = \bigcup_{j=1}^m P_j$. Furthermore, for two states σ and $\hat{\sigma}$, we write $\Delta_{\mathcal{P}}(\sigma, \hat{\sigma}) := |\{P \in \mathcal{P} \mid P(\sigma) \neq P(\hat{\sigma})\}|$ for the number of partitions in $\mathcal{P}$ that have different values between states σ and $\hat{\sigma}$.

Figure 1a illustrates a partitioning $\mathcal{P} = \{P_1, P_2\}$ where $P_1 = \{r_1, r_2\}$ and $P_2 = \{r_3\}$.

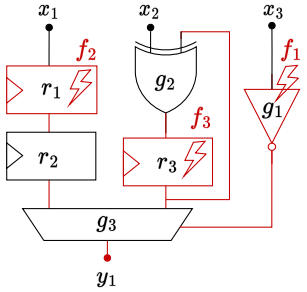
2.2 Fault Injection Attacks

As mentioned in the introduction, attackers can cause faults in the computation. In the following, we formalize the transient fault model, fault attacks, and an attacker goal when attacking a system.

Definition 5 (Transient Fault Model)**.** Let $\mathcal{C} = (G, W)$ be a circuit. A *transient fault model* for circuit $\mathcal{C}$ is characterized as a set of pairs $\mathcal{F} \subseteq G \times U$, with $U = \{x \mapsto 0, x \mapsto 1, x \mapsto \neg x\}$. Each fault $(g, u) \in \mathcal{F}$ describes a potential transient fault with the *fault location* $g \in G$ and the *fault effect* $u \in U$, which encompasses bit-reset, bit-set, and bit-flip fault effects.



(a) Circuit partitioning.



(b) Fault propagation.

Figure 1: Simple circuit examples.

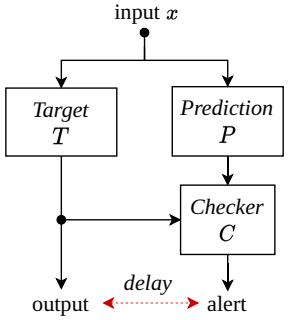


Figure 2: CED scheme.

Here, $\mathcal{F}$ describes which gates in the hardware circuit can be faulted with which types of faults. While in general $\mathcal{F} = G \times U$, Definition 5 also allows for cases where the attacker either cannot fault certain gates due to protection or infeasibility, or can only introduce specific kinds of faults due to circuit technology or fault injection method.

Definition 6 (Fault Attack)**.** Let $\mathcal{C} = (G, W)$ be a circuit, $(\sigma_i)_{i=1}^n$ be an execution trace of $\mathcal{C}$, and $\mathcal{F}$ be a fault model. A *fault attack* $\mathbf{F} \subseteq \mathcal{F} \times [1, n]$ is a set of timed faults injected into circuit $\mathcal{C}$ with *attack order* $|\mathbf{F}|$. $\mathbf{F}$ causes a faulty execution trace $\left(\sigma_i^{\mathbf{F}}\right)_{i=1}^n$, where each fault $(g, u, j) \in \mathbf{F}$ causes gate g to compute $u \circ g$ at clock cycle j. Furthermore, we write $\mathbf{F}_J = \{(g, u, j) \in \mathbf{F} \mid j \in J\}$ for the part of $\mathbf{F}$ whose faults are in clock cycles $J \subseteq [1, n]$.

Figure 1b illustrates a fault attack on a simple circuit and different kinds of consequences. Here, fault $f_1 = (g_1, u, j)$ has *immediate consequences* i.e., in the same clock cycle, on the combinational gates g_1 and g_3, and the output y_1. Fault $f_2 = (r_1, u, j)$ on the register r_1 propagates in the circuit and has *delayed consequences* on r_2, g_3, and y_1 at clock cycle $j + 1$. Fault f_2 can also have *no consequences* if the effect u does not induce a different value or if the mux g_3 does not select the output from r_2. Finally, $f_3 = (r_3, u, j)$ illustrates a specific case of delayed consequences as the fault can stay *hidden* in the register r_3 for an unknown amount of time without being propagated to the output y_1 according to the value of x_3, respectively g_1.

From a security perspective, an attacker wants to perform a fault attack on a circuit to create an exploit. Definition 7 formalizes how we model the goals of such an attacker.

Definition 7 (Attacker Goal)**.** Let $\mathcal{C} = (G, W)$ be a circuit with inputs $I \subseteq G$ and registers $R \subseteq G$. Furthermore, let $\mathcal{F}$ be a fault model. An *attacker goal* is a Boolean predicate φ over circuit states determining whether they are desirable, i.e., $\varphi : \{0,1\}^{|I \cup R|} \to \{0,1\}$. An attacker can reach goal φ at attack order k if they can find a fault attack $\mathbf{F} \subseteq \mathcal{F} \times [1, n]$, with $|\mathbf{F}| \leq k$, such that the resulting execution trace $\left(\sigma_i^{\mathbf{F}}\right)_{i=1}^n$ fulfills $\varphi\left(\sigma_n^{\mathbf{F}}\right) = 1$.

For example, an attack on a CPU circuit can target a specific program counter value to determine if a given sequence of instructions can be executed. Alternatively, the attack may involve a memory address to detect if sensitive data can be read/written.

2.3 Concurrent Error Detection Schemes

Concurrent Error Detection schemes (CEDs) attempt to protect a system against fault attacks using spatial redundancy [MM00]. Figure 2 depicts such a scheme where the *target* function T produces an output $T(x)$ for a given input x, while the *prediction* function P independently generates a predicted characteristic of the output based on the input x, and the *checker* function compares the outputs and raises an *alert* signal on a mismatch.

In its simplest form, the prediction circuit is a duplication of the target and the checker simply compares for equality [KWMK02]. Alternatively, P can also be implemented with error detection codes [BBK+03, AMR+20]. Some implementations also introduce a *delay* between the operation of the target and the prediction functions [VM02, MP23]. This delay has the advantage of increasing the practical difficulty of faulting both functions but introduces a circuit area overhead due to buffering. Definition 8 formalizes CED schemes.

Definition 8 ((d, A)-CED)**.** A circuit $\mathcal{C} = (G, W)$ with outputs O implements a (d, A)-CED when its outputs are divided into alert signals $A \subseteq O$ with associated delay of d clock cycles and primary outputs $O' = O \setminus A$. Without loss of generality, we say that $\mathcal{C}$ raises an alert at clock cycle i if $A(\sigma_i) \neq (0, \ldots, 0)$, which we will write $A(\sigma_i) \neq 0$ for brevity.

The definition of a *fault-secure* CED, first introduced in [SS92], tends to generalize in the security field, particularly within the gadgets domain, under the name *k-order active security* [DN20]. Definition 9 formalizes *k-fault security*, extending the previous definitions to consider a possible detection delay.

Definition 9 (k-fault secure (d, A)-CED)**.** Let $\mathcal{C} = (G, W)$ be a circuit implementing a (d, A)-CED, $(\sigma_i)_{i=1}^{n+d}$ be an arbitrary execution trace of length $n + d$, $\mathcal{F}$ be a fault model and k be the attack order. We say that the (d, A)-CED is *k-fault secure* against the fault model $\mathcal{F}$ if and only if, $\forall n \in \mathbb{N}^*$,

$$\begin{aligned} &\forall (\sigma_i)_{i=1}^{n+d},\ \forall \mathbf{F} \subseteq \mathcal{F} \times [1, n+d],\ |\mathbf{F}| \leq k : \\ &\left(\bigwedge_{i=1}^{n+d} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) = 0 \right) \Longrightarrow \left(\bigwedge_{i=1}^{n} O'(\sigma_i) = O'\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) \right). \end{aligned} \tag{1}$$

Intuitively, Definition 9 says that k-fault security against fault model $\mathcal{F}$ guarantees that whenever there are no alerts in the first $n + d$ clock cycles, the primary outputs are correct up to clock cycle n. Since this must hold for all executions of arbitrary length, we can infer that an alert is raised at most d cycles after a corrupted primary output. A delay $d = 0$ implies an immediate detection, whereas $d = 2$ means the alert is raised up to two cycles after the corrupted output.

Proving the k-fault security of a (d, A)-CED against the fault model $\mathcal{F}$ often relies on bounded equivalence checking [RSS+21, NOV+22]. As described later in Figure 4c, this approach considers a golden trace $(\sigma_i)_{i=1}^{n+d}$ and a faulty trace $\left(\sigma_i^{\mathbf{F}}\right)_{i=1}^{n+d}$ starting from the same initial state σ_1. The k-fault security is ensured by checking that the two traces have the same outputs at each state, assuming no alert is raised. However, as illustrated in Figure 1b, the duration of the fault propagation is not always known *a priori* and a bound n is difficult to find as faults can stay hidden in the circuit indefinitely. Consequently, bounded techniques provide guarantees assuming the fault propagation bound n, but cannot prove the k-fault security in the general case, and may struggle as the checking complexity increases with n.

The theory and methods described in this paper also apply to correction-based countermeasures as these protections are equivalent to a (d, A)-CED scheme without alert signals or delay, i.e., $A = \emptyset$ and $d = 0$.

3 Co-Verification Methodology

This section introduces our hardware/software co-verification methodology, which analyzes the system's robustness against fault injections. The methodology, illustrated in Figure 3, consists of two steps. In the first step, highlighted in the top box of Figure 3, we evaluate the security of the implemented CED schemes against faults at order k. The second step, depicted in the bottom box of Figure 3, is dedicated to the full system verification.

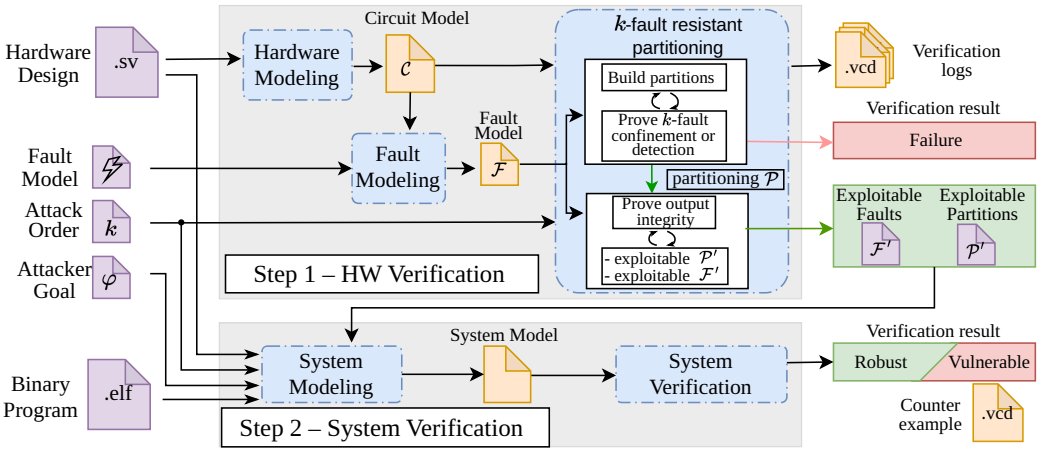


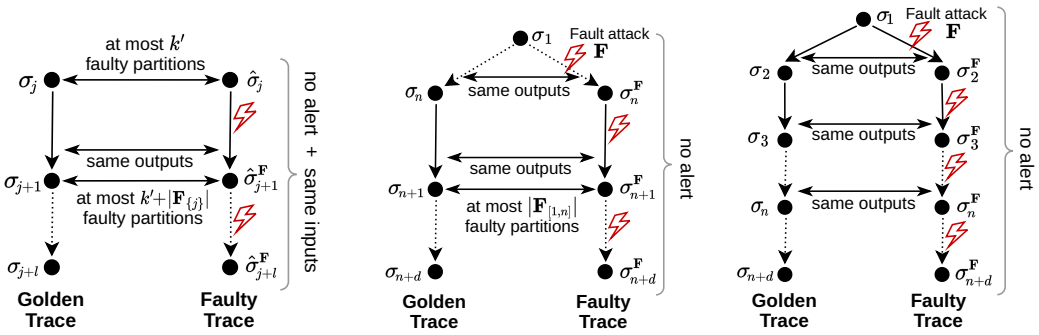
Figure 3: Co-verification methodology to evaluate SW/HW systems against faults attacks.

3.1 Step 1 — Hardware Verification Flow

The hardware verification step can be done either at the Register Transfer Level (RTL) or the netlist level, i.e., after circuit synthesis. However, the latter ensures that the effects of the synthesis on the countermeasures' implementation are captured and that the analysis is performed on a circuit model as close as possible to the tape-out. The input hardware design provided in RTL is thus converted, after synthesis, to a cycle-accurate bit-accurate circuit model $\mathcal{C}$ (Definition 1). The fault model $\mathcal{F}$ describing all possible fault injections is then derived from the input fault model and the produced circuit model.

The analysis of the k-fault security of circuit $\mathcal{C}$ is performed by formally verifying the *k-fault-resistant partitioning* property using an inductive approach to provide unbounded guarantees. In the next section, we formally define this invariant and prove it implies the k-fault secure property of the design. Informally, this property holds under two conditions. First, circuit outputs are correct under any k fault injections that do not raise an alert. Second, the sequential elements of the circuit can be partitioned such that any k fault injections in the circuit are either detected or confined in partitions. *Fault confinement* in partitions means that no fault in a partition can propagate to some partitions without being detected. Note that the effects of a fault injection can freely propagate in a partition without further consequences. Fault confinement ensures that the injection of k faults cannot corrupt more than k partitions without being detected. Therefore, a k-fault-resistant partitioning necessarily has at least $k+1$ partitions to ensure fault detection at attack order k. Indeed, with k partitions or less, all the partitions could be corrupted, preventing detection.

The verification of the k-fault-resistant partitioning is performed by first building iteratively a partitioning that ensures fault confinement or detection for the attack order k and the fault model $\mathcal{F}$. When this construction fails, the user can inspect the verification logs to understand the reason for the failure. Once a suitable partitioning $\mathcal{P}$ is built, a second step verifies iteratively until success that the partitioning $\mathcal{P}$ also ensures the outputs' integrity for the attack order k and the fault model $\mathcal{F}$. When the verification fails, the faults that lead to outputs' corruption are added to a set $\mathcal{F}'$, denoted set of *exploitable faults*. Similarly, the partitions targeted by the faults are added to the set $\mathcal{P}'$ that contains the partitions whose corruption by faults alters outputs' integrity. We refer to these partitions as *exploitable partitions* in the remainder. This growing set $\mathcal{F}'$ is excluded from the fault set $\mathcal{F}$ considered for the next verifications. Also, no fault can be injected in registers belonging to a partition of $\mathcal{P}'$ in the next iterations. The verification eventually succeeds and outputs the sets $\mathcal{F}'$ and $\mathcal{P}'$.



(a) k-fault-resistant partitioning. **(b)** Inductive property. **(c)** k-fault secure (d, A)-CED.

Figure 4: Overview of different properties a circuit implementing CED can fulfill, where property **(a)** is the strongest and implies property **(b)**, which in turn implies property **(c)**.

Note that Step 1 is independent of the executed program. Consequently, it only has to be run once, and the verification results can be used for multiple software evaluations. In the case where no exploitable faults or partitions are identified, the circuit is robust to k faults unconditionally of the executed software. There is no need to perform Step 2.

3.2 Step 2 — System Verification Flow

Step 2 is a *system verification* process that analyzes program executions to detect if an attacker can reach his goal. This verification is performed by considering only the faults that have not been formally proven, at Step 1, to be detected by hardware protections.

The software and hardware co-verification takes as input the hardware design, a binary program, the attack order, the attacker goal, and the set of exploitable faults $\mathcal{F}'$ and partitions $\mathcal{P}'$ computed in Step 1. The *system modeling* process combines all these elements in a single model. The generated model maps the software execution on the underlying hardware whose behavior is modified by the possible faults in $\mathcal{F}'$ and $\mathcal{P}'$. Exploitable fault locations derived from $\mathcal{F}'$ and $\mathcal{P}'$ help to select the best-suited abstraction level during the modeling step. For example, an ISA-level model is sufficient when only the values read from memory can be corrupted. When the hardware description is necessary, the system modeling process can optimize sub-circuits if faults in $\mathcal{P}'$ and $\mathcal{F}'$ do not target them. Indeed, there is no need to consider every micro-architectural detail of protected parts of the circuit for which a behavioral modeling is enough.

As a result of the analysis, the verification step reports whether the system is robust against the considered attacker, and produces counterexamples as Value Change Dump (VCD) files if vulnerabilities have been found. Counterexamples help the user to understand where the fault was injected and how it propagates in the system to create the vulnerability.

4 Fault-Resistant Partitioning

This section formally defines the notion of *k-fault-resistant partitioning* before proving it implies the *k-fault security* of a circuit implementing CED. Afterward, we provide an algorithm that automatically identifies such a partitioning and proves its k-fault resistance.

4.1 Formal Definition of k-Fault-Resistant Partitioning

As discussed in Section 2.3, directly proving that a circuit implementing a CED fault countermeasure provides k-fault security is not always feasible. As depicted in Figure 4c,

direct bounded proofs would have to unroll both the golden and faulty versions of the circuit an *a priori* unknown number of times, until reaching a completeness threshold. Considering that transient faults can often linger within the state of the circuit indefinitely, this methodology quickly becomes intractable. However, all is not lost and it is possible to find simple properties provable with a small fixed bound that implies the k-fault security of a CED implementation, circumventing such problems. In the following, we define such a property called *k-fault-resistant partitioning* and prove it guarantees the k-fault security of a (d, A)-CED.

Definition 10 (k-Fault-Resistant Partitioning)**.** Let $\mathcal{C} = (G, W)$ be a circuit implementing a (d, A)-CED. Let $j \in \mathbb{N}^*$ be an arbitrary offset and let $(\sigma_i)_{i=j}^{j+l}$ and $(\hat{\sigma}_i)_{i=j}^{j+l}$ be two arbitrary execution traces of length $l+1$, where $l = \max(1, d)$. Finally, let $\mathcal{P}$ be a partitioning of circuit $\mathcal{C}$, $\mathcal{F}$ be a fault model, and let $k \in \mathbb{N}^*$ be an attack order. We say that $\mathcal{P}$ is a *k-fault-resistant partitioning* of $\mathcal{C}$ against the fault model $\mathcal{F}$ if and only if

$$\begin{aligned}
&\forall\, (\sigma_i)_{i=j}^{j+l},\ (\hat{\sigma}_i)_{i=j}^{j+l},\ \mathbf{F} \subseteq \mathcal{F} \times [j, j+d],\ k' \in \mathbb{N},\ |\mathbf{F}| + k' \leq k : \\
&\left(\bigwedge_{i=j}^{j+d} I\left(\sigma_i\right) = I\left(\hat{\sigma}_i\right)\right) \wedge \left(\Delta_{\mathcal{P}}\left(\sigma_j, \hat{\sigma}_j\right) \leq k'\right) \wedge \left(\bigwedge_{i=j}^{j+d} A\left(\hat{\sigma}_i^{\mathbf{F}_{[j,i]}}\right) = 0\right) \implies \\
&\left(\Delta_{\mathcal{P}}\left(\sigma_{j+1}, \hat{\sigma}_{j+1}^{\mathbf{F}_{\{j\}}}\right) \leq k' + \left|\mathbf{F}_{\{j\}}\right|\right) \wedge \left(O'\left(\sigma_j\right) = O'\left(\hat{\sigma}_j^{\mathbf{F}_{\{j\}}}\right)\right).
\end{aligned} \tag{2}$$

Similar to k-fault security, the definition of k-fault-resistant partitioning also considers two execution traces $(\sigma_i)_{i=j}^{j+l}$ and $(\hat{\sigma}_i)_{i=j}^{j+l}$ where the former is the reference trace and a fault attack targets the latter. In Equation (2), the implication's left-hand side can be considered as assumptions under which the design must guarantee that the right-hand side holds. First, it is assumed that both execution traces have the same inputs, their initial states σ_j and $\hat{\sigma}_j$ differ in at most k' partitions at clock cycle j, and no alerts are triggered in the faulty trace $\left(\hat{\sigma}_i^{\mathbf{F}}\right)_{i=j}^{j+d}$. Intuitively, this situation represents two execution traces of circuit $\mathcal{C}$, depicted in Figure 4a, processing the same inputs but where at most k' partitions have a different state due to faults injected before clock cycle j. In addition, we consider a fault attack $\mathbf{F}$ with an attack order $|\mathbf{F}| \leq k - k'$ modifying execution trace $(\hat{\sigma}_i)_{i=j}^{j+l}$ between clock cycles j and $j+d$ but without triggering any alert signal. The right-hand side of Equation (2) specifies the two characteristics a k-fault-resistant partitioning must fulfill. First, the number of newly corrupted partitions is less than or equal to $\left|\mathbf{F}_{\{j\}}\right|$ which is equal to the number of faults introduced by the fault attack $\mathbf{F}$ at clock cycle j (*k-fault confinement*). Newly corrupted partitions are evaluated after one transition, i.e., at clock cycle $j+1$, since faults have delayed consequences on registers. Second, the circuit's primary outputs must be identical at clock cycle j between the two execution traces since faults have immediate consequences on the outputs (*outputs' integrity*).

Theorem 1 states that a circuit with a k-fault-resistant partitioning is necessarily also k-fault secure.

Theorem 1 (k-fault-resistant partitioning implies k-fault security)**.** *Let $\mathcal{C} = (G, W)$ be a circuit implementing a (d, A)-CED and let $\mathcal{F}$ be a fault model targeting the circuit. If there exists a* k-fault-resistant partitioning *$\mathcal{P}$ against $\mathcal{F}$ then $\mathcal{C}$ is k-fault secure.*

Proof. To prove that Definition 10 (Figure 4a) implies Definition 9 (Figure 4c), we first show that it implies a stronger inductive property (Figure 4b), which in turn, implies Definition 9. Figure 4 shows the proof intuition, where k' faulty partitions in the initial state $\hat{\sigma}_i$ of (4a) correspond to k' faults injected during the previous clock cycles of the same execution trace $\left(\sigma_i^{\mathbf{F}}\right)_{i=1}^{n}$ in (4b). The complete proof is given in Appendix A. □

Theorem 1 provides a new strategy to prove the k-fault security of a (d, A)-CED circuit, giving unbounded guarantees on the fault attack consequences. Although k-fault-resistant

Algorithm 1: Build and prove a k-fault-resistant partitioning of circuit $\mathcal{C}$.

Input: a circuit $\mathcal{C} = (G, W)$ implementing a (d, A)-CED, a fault model $\mathcal{F} \subseteq G \times U$, an attack order k, and an initial partitioning $\mathcal{P}$

Output: On success, returns a k-fault-resistant partitioning $\mathcal{P} = \{P_j\}_{j=1}^{m}$, a set of exploitable fault injections $\mathcal{F}' \subseteq \mathcal{F}$, and a set of exploitable partitions $\mathcal{P}' \subseteq \mathcal{P}$.

Create symbolic executions $(\sigma_i)_{i=1}^{l+1}$ and $(\hat{\sigma}_i)_{i=1}^{l+1}$, with $l = \max(1, d)$;
Create symbolic fault attack $\mathbf{F} \subseteq \mathcal{F} \times [1, d+1]$;
$\psi_{\text{InsEqAndNoAlert}} \leftarrow \left(\bigwedge_{i=1}^{d+1} I\left(\sigma_i\right) = I\left(\hat{\sigma}_i\right)\right) \wedge \left(\bigwedge_{i=1}^{d+1} A\left(\hat{\sigma}_i^{\mathbf{F}}\right) = 0\right)$;
Procedure *BuildPartitioning*: `/* Build a k-fault confining partitioning */`
  for k' **from** 0 **to** k **do**
    $\psi_{\text{MoreInfected}} \leftarrow (\Delta_{\mathcal{P}}(\sigma_1, \hat{\sigma}_1) \leq k') \wedge (|\mathbf{F}| \leq k - k') \wedge \left(\Delta_{\mathcal{P}}(\sigma_2, \hat{\sigma}_2) > k' + \left|\mathbf{F}_{\{1\}}\right|\right)$;
    while $(\psi_{InsEqAndNoAlert} \wedge \psi_{MoreInfected})$ *is SAT* **do**
      $\mathcal{P}_{\text{Init}} \leftarrow \{P \in \mathcal{P} \mid P(\sigma_1) \neq P(\hat{\sigma}_1)\}$;
      $\mathcal{P}_{\text{Next}} \leftarrow \left\{P \in \mathcal{P} \mid P(\sigma_2) \neq P\left(\hat{\sigma}_2^{\mathbf{F}}\right)\right\}$;
      $\mathcal{P} \leftarrow \text{merge}(\mathcal{P}, \mathcal{P}_{\text{Init}}, \mathcal{P}_{\text{Next}})$;
if $|\mathcal{P}| \leq k$ **then return failure**;
$\mathcal{F}' \leftarrow \{\}$, $\mathcal{P}' \leftarrow \{\}$;
Procedure *CheckIntegrity*: `/* Find faults that compromise outputs */`
  for k' **from** 0 **to** k **do**
    $\psi_{\text{NoFaultsOnForbidden}} \leftarrow \left(\bigwedge_{P \in \mathcal{P}'} P(\sigma_1) = P(\hat{\sigma}_1)\right) \wedge (\mathbf{F} \cap \mathcal{F}' \times [1, d+1] = \emptyset)$;
    $\psi_{\text{OutsBad}} \leftarrow (\Delta_{\mathcal{P}}(\sigma_1, \hat{\sigma}_1) \leq k') \wedge (|\mathbf{F}| \leq k - k') \wedge \left(O'(\sigma_1) \neq O'\left(\hat{\sigma}_1^{\mathbf{F}}\right)\right)$;
    while $(\psi_{InsEqAndNoAlert} \wedge \psi_{NoFaultsOnForbidden} \wedge \psi_{OutsBad})$ *is SAT* **do**
      $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{P \in \mathcal{P} \mid P(\sigma_1) \neq P(\hat{\sigma}_1)\}$;
      $\mathcal{F}' \leftarrow \mathcal{F}' \cup \{(g, u) \in G \times U \mid \exists j, (g, u, j) \in \mathbf{F}\}$;
return $(\mathcal{P}, \mathcal{P}', \mathcal{F}')$;

partitioning is only a sufficient condition for k-fault security, it significantly simplifies the endeavor of the proof since the circuit is only unrolled $max(1, d)$ times, compared to the bounded equivalence checking approach. In converse, a k-fault secure circuit may not fulfill the k-fault-resistant partitioning property. Our approach is not sufficient to highlight genuine vulnerabilities, and counterexamples require further analysis as false positives exist. The following section provides an algorithm to build such a k-fault-resistant partitioning.

4.2 Algorithm to Identify a k-Fault-Resistant Partitioning

Algorithm 1 describes how to identify a circuit partitioning $\mathcal{P}$ resistant to k fault injections using SAT solving. It takes as input a circuit model $\mathcal{C}$, a fault model $\mathcal{F}$, an attack order k, and an initial partitioning $\mathcal{P}$. Algorithm 1 comprises two main procedures, *BuildPartitioning* and *CheckIntegrity*, where the former builds a partitioning ensuring k-fault confinement or detection, and the latter finds all remaining fault injections compromising the integrity of outputs. Eventually, the algorithm either returns a k-fault-resistant partitioning $\mathcal{P}$ with a set of assumptions under which the circuit is k-fault secure or fails to find such a partitioning and provides counterexamples detailing what happened.

The initial partitioning can be chosen freely, but for an initial run of the algorithm, it should be chosen as $\mathcal{P} = \{\{r\} \mid r \in R\}$, where each register $r \in R$ belongs to a separate partition. In subsequent runs of the algorithm, one can set the partitioning $\mathcal{P}$ to one that was previously computed. At the start, the algorithm first creates two symbolic execution traces $(\sigma_i)_{i=1}^{l+1}$ and $(\hat{\sigma}_i)_{i=1}^{l+1}$ of length $l+1$, with $l = \max(1, d)$, and a symbolic fault attack $\mathbf{F}$ that describes all possible faults an attacker can induce.

Procedure *BuildPartitioning* iteratively analyzes whether the current partitioning $\mathcal{P}$ guarantees that, whenever k' partitions are compromised and there are $\left|\mathbf{F}_{\{1\}}\right|$ new faults in the first clock cycle, there are at most $k' + \left|\mathbf{F}_{\{1\}}\right|$ compromised partitions in the second clock cycle (line 6). It does this by iterating through all combinations of k' and $|\mathbf{F}| = k - k'$ (line 5) and asking a SAT solver whether there are execution traces $(\sigma_i)_{i=1}^{l+1}$ and $(\hat{\sigma}_i)_{i=1}^{l+1}$ as well as a concrete fault attack $\mathbf{F}$ where the left-hand side of (2) is true but the first part of the right-hand side, i.e., $\left(\Delta_{\mathcal{P}}\left(\sigma_2, \hat{\sigma}_2\right) \leq k' + \left|\mathbf{F}_{\{1\}}\right|\right)$, is false (line 7). If the solver finds such an example, i.e., the formula is SAT, the procedure gathers the compromised partitions $\mathcal{P}_{\text{Init}}$ and $\mathcal{P}_{\text{Next}}$ from respectively the first and second clock cycle, and then merges partitions from $\mathcal{P}_{\text{Next}}$ until there are only $k' + \left|\mathbf{F}_{\{1\}}\right|$ left, while avoiding merges between the partitions also present in $\mathcal{P}_{\text{Init}}$ (lines 8 to 10). As $|\mathcal{P}|$ decreases at each iteration, procedure *BuildPartitioning* converges to a fixed point where partitioning $\mathcal{P}$ fulfills the relevant part of (2), and the solver must return UNSAT. After all k' are analyzed, the procedure concludes.

After *BuildPartitioning* finishes, the algorithm checks whether the partitioning has less than k partitions (line 11), as such a partitioning cannot guarantee the output integrity for the second procedure (cf. Section 3). Procedure *BuildPartitioning* may fail for one of the three following reasons: *(i)* the circuit $\mathcal{C}$ has some flaws and is not k-fault secure, *(ii)* there is no k-resistant partitioning even if $\mathcal{C}$ is k-fault secure (cf. Section 4.1), or *(iii)* the merging heuristics fails to build a k-resistant partitioning even though one exists. We generate log files for each counterexample given by the solver and the corresponding partition merges the algorithm performs. In general, the logs are invaluable for understanding why a design might be insecure, but this analysis must be performed manually.

For higher-order fault analysis, one should start with $k = 1$ and iteratively feed the found partitioning $\mathcal{P}$ back into the algorithm for the next higher k as a k-fault resistant partitioning must be $(k-1)$-fault resistant.

Procedure *CheckIntegrity* iteratively determines the sets $\mathcal{P}'$ and $\mathcal{F}'$ of partitions and locations where faults can compromise the output integrity. The procedure iteratively verifies if the partitioning $\mathcal{P}$ guarantees outputs' integrity in the presence of k' faulty partitions and $k - k'$ new faults while not targeting the known-to-be exploitable partitions $\mathcal{P}'$ and fault locations $\mathcal{F}'$ identified in previous iterations (lines 15 and 16). Whenever the solver returns SAT, it means that it found a new set of fault locations and initially corrupted partitions that compromises output integrity and must be added to $\mathcal{P}'$ and $\mathcal{F}'$, respectively (lines 17 to 19). If the solver returns UNSAT instead, it means that the partitioning $\mathcal{P}$ is proven k-fault secure assuming there are no faults on the $\mathcal{P}'$ and $\mathcal{F}'$.

5 Implementation

This section details how we implement the methodology introduced in Section 3. First, we describe Step 1 to formally analyze the k-fault security of a CED circuit using k-fault-resistant partitioning. Second, we illustrate how potential remaining faults, undetected by the hardware countermeasures, are integrated into a co-verification framework.

5.1 Step 1 — Hardware Verification Flow

The hardware design is converted into a bit-level netlist using the synthesis tool Yosys [Wol] to match the circuit model given in Definition 1. Additionally, input, output and alert signals must be provided to define the CED circuit to be analyzed.

We then rely on the C++ API of the CaDiCaL SAT solver [BFFH20] for the formal analysis described in Algorithm 1. Circuit elements are encoded with Boolean variables and execution traces $(\sigma_i)_{i=1}^{l+1}$ and $(\hat{\sigma}_i)_{i=1}^{l+1}$ are modeled unrolling the circuit $l = \max(1, d)$ times. Fault injections are applied to execution traces using new Boolean variables to

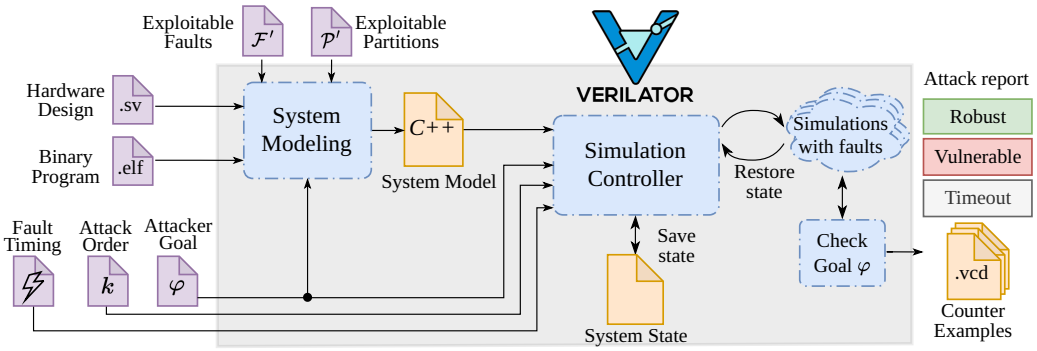


Figure 5: Software/Hardware co-verification flow (Step 2) using Verilator.

control the effect of faults. Finally, assumptions ψ made by Algorithm 1 are provided to the SAT solver to check their satisfiability. CaDiCaL is used in incremental mode to update assumptions as we build the circuit partitioning. Log files and VCD waveforms are generated to keep track of successive iterations, understand how the algorithm builds the circuit partitioning, and analyze why the proof may fail. The implementation is about 4000 lines of code and is publically available[3].

5.2 Step 2 — System Co-Verification

Figure 5 illustrates our simulation-based co-verification framework. First, the *system modeling* step relies on the open-source tool Verilator [Sny] to convert the hardware design and the binary program into a cycle-accurate C++ model. The system modeling also takes as input the sets $\mathcal{F}'$ and $\mathcal{P}'$ computed in Step 1 to determine the remaining fault locations in gates and registers. Verilator optimizes the generated model for simulation performance reasons, and the effectiveness of optimizations depends on the number of fault locations. Then, the *simulation controller* simulates the circuit with a maximum of k faults. The fault timing specifies the cycles where the faults must be injected during the simulation. The predicate φ is evaluated on the system state at each clock cycle to determine if the attacker can reach its goal. For example, such a predicate may evaluate the program counter value to analyze the control flow or look at a value in the memory or in the register file. Simulations are run in parallel. Finally, the framework provides an attack report for each fault attack evaluated. The report classifies the attack between i) *robust*, i.e., the fault attack does not fulfill φ and the simulation terminates as expected, ii) *vulnerable*, i.e., φ has been reached, and iii) *timeout*, i.e., neither the attacker goal nor the normal program exit point has been reached and the simulation stops after a timeout. The timeout is computed according to the program length. Verilator generates logs such as the ISA states or VCD waveforms to understand where the faults were injected and how they propagate in the system to create the vulnerability.

To speed up the analysis, we adapted a simulator feature to save the system state in a file. The state is restored for each new verification, which avoids simulating irrelevant parts of the program for the fault analysis. In addition, we used the Verification Procedural Interface (VPI), supported by Verilator, to observe the circuit state and compute φ or to inject faults on circuit elements retrieved according to their hierarchical names.

This co-verification framework has the same limitations as simulation-based analysis tools. It is not exhaustive on program inputs and does not provide security guarantees in the general case. In addition, a timeout is needed to stop the simulation when the control flow has been modified by the attack but without reaching the attacker goal.

[3] https://github.com/CEA-LIST/Fault-Resistant-Partitioning

Table 1: Evaluation of Skinny-64 and AES-128 circuits using k-fault-resistant partitioning.

Circuit Characteristics			Faults		Algorithm 1 Performance		Results	
Name	**Size (GE)**	**Regs (#)**	**Loc. (#)**	**Order k**	***Build Partitioning* (s)**	***Check Integrity* (s)**	**Partitions (#)**	**Exploitable Faults (#)**
Skinny-64 red-1	3 270	235	1 707	1	1.18	0.043	235	64 outputs
Skinny-64 red-3	4 163	305	2 959	1	1.32	0.052	305	
				2	9.06	0.324	305	
Skinny-64 red-4	6 316	341	3 417	1	2.48	0.127	341	
				2	10.23	0.404	341	
				3	38.50	0.693	341	+ 335 in checker
AES-128 red-1	20 532	427	16 262	1	597	1.20	427	129 outputs
AES-128 red-4	29 092	527	23 390	1	1 073	2.14	527	
				2	13 983	2.28	527	
AES-128 red-5	32 284	561	26 166	1	1 471	2.32	561	
				2	17 376	2.57	561	
				3	201 272	2.79	561	+ 22 in checker

6 Experimental Validation on Impeccable Circuits

This section validates our methodology against prior work on formal verification of CED schemes. We evaluate the robustness of Skinny-64 and AES-128 implementations from Impeccable Circuits [AMR+20] protected with code-based CEDs against faults attacks. Although providing unbounded guarantees on cryptographic circuits is not as crucial as on a CPU, i.e., their operation usually takes a few clock cycles, these case studies allow us to compare against related work, e.g., FIVER [RSS+21], as no similar work exists on CPUs.

Performance-wise, our algorithm successfully proves the 2-fault security of AES-128 (resp. Skinny-64) implementation in less than 4 h (resp. 10 s) using an Intel Core i7-1185G7 laptop, as shown in Table 1. In comparison, the authors of FIVER reported 130 h to prove the 2-fault security of the same AES circuit using an Intel Xeon server. Our approach also reports that circuit inputs and outputs can be faulted as they are not protected with EDC.

Our methodology was also able to assess the 3-fault security of these circuits, an attacker model unreachable by state-of-the-art tools. However, our fault analysis first fails to build a circuit partitioning. The manual investigation of the logs produced during procedure *BuildPartitioning* shows that three faults defeat the EDC protection by targeting simultaneously: 1) the original function, 2) the redundant function, and 3) the checker disabling the alert at a specific clock cycle. During the following clock cycles, the injected faults propagate and lead to collisions that are undetected by the checker mechanism. Explaining the exploitable faults in more detail is beyond the scope of this paper. However, assuming that the exploitable faults identified in the checker cannot be corrupted, i.e., 335 bits in Skinny-64, and only 22 bits in AES-128 where fewer collisions exist, we prove the 3-fault security of AES (resp. Skinny) in 55 h (resp. 39 s). For each algorithm, we reproduced the attack for a full encryption to ensure that the reported counterexamples are not false positives. Attack scenarios are given in Appendix B.

7 Evaluation on OpenTitan

In this section, we apply our methodology to analyze the resilience of a development version of the Secure Ibex processor and determine whether an attacker can exploit potential hardware vulnerabilities in three different programs running on the OpenTitan platform. First, we evaluate the robustness of the hardware countermeasures implemented in the Secure Ibex processor. Second, we leverage the hardware verification results to analyze

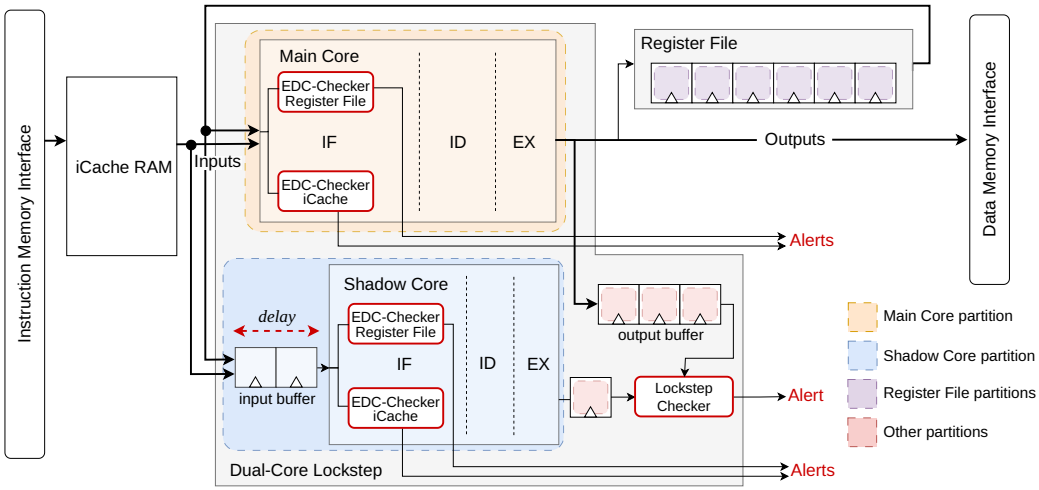


Figure 6: Secure Ibex countermeasures and partitioning obtained with Algorithm 1.

whether the identified vulnerabilities can be exploited in different programs. Third, we provide a hardware fix for the vulnerability discovered and re-evaluate the security.

OpenTitan Threat Model. The OpenTitan project [JRR+18] provides an open-source secure element design [Opeb]. Internally, OpenTitan uses the 32-bit RISC-V Ibex processor, implements several hardware-based countermeasures, and provides hardened software and secure boot. Globally, these countermeasures aim to protect the chip's confidentiality, integrity, and authenticity [Opea]. We consider an attacker having physical access to the platform capable of interfering with its operation by performing fault injection attacks (*attacker goal*). We consider a single transient bit-flip everywhere in the microarchitecture (*fault model*), which is in line with the protection level provided by the countermeasures.

Secure Ibex Hardware Countermeasures. Our hardware analysis focuses on the secure configuration of the Ibex core [IBE], which uses different spatial Concurrent Error Detection (CED) schemes (Figure 6). The *dual-core lockstep* (DCLS) mechanism instantiates the Ibex core twice, compares the outputs between the main core and the shadow core, and triggers an alert signal on a mismatch. To increase the protection against faults, the shadow core inputs are delayed for d cycles, where d is fixed at synthesis time. Our evaluation uses the default value $d = 2$ but also discusses the results for $d = 3$. Both core instances share the register file, which is protected against faults with Error Detection Codes (EDC) and a write-enable glitch detection mechanism.

7.1 Step 1 — Hardware Verification of Secure Ibex

In the following, we apply our hardware verification methodology individually to the register file and the DCLS before analyzing the entire Ibex core. Table 2 summarizes the area in gate equivalent (GE) for each circuit, provides the number of possible fault locations, and reports verification results and performance using an Intel Xeon Gold 6154 CPU. Our analysis does not consider the sleep mode of the Ibex processor that disables the clock signal, as our circuit model only considers synchronous circuits (Definition 1). First, we focus on $k = 1$ since the countermeasures of Secure Ibex aim to mitigate a single fault before discussing the evaluation results with $k = 2$.

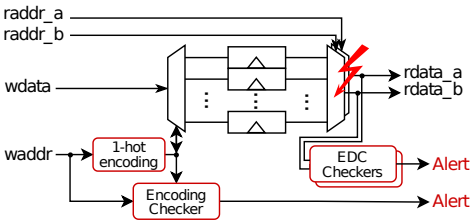


Figure 7: Register file protections and the identified vulnerability.

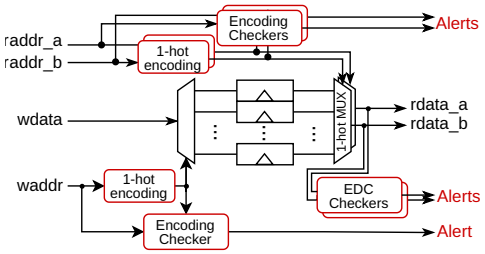


Figure 8: Register file with fixed vulnerability.

Register File Analysis. The register file consists of thirty-two 32-bit registers, each protected by a 7-bit EDC. Register file countermeasures ensure that written data is stored at the correct address (*encoding checker* box in Figure 7) and that read data have not been modified (*EDC checkers* box in Figure 7). Procedure *BuildPartitioning* has proven that a fault injected in the circuit cannot propagate to multiple registers without being detected by the protections. Table 2 reports that each register is an independent partition.

However, procedure *CheckIntegrity* has enumerated 172 fault locations in the combinational logic that lead to the corruption of primary outputs. As shown in Figure 7, the internal mux tree that selects the register to read according to the inputs signals `raddr_a_i` or `raddr_b_i` is not protected. Hence, a single fault in the mux logic can change which register file value is written back to the core. This is not detected by DCLS as the register file is only read once by the main core, and the value is then stored in the input buffer of the shadow core, i.e., both cores retrieve the same faulty register file value (Figure 6). We discuss the mitigation we designed in Section 7.3.

In addition, we also evaluate the register file against a weaker fault model targeting only the sequential logic. The EDC protection claims to be robust against 3 faults injected in the data and our analysis proves it as reported in Table 2.

Dual-Core Lockstep (DCLS) Analysis. At first, procedure *BuildPartitioning* described in Algorithm 1 failed to build a correct partitioning of the DCLS and grouped every register inside the same partition. Counterexamples provided by the analysis showed that the checker mechanism can be disabled when initializing a specific register to 0. This register drives the `enable_cmp_q` signal and is intended to disable the alert during the first d clock cycles after a system reset as the shadow core and the main core produce different outputs because of the delay. Formal verification leverages this register to turn off the protection

Table 2: Evaluation of the Secure Ibex and its modules using k-fault-resistant partitioning.

Circuit Characteristics			Faults		Algorithm 1 Performance			Results		
Name	Size (GE)	Regs (#)	Loc. (#)	Order k	*BuildPartitioning* Iter. (#)	*BuildPartitioning* Time	*CheckIntegrity* Time	Partitions (#)	Exploitable Faults $\mathcal{P}'$ (#)	Exploitable Faults $\mathcal{F}'$ (#)
Register File	12 075	1 326	8 331	1	172	38 s	53 s	1 326	0	172
			1 326[a]	3	1	349 s	344 s	1 326	0	0
Register File with fix	11 913	1 326	8 667	1	1	17 s	73 s	1 326	0	0
			1 326[a]	3	1	135 s	383 s	1 326	0	0
DCLS	117 998	5 918	116 561	1	508	20 h 12	5 h 10	1 108	0	0
				2	11	11 s	—	445	—	—
Secure Ibex (no iCache)	130 194	7 248	125 080	1	1	10 h 45	30 h 50	2 438	0	0 (+172)
				2	48	53 s	—	421	—	—

[a] Restricted fault model targeting the sequential logic only

failing to prove system 1-fault security. In the following, and without loss of generality, we assume this register is initialized to 1 as it should be during the normal processor operation. Faults can still be injected into this register. Nonetheless, this highlights that the whole DCLS security relies on a 1-bit register that can be written to 0 to disable the protection. We reported this finding to the OpenTitan project and provided a security enhancement that got integrated[4] into the project.

Assuming `enable_cmp_q` = 1, our analysis builds 1 108 partitions. The *main core* and the *shadow core* are two of these, while the others are registers that faults cannot corrupt without raising an alert. Figure 6 denotes them as *other partitions*. Building $\mathcal{P}$ takes 508 iterations in 11 h, and then the proof of fault confinement in $\mathcal{P}$ takes 9 h 20 (Table 2). Finally, procedure *CheckIntegrity* proves that the DCLS can detect any single bit-flip in one of the two cores and in its internal comparison logic in 5 h 10.

To observe the influence of the DCLS detection delay on the evaluation, we also carried out experiments with $d = 3$. As a result, the design size increases by 3.1%, the number of registers by 13.4%, and the verification time by 24.6%, since the circuit has to be unrolled once more. The analysis concludes with the same results as with $d = 2$.

Full Ibex Analysis. The full Ibex comprises the DCLS and the register file. The remaining gates are involved in the sleep unit module, which we disabled. First, we assume that the 172 faults already identified in the register file cannot be reproduced here. Then, we reuse the partitions found when verifying the DCLS and the register file modules to initialize Algorithm 1. As a result, procedure *BuildPartitioning* only needs one iteration to prove the fault confinement (Table 2), and our methodology proves the 1-fault security of the full Ibex processor against a single fault injection.

Discussion on Ibex Analysis with $k = 2$. OpenTitan is designed to be 1-fault secure. Two faults are logically not detected when targeting both cores or disabling the 1-bit alert. However, we report in Table 2 how our methodology behaves, with $k = 2$, on an unprotected design. For both the DCLS and the Secure Ibex, procedure *BuildPartitioning* merges most of the previously identified partitions, with $k = 1$, within a few seconds. Multiple partitions remain due to structurally impossible merges or because of non-faultable registers that drive the alert signal directly. Notably, the main core and the shadow core are merged and the resulting partitioning no longer guarantees the output integrity against one fault. We omit the *CheckIntegrity* results as enumerating all the exploitable faults is irrelevant and does not necessarily report genuine vulnerabilities.

7.2 Step 2 — Co-Verification of Programs Running on OpenTitan

In this section, we analyze if the exploitable faults previously identified (register file) can be exploited in an attack on the running software. All co-verifications have been conducted using the framework described in Section 5.2 simulating a complete OpenTitan chip. Our analysis focuses on the secure boot provided by the OpenTitan project. The other evaluated programs are typical fault injection benchmarks [DRPR19, PHB+19, TAC+22], i.e., VerifyPIN and tiny AES that are not provided by the OpenTitan project. Table 3 reports evaluation results and performance using an Intel Xeon Gold 6154 CPU.

7.2.1 Secure Boot

The secure boot process guarantees the integrity and authenticity of the code running on the device after a system reset. The first stage configures the peripherals, sets up the software environment, and also verifies the integrity of the second boot stage, `ROM_EXT`,

[4] https://github.com/lowRISC/ibex/pull/2129

Table 3: Co-verification results on software cases exploiting the register file vulnerability.

Program Characteristics			Fault Characteritics			Analysis Results			Performance	
Name	**Function/ Version**	**Instr. (#)**	**Attacker Goal φ**	**Timing (clock cycles)**	**Loc. (#)**	**Success**	**Fail**	**Timeout**	**Threads (#)**	**Verification Time (s)**
Secure Boot	Mask ROM *signature check*	2 526	φ_{boot_flash}	0 - 1 907	122 048	0	95 238	26 810	8	9 235
Tiny AES	Key Schedule *8th-9th round*	221	φ_{key_sched}	0 - 90	5 760	532	4 666	562	2	458
	AES *7th-8th round*	1 144	φ_{aes}	0 - 610	38 912	4 084	29 477	5 228	8	1 742
VerifyPIN	v0	114	φ_{authen} / φ_{ptc}	0 - 49	3 136	2 / 84	2 890	160	1	145
	v1	121	φ_{authen} / φ_{ptc}	0 - 51	3 264	1 / 89	2 990	185	1	154
	v2	162	φ_{authen} / φ_{ptc}	0 - 91	5 824	1 / 87	5 200	537	1	311
	v3	166	φ_{authen} / φ_{ptc}	0 - 95	6 080	1 / 87	5 456	537	1	309
	v4	189	φ_{authen} / φ_{ptc}	0 - 117	7 488	1 / 95	6 714	679	1	468
	v5	169	φ_{authen} / φ_{ptc}	0 - 97	6 208	0 / 77	5 503	628	1	311
	v6	160	φ_{authen} / φ_{ptc}	0 - 88	5 632	0 / 85	5 019	528	1	264
	v7	187	φ_{authen} / φ_{ptc}	0 - 116	7 424	1 / 61	6 682	681	1	399

stored in Flash memory before booting on it. The second stage of the secure boot provides boot services and verifies the next stage's integrity, i.e., the boot loader (`BL0`) code for the kernel. We focus on verifying the first boot stage, a typical target for fault injection attacks, since it is stored in read-only memory and cannot be modified. We analyze the `rom_verify` function in the `Mask_ROM` code, which is responsible for verifying the authenticity and integrity of the next boot stage. It first computes the digest of the `ROM_EXT` image and checks its RSA signature against the signature stored in the boot manifest.

Attacker Goal. Assuming a malicious `ROM_EXT` code, the attacker wants to bypass the signature check and call the `rom_boot` function, i.e., φ_{boot_flash} : (PC = @`rom_boot`).

Our analysis evaluates faults injected in the `rom_verify` function, assuming that the RSA hardware accelerator (OTBN module) has already computed the signature. Our framework shows that controlling, with a fault, the register file value that is written back is insufficient to bypass the first stage of the secure boot. Even if not detected by the hardware, these faults are captured by the software countermeasures. Hence, the secure boot's signature verification is robust to single bit-flip attacks.

7.2.2 Differential Fault Analysis on tiny AES

Differential fault analysis [BS97] enables adversaries to retrieve the cryptographic key by injecting faults during the AES encryption. These attacks can be performed on hardware or software implementations of AES. As our work focuses on the evaluation of hardened CPUs, we do not analyze the AES driver provided in the OpenTitan cryptography library as it utilizes the AES hardware accelerator. Instead, we port the tiny AES [kok] program, which is not officially provided by the project, to OpenTitan. As previously, we used the framework described in Section 5.2 to inject faults into the register file during the AES execution. We illustrate how an attacker can exploit these faults at the software level by reproducing the requirements of two attacks known from the literature [KQ08, TMA11]. An arbitrary plaintext and symmetric key were used for the analysis.

Attacker Goal. The first attack targets the *key schedule* function to corrupt one byte in the first column of the 9^{th} round key (φ_{key_sched}) [TFY07, KQ08]. The second attack targets the *AES algorithm* to corrupt a single byte in the 8^{th} round state matrix (φ_{aes}) [TMA11].

For each experiment, the fault is injected during the round preceding the round of interest. We observe the 9^{th} round key and the 8^{th} round state matrix stored in the data

memory[5] and compare them against the precomputed reference values to determine if the fault induced a single-byte corruption. Table 3 summarizes our evaluation results for each attack. Our analysis reported 532 successful fault injections over the 5 760 possibilities to satisfy φ_{key_sched}. Similarly, 4 084 successful fault injections were identified over the 38 912 configurations tested to reach φ_{aes}. Inspecting the analysis reports shows that successful fault injections are mainly applied to memory load and store operations.

7.2.3 VerifyPIN

For the last software verification, we focus on the VerifyPIN test suite that we port to the chip as it is not part of the OpenTitan project. This test suite [DPP+16] implements a simple authentication mechanism where a user has a maximum number of `g_ptc` attempts to enter the correct 4-digit `userPIN` matching the secret `cardPIN`. When the authentication succeeds, a global variable `g_authenticated` is set to true. The program is available in eight versions with an increasing number of protections against fault attacks. VerifyPIN_v0 has no protection, while VerifyPIN_v7 is the version with the highest number of countermeasures. It implements hardened Boolean variables, constant iteration loops, loop counter checks, inline function calls, and duplication of critical tests.

Attacker Goal. The attacker aims to *i)* bypass the secure authentication, i.e., φ_{authen} : (`g_authenticated` = *true*), or *ii)* manipulate the maximum number of authentication tries, i.e., φ_{ptc} : (`g_ptc` ≥ 3).

For each analysis, the attacker's goal is evaluated at the end of VerifyPIN function once the program counter reaches the exit point. Faults can be injected during the entire execution of the program. As a result, Table 3 illustrates that φ_{authen} and φ_{ptc} are reachable by an attacker in most of the eight versions of VerifyPIN. For example, injecting a fault when decrementing `g_ptc` fulfills φ_{ptc}. In addition, we observed that φ_{authen} can be reached by setting the `cardPIN` pointer equal to the `userPIN` pointer and comparing the `cardPIN` code to itself.

7.3 Fixing Register File Vulnerability

As demonstrated in Section 7.1, a single fault into the output mux tree of the register file could modify which value is written back to the Secure Ibex. Figure 8 depicts our hardware modifications to protect the register file from faults.

First, the read addresses *raddr_a* and *raddr_b* are converted to one-hot encoded signals, and their integrity is ensured by checker modules. Then, the one-hot encoded read addresses are each fed into a mux directly operating on these signals. Internally, the one-hot mux selects each output bit individually by performing AND- and OR-reductions on the one-hot encoded addresses and the register file values. As a result, a single bit-flip is immediately detected either by the one-hot encoding checkers or the EDC protections.

As reported in Table 2, our verification flow proves the 1-fault security of the fixed register file. Since the fixed Ibex is robust to one single fault injection, no exploitable faults need to be verified in the system verification step, which reduces the overall security verification time. We reported this finding to the OpenTitan project and provided a security enhancement that got integrated[6] into the project.

8 Related Work

This section discusses our k-fault-resistant partitioning notion to an existing security definition and compares our methodology to software/hardware fault verification approaches.

[5] Actually, we observe values on the data memory interface as OpenTitan implements memory scrambling.
[6] https://github.com/lowRISC/ibex/pull/2117

Non-Accumulative Gadgets. Dhooghe and Nikova introduced the composable notion of *Non-Accumulation (NA)* to build secure systems using gadgets [DN20]. In detail, a *k-NA* gadget attacked by $k' \leq k$ faults either aborts or gives an output with at most k' faults. However, this notion does not consider sequential logic, where faults can stay hidden in registers and modify outputs over multiple clock cycles. Besides, it cannot model delayed detection, which is crucial for verifying current CPU countermeasures. Furthermore, they provide guidelines for building secure gadgets but no methodology for evaluating off-the-shelf systems. In contrast, our approach solves the problem of fault propagation in sequential logic. We also provide an algorithm for building partitioning that showed to be efficient on an industry-grade CPU.

Hardware Fault Verification. FIVER [RSS+21] transforms the circuit to analyze into a binary decision diagram. It compares the outputs of the golden to the faulty model to reveal the consequences of the fault on cryptographic circuits. Similarly, SYNFI [NOV+22] is a pre-silicon fault analysis framework that allows hardware designers to evaluate the robustness of a circuit and its countermeasures against faults. These tools perform bounded equivalence checking, meaning the circuit is unrolled over several clock cycles to be analyzed. This technique is not suitable for processor verification, as the consequences of faults may occur after an unknown amount of time. In contrast, our methodology provides unbounded security guarantees, making it possible to introduce software co-verification as a second step. As a result, we can thoroughly verify large processors such as the Secure Ibex.

Furthermore, our approach can prove the security of cryptographic circuits against multiple fault injections, as demonstrated in Section 6. Performance-wise, we outperform FIVER by orders of magnitude when proving the AES security against two faults, and our inductive property is valid for multi-round encryption, while FIVER's results hold for a single round. Additionally, we analyzed the 3-fault security of AES, which was previously impossible with state-of-the-art tools. As SYNFI only analyzed parts of the AES block (e.g., the FSM) and parts of the CPU, a performance comparison is not possible.

Software Fault Verification. ARMORY [HSP21] is a framework capable of automatically injecting faults during program execution using an ARMv7-M emulator to analyze their effects. Like ARMORY, ARCHIE [HGA+21] injects faults into software when executed on an emulator. However, ARCHIE performs the fault analysis architecture independently, i.e., ARM, RISC-V, x86, and other architectures are supported. FiSim [Ris] injects faults into instructions to analyze whether a specific attack goal, e.g., skipping a password check, can be achieved. However, as these frameworks perform their analysis using architectural models instead of actual implementations, they are unable to spot vulnerabilities induced by subtle effects of the microarchitecture [TAC+22].

Hardware/Software Fault Verification. A first work [TAC+23] has jointly modeled hardware and software in a framework based on Yosys. However, modeling all the system's components in the same formal model leads to scalability issues and state-of-the-art hardware model checkers [NPWB18, MIL+21] cannot cope with such complex models.

Complexity-wise, the state space to analyze depends on the design size in terms of gates $|G|$, the program length n, and the size of fault model $|\mathcal{F}|$ to the power of the fault order k. Our approach divides the verification into two steps to reduce this complexity. First, by analyzing the hardware design independently of the program, we no longer depend on the program length n and only need to unroll the circuit d times where the countermeasure delay $d \ll n$ is much smaller than n. Second, the co-verification introduces the software program in the model while the design complexity $|G|$ and the remaining exploitable faults $|\mathcal{F}'|$ can be tremendously reduced, provided countermeasures are proven robust.

Performance-wise, [TAC+23] evaluated the Secure Ibex with a restricted fault model $\mathcal{F}$ targeting the shadow core's registers only, i.e., 2 500 fault locations, and verifying a 46-instruction program in 5 min. However, the authors reported they encountered scalability issues and could not process more than a hundred instructions nor evaluate a larger design with more faults. In contrast, we considered any possible bit-flips, i.e., 125 000 faults, and reduced them to a smaller set of exploitable faults $\mathcal{F}'$ with 172 undetected faults in the register file, proving that only targeting the shadow core is futile. The co-verification step is then able to verify the robustness of 2 526 instructions of the secure boot in 2 h 30.

Security-wise, our approach provides much stronger guarantees. Our approach checks the hardware only once. If no vulnerabilities are found, all programs are secure. Otherwise, we use hardware verification results for any program co-verification. In the specific case of the Ibex processor, we formally prove that only faults in the register file are possible and analyze their consequences on various programs. On the opposite, [TAC+23]'s results only hold for VerifyPIN, and the entire software/hardware verification must be rerun for each program. Finally, our methodology is agnostic from the third-party co-verification tool.

9 Conclusion and Future Work

This paper introduced a novel notion of *k-fault-resistant partitioning* to enable the assessment of redundancy-based hardware countermeasures to fault injections. We provide unbounded fault verification proofs for the k-fault security of a circuit using *k-fault-resistant partitioning*. The hardware security vulnerabilities identified by our methodology are then exploited in a second verification phase taking into account the software. This enables us to verify previously intractable problems, such as analyzing the robustness of OpenTitan running a secure boot process. To demonstrate the capabilities of *k-fault-resistant partitioning*, we provided a complete formal analysis of a development version of the Secure Ibex processor used in the OpenTitan chip. Hereby, we identified a security vulnerability in the register file, showed that it could be exploited in third-party software, and provided a fix to mitigate the security issue that got integrated into the OpenTitan project.

Future work will extend the *k-fault-resistant partitioning* notion to support non-separable CED-based hardware countermeasures or mixed hardware/software protections, like control-flow integrity, where software- or hardware-only verification techniques cannot be used. The methodology and the tool could also be extended to support permanent faults.

Acknowledgement

This project has received funding from the Austrian Research Promotion Agency (FFG) via the AWARE project (grant number 891092). This work was also funded in part by the French National Research Agency (ANR) under the ARSENE (ANR-22-PECY-0004) and LOTR (ANR-23-CE25-0016) projects, in the framework of the Investissements d'Avenir program (ANR-10-AIRT-05, IRTNanoElec), Carnot Flexsecurity, and by Key Digital Technologies Joint Undertaking (KDT JU) under the TRISTAN project (101095947). Finally, lowRISC C.I.C. contributed to the security fixes introduced in this paper and their work has been funded by the OpenTitan project.

References

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

[BBK+03] Guido Bertoni, Luca Breveglieri, Israel Koren, Paolo Maistri, and Vincenzo Piuri. Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard. *IEEE Trans. Computers*, 52:492–505, 2003.

[BBK+10] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security, WESS 2010, Scottsdale, AZ, USA, October 24, 2010*, page 7, 2010.

[BCN+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proc. IEEE*, 94:370–382, 2006.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology – EUROCRYPT'97*, pages 37–51, 1997.

[BEMP20] Etienne Boespflug, Cristian Ene, Laurent Mounier, and Marie-Laure Potet. Countermeasures Optimization in Multiple Fault-Injection Context. In *Fault Diagnosis and Tolerance in Cryptography – FDTC'20*, pages 26–34, 2020.

[BFFH20] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, pages 51–53, 2020.

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Advances in Cryptology – CRYPTO'97*, volume 1294 of *LNCS*, pages 513–525, 1997.

[CCH23] Thomas Chamelot, Damien Couroussé, and Karine Heydemann. MAFIA: Protecting the Microarchitecture of Embedded Systems Against Fault Injection Attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[DBP23] Soline Ducousso, Sébastien Bardin, and Marie-Laure Potet. Adversarial Reachability for Program-level Security Analysis. In *European Symposium on Programming – ESOP'23*, volume 13990 of *LNCS*, pages 59–89, 2023.

[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.*, 2018:547–572, 2018.

[dHOGT21] Jan Van den Herrewegen, David F. Oswald, Flavio D. Garcia, and Qais Temeiza. Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021:56–81, 2021.

[DN20] Siemen Dhooghe and Svetla Nikova. My Gadget Just Cares for Me - How NINA Can Prove Security Against Combined Attacks. In *Topics in Cryptology – CT-RSA'20*, volume 12006 of *LNCS*, pages 35–55, 2020.

[DPP+16] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. FISSC: A Fault Injection and Simulation Secure Collection. In *Computer Safety, Reliability, and Security – SAFECOMP'16*, volume 9922 of *LNCS*, pages 3–11, 2016.

[DRPR19] Jean-Max Dutertre, Timothée Riom, Olivier Potin, and Jean-Baptiste Rigaud. Experimental Analysis of the Laser-Induced Instruction Skip Fault Model. In *Nordic Conference on Secure IT Systems – NordSec'19*, volume 11875 of *LNCS*, pages 221–237, 2019.

[HGA+21] Florian Hauschild, Kathrin Garb, Lukas Auer, Bodo Selmke, and Johannes Obermaier. ARCHIE: A QEMU-Based Framework for Architecture-Independent Evaluation of Faults. In *Fault Diagnosis and Tolerance in Cryptography – FDTC'21*, pages 20–30, 2021.

[HSP21] Max Hoffmann, Falk Schellenberg, and Christof Paar. ARMORY: Fully Automated and Exhaustive Fault Simulation on ARM-M Binaries. *IEEE Trans. Inf. Forensics Secur.*, 16:1058–1073, 2021.

[IBE] Ibex RISC-V Core github repository. https://github.com/lowRISC/ibex. Accessed: December 22, 2023.

[JRR+18] Scott Johnson, Dominic Rizzo, Parthasarathy Ranganathan, Jon McCune, and Richard Ho. Titan: enabling a transparent silicon root of trust for cloud. In *Hot Chips: A Symposium on High Performance Chips*, volume 194, 2018.

[kok] kokke. Tiny AES. https://github.com/kokke/tiny-AES-c. Accessed: June 19, 2023.

[KQ08] Chong Hee Kim and Jean-Jacques Quisquater. New Differential Fault Analysis on AES Key Schedule: Two Faults Are Enough. In *Smart Card Research and Advanced Applications – CARDIS'08*, volume 5189 of *LNCS*, pages 48–60, 2008.

[KR23] Keerthi K. and Chester Rebeiro. FaultMeter: Quantitative Fault Attack Assessment of Block Cipher Software. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023:212–240, 2023.

[KSV13] Dusko Karaklajic, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware Designer's Guide to Fault Attacks. *IEEE Trans. Very Large Scale Integr. Syst.*, 21:2295–2306, 2013.

[KWMK02] Ramesh Karri, Kaijie Wu, Piyush Mishra, and Yongkook Kim. Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 21:1509–1517, 2002.

[LBD+18] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. On the Importance of Analysing Microarchitecture for Accurate Software Fault Models. In *Digital System Design – DSD'18*, pages 561–564, 2018.

[LDPB21] Johan Laurent, Christophe Deleuze, Florian Pebay-Peyroula, and Vincent Beroulle. Bridging the Gap between RTL and Software Fault Injection. *ACM J. Emerg. Technol. Comput. Syst.*, 17:38:1–38:24, 2021.

[MIL+21] Makai Mann, Ahmed Irfan, Florian Lonsing, Yahan Yang, Hongce Zhang, Kristopher Brown, Aarti Gupta, and Clark W. Barrett. Pono: A Flexible and Extensible SMT-Based Model Checker. In *Computer Aided Verification – CAV'21*, volume 12760 of *LNCS*, pages 461–474, 2021.

[MM00] Subhasish Mitra and Edward J. McCluskey. Which concurrent error detection scheme to choose ? In *Proceedings IEEE International Test Conference 2000, Atlantic City, NJ, USA, October 2000*, pages 985–994, 2000.

[MP23] Krzysztof Marcinek and Witold A Pleskacz. Variable delayed dual-core lockstep (vdcls) processor for safety and security applications. *Electronics*, 12(2):464, 2023.

[NM23] Pascal Nasahl and Stefan Mangard. SCRAMBLE-CFI: Mitigating Fault-Induced Control-Flow Attacks on OpenTitan. In *ACM Great Lakes Symposium on VLSI – GLSVLSI'23*, pages 45–50, 2023.

[NOV+22] Pascal Nasahl, Miguel Osorio, Pirmin Vogel, Michael Schaffner, Timothy Trippel, Dominic Rizzo, and Stefan Mangard. SYNFI: Pre-Silicon Fault Analysis of an Open-Source Secure Element. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022:56–87, 2022.

[NPWB18] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , BtorMC and Boolector 3.0. In *Computer Aided Verification – CAV'18*, volume 10981 of *LNCS*, pages 587–595, 2018.

[NSL+23] Pascal Nasahl, Salmin Sultana, Hans Liljestrand, Karanvir Grewal, Michael LeMay, David M. Durham, David Schrammel, and Stefan Mangard. EC-CFI: Control-Flow Integrity via Code Encryption Counteracting Fault Attacks. In *Hardware Oriented Security and Trust – HOST'23*, pages 24–35, 2023.

[NT19] Pascal Nasahl and Niek Timmers. Attacking AUTOSAR using Software and Hardware Attacks. In *Embedded Security in Cars USA (escar)*, 2019.

[Opea] OpenTitan: Lightweight Threat Model. https://opentitan.org/book/doc/security/threat_model/index.html. Accessed: December 22, 2023.

[Opeb] OpenTitan: Open source silicon root of trust. https://github.com/lowRISC/opentitan. Accessed: December 22, 2023.

[PHB+19] Julien Proy, Karine Heydemann, Alexandre Berzati, Fabien Majéric, and Albert Cohen. A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures: A Secure Software Perspective. In *Availability, Reliability and Security – ARES'19*, pages 7:1–7:10, 2019.

[PMPD14] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections. In *International Conference on Software Testing, Verification and Validation – ICST'14*, pages 213–222, 2014.

[Ris] Riscure. Fisim. https://github.com/Riscure/FiSim. Accessed: May 30, 2023.

[RSS+21] Jan Richter-Brockmann, Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. FIVER - Robust Verification of Countermeasures against Fault Injections. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021:447–473, 2021.

[Sny] Wilson Snyder. Verilator. https://veripool.org/verilator/. Accessed: June 6, 2023.

[SS92] Daniel P. Siewiorek and Robert S. Swarz. *Reliable computer systems - design and evaluation (2. ed.)*. 1992.

[TAC+22] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. Exploration of Fault Effects on Formal RISC-V Microarchitecture Models. In *Fault Diagnosis and Tolerance in Cryptography – FDTC'22*, pages 73–83, 2022.

[TAC+23] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. μARCHIFI: Formal Modeling and Verification Strategies for Microarchitectural Fault Injections. In *Formal Methods in Computer-Aided Design – FMCAD'23*, pages 101–109, 2023.

[TFY07] Junko Takahashi, Toshinori Fukunaga, and Kimihiro Yamakoshi. DFA Mechanism on the AES Key Schedule. In *Fault Diagnosis and Tolerance in Cryptography – FDTC'07*, pages 62–74, 2007.

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In *Information Security Theory and Practice – WISTP'11*, volume 6633 of *LNCS*, pages 224–233, 2011.

[VM02] Thomas Verdel and Yiorgos Makris. Duplication-Based Concurrent Error Detection in Asynchronous Circuits: Shortcomings and Remedies. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems – DFT'02*, pages 345–353, 2002.

[VTM+17] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adèle Morisset, and Sébastien Ermeneux. Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot. In *Fault Diagnosis and Tolerance in Cryptography – FDTC'17*, pages 41–48, 2017.

[Wol] Claire Wolf. Yosys open synthesis suite. https://yosyshq.net/yosys. Accessed: July 24, 2023.

[YGS+16] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. Software Fault Resistance is Futile: Effective Single-Glitch Attacks. In *Fault Diagnosis and Tolerance in Cryptography – FDTC'16*, pages 47–58, 2016.

[YSW18] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. *J. Hardw. Syst. Secur.*, 2:111–130, 2018.

A Proof of Theorem 1

Proof. To prove that a circuit $\mathcal{C}$ with partitioning $\mathcal{P}$ fulfilling Definition 10 also fulfills Definition 9, we first prove that it satisfies a stronger inductive property for all $n \in \mathbb{N}^*$:

$$\begin{aligned} &\forall \left(\sigma_i\right)_{i=1}^{n+d},\ \forall \mathbf{F} \subseteq \mathcal{F} \times [1, n+d],\ |\mathbf{F}| \leq k : \left(\bigwedge_{i=1}^{n+d} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) = 0 \right) \Longrightarrow \\ &\left(\Delta_{\mathcal{P}}\left(\sigma_{n+1}, \sigma_{n+1}^{\mathbf{F}_{[1,n]}}\right) \leq \left|\mathbf{F}_{[1,n]}\right|\right) \wedge \left(\bigwedge_{i=1}^{n} O'\left(\sigma_i\right) = O'\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) \right). \end{aligned} \tag{3}$$

Trivially, (3) implies it must also satisfy (1) for all $n \in \mathbb{N}^*$. As mentioned, the proof proceeds inductively over n, generalizing from an arbitrary execution $(\sigma_i)_{i=1}^{n+d}$.

*(**Basis.**)* For the base case, we must demonstrate (3) for $n = 1$, i.e.,

$$\begin{aligned}&\forall (\sigma_i)_{i=1}^{d+1},\ \forall \mathbf{F} \subseteq \mathcal{F} \times [1, d+1],\ |\mathbf{F}| \leq k:\ \left(\bigwedge_{i=1}^{d+1} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) = 0\right) \implies \\ &\left(\Delta_{\mathcal{P}}\left(\sigma_2, \sigma_2^{\mathbf{F}_{[1,1]}}\right) \leq \left|\mathbf{F}_{[1,1]}\right|\right) \wedge \left(O'(\sigma_1) = O'\left(\sigma_1^{\mathbf{F}_{[1,1]}}\right)\right).\end{aligned} \tag{4}$$

This follows directly from (2). Let $\mathbf{F} \subseteq \mathcal{F} \times [1, d+1]$ be an attack with $|\mathbf{F}| \leq k$, $(\sigma_i)_{i=1}^{d+1}$ be an execution, and lastly, $(\hat{\sigma}_i)_{i=1}^{d+1}$ be a second execution with $\hat{\sigma}_i = \sigma_i^{\mathbf{F}_{[1,i-1]}}$. Applying (2) with $j = 1$ and $k' = 0$ yields

$$\begin{aligned}&\left(\bigwedge_{i=1}^{d+1} I(\sigma_i) = I\left(\sigma_i^{\mathbf{F}_{[1,i-1]}}\right)\right) \wedge \left(\Delta_{\mathcal{P}}\left(\sigma_1, \sigma_1^{\mathbf{F}_{[1,0]}}\right) \leq 0\right) \wedge \left(\bigwedge_{i=1}^{d+1} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) = 0\right) \\ &\implies \left(\Delta_{\mathcal{P}}\left(\sigma_2, \sigma_2^{\mathbf{F}_{[1,1]}}\right) \leq \left|\mathbf{F}_{[1,1]}\right|\right) \wedge \left(O'(\sigma_1) = O'\left(\sigma_1^{\mathbf{F}_{[1,1]}}\right)\right).\end{aligned}$$

As faults in prior states cannot corrupt the input in the current state, we can conclude that $\left(\bigwedge_{i=1}^{d+1} I(\sigma_i) = I\left(\sigma_i^{\mathbf{F}_{[1,i-1]}}\right)\right) = \top$. Furthermore, since $\sigma_1^{\mathbf{F}_{[1,0]}} = \sigma_1^{\mathbf{F}_\varnothing} = \sigma_1$, we get $\Delta_{\mathcal{P}}\left(\sigma_1, \sigma_1^{\mathbf{F}_{[1,0]}}\right) = 0$, simplifying the left-hand side of the implication to just the last term. Generalizing the result, i.e., introducing quantification over free variables $(\sigma_i)_{i=1}^{d+1}$ and $\mathbf{F} \subseteq \mathcal{F} \times [1, d+1]$ with $|\mathbf{F}| \leq k$, produces (4) and concludes the induction basis.

*(**Step.**)* For the induction step, we have to show that necessarily

$$\begin{aligned}&\forall (\sigma_i)_{i=1}^{n+d+1},\ \forall \mathbf{F} \subseteq \mathcal{F} \times [1, n+d+1],\ |\mathbf{F}| \leq k:\ \left(\bigwedge_{i=1}^{n+d+1} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) = 0\right) \implies \\ &\left(\Delta_{\mathcal{P}}\left(\sigma_{n+2}, \sigma_{n+2}^{\mathbf{F}_{[1,n+1]}}\right) \leq \left|\mathbf{F}_{[1,n+1]}\right|\right) \wedge \left(\bigwedge_{i=1}^{n+1} O'(\sigma_i) = O'\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)\right)\end{aligned} \tag{5}$$

under the assumption that (3) holds. First, let $(\sigma_i)_{i=1}^{n+d+1}$ be an arbitrary execution and $\mathbf{F} \subseteq \mathcal{F} \times [1, n+d+1]$ be an arbitrary attack with $|\mathbf{F}| \leq k$. Consider the expression $\left(\bigwedge_{i=1}^{n+d+1} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) = 0\right)$ and assume it is true ($\top$). Consequently, the weaker expression from 1 up to $n+d$ is also true, i.e., $\left(\bigwedge_{i=1}^{n+d} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) = 0\right) = \top$. This, together with an application of (3) to the execution $(\sigma_i)_{i=1}^{n+d}$, means that $\left(\Delta_{\mathcal{P}}\left(\sigma_{n+1}, \sigma_{n+1}^{\mathbf{F}_{[1,n]}}\right) \leq \left|\mathbf{F}_{[1,n]}\right|\right) = \top$ and $\left(\bigwedge_{i=1}^{n} O'(\sigma_i) = O'\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)\right) = \top$. Next, instantiate (2) for the executions $(\sigma_i)_{i=n+1}^{n+l+1}$ and $(\hat{\sigma}_i)_{i=n+1}^{n+l+1}$, with $\hat{\sigma}_i = \sigma_i^{\mathbf{F}_{[1,i-1]}}$, the number $k' = \left|\mathbf{F}_{[1,n]}\right|$ and fault attack $\mathbf{F}_{[n+1,n+d+1]}$, which works because $\left|\mathbf{F}_{[n+1,n+d+1]}\right| + k' = \left|\mathbf{F}_{[n+1,n+d+1]}\right| + \left|\mathbf{F}_{[1,n]}\right| = |\mathbf{F}| \leq k$, to get

$$\begin{aligned}&\left(\bigwedge_{i=n+1}^{n+d+1} I(\sigma_i) = I\left(\sigma_i^{\mathbf{F}_{[1,i-1]}}\right)\right) \wedge \left(\Delta_{\mathcal{P}}\left(\sigma_{n+1}, \sigma_{n+1}^{\mathbf{F}_{[1,n]}}\right) \leq \left|\mathbf{F}_{[1,n]}\right|\right) \wedge \left(\bigwedge_{i=n+1}^{n+d+1} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) = 0\right) \\ &\implies \left(\Delta_{\mathcal{P}}\left(\sigma_{n+2}, \sigma_{n+2}^{\mathbf{F}_{[1,n+1]}}\right) \leq \left|\mathbf{F}_{[1,n]}\right| + \left|\mathbf{F}_{\{n+1\}}\right|\right) \wedge \left(O'(\sigma_{n+1}) = O'\left(\sigma_{n+1}^{\mathbf{F}_{[1,n+1]}}\right)\right).\end{aligned}$$

Similarly to the basis step, past faults cannot lead to different inputs in the current state, and therefore $\left(\bigwedge_{i=n+1}^{n+d+1} I(\sigma_i) = I\left(\sigma_i^{\mathbf{F}_{[1,i-1]}}\right)\right) = \top$. Moreover, the weaker term from $n+1$ to $n+d+1$ of our assumption must also be true, i.e., $\left(\bigwedge_{i=n+1}^{n+d+1} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) = 0\right) = \top$.

The left-hand side of the implication is $\top$, yieling $\left(\Delta_{\mathcal{P}}\left(\sigma_{n+2}, \sigma_{n+2}^{\mathbf{F}_{[1,n+1]}}\right) \leq \left|\mathbf{F}_{[1,n+1]}\right|\right) = \top$ and $\left(O'\left(\sigma_{n+1}\right) = O'\left(\sigma_{n+1}^{\mathbf{F}_{[1,n+1]}}\right)\right) = \top$. Joining the previous facts about the output into $\left(\bigwedge_{i=1}^{n+1} O'\left(\sigma_i\right) = O'\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)\right)$, we have proven the implication in (5). After generalization, we get (5) itself. □

B Vulnerabilities in Impeccable Circuits Implementations

The Skinny-64 and AES-128 implementations from Impeccable Circuit [AMR+20] protected against 3 faults revealed to be vulnerable. For each cipher, we detail reproducible attack scenarios providing the Plaintext, the Key, and the faults to be injected according to the circuit nomenclature to produce incorrect ciphertexts without triggering an alert.

Skinny-64 red-4. Plaintext: `0x06034f957724d19d`; Key: `0xf5269826fc681238`; Expected ciphertext: `0xbb39dfb2429b8ac7`; 1^{st} fault: (bit-flip, round 0, `Red_StateReg.s_current_state[44]`); 2^{nd} fault: (bit-flip, round 0, `SubCellOutput[46]`); 3^{rd} fault: (bit-flip, round 0, `Check1.in1[244]`); Faulty ciphertext: `0x0897810d2aa02f8e`.

AES-128 red-5. Plaintext: `0xd2228cc9f8b8f239b0162a9ad3632127`; Key: `0x7f287089-fbbebdb8f364377b97f5c9ef`; Expected ciphertext: `0x6666b1677c13464929f286aca090eb74`; 1^{st} fault: (bit-flip, at round 0, `InputMUX.Q[31]`); 2^{nd} fault: (bit-flip, at round 0, `RedFinalRoundControlLogicInst.Red_FinalRoundBit[2]`); 3^{rd} fault: (bit-flip, at round 0, `Check1.result[2]`); Faulty ciphertext: `0xfd711dada3bfa30b6406f71be54e20a1`.