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. To assess the robustness of CPU-based systems against fault injection attacks, it is necessary to analyze the consequences of the fault propagation resulting from the intricate interaction between the software and the processor. However, current formal methodologies that combine both hardware and software aspects experience scalability issues, primarily due to the use of bounded verification techniques. This work formalizes the notion of *k-fault resistant partitioning* as an inductive solution to this fault propagation problem when assessing redundancy-based hardware countermeasures to fault injections. Proven security guarantees can then reduce the remaining hardware attack surface to consider in a combined analysis with the software, enabling a full co-verification methodology. As a result, we formally verify the robustness of the hardware lockstep countermeasure of the OpenTitan secure element 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, verify 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 [BDL, 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. For pre-silicon security evaluation and to improve fault coverage, simulation or formal verification tools are used. 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-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.

To demonstrate the capabilities of such an enhanced fault injection analysis methodology, we analyze the k-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-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 [kok19]. 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 in the Ibex core used in a development version of OpenTitan to the 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-secure. Our fix was integrated[2] into the OpenTitan project.

Outline. This paper is structured as follows. Section 2 introduces the notations and background necessary for this work. Section 3 describes our hardware/software co-verification methodology. Section 4 is dedicated to the k-fault resistant partitioning property at the root of our methodology, its security implication and how to use it in practice. Section 5 presents the implementation of the proposed methodology we use for the experimental evaluations. In Section 6, we evaluate the fault resilience of OpenTitan's secure processor. Section 7 highlights and compares related work to our pre-silicon fault verification approach. Finally, Section 8 concludes this work.

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.

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.

[1] The link will be provided after paper acceptance.
[2] https://github.com/lowRISC/ibex/pull/2117

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 simple circuit partitioning where r_1 and r_2 belongs to partition P_1 and r_3 to P_2.

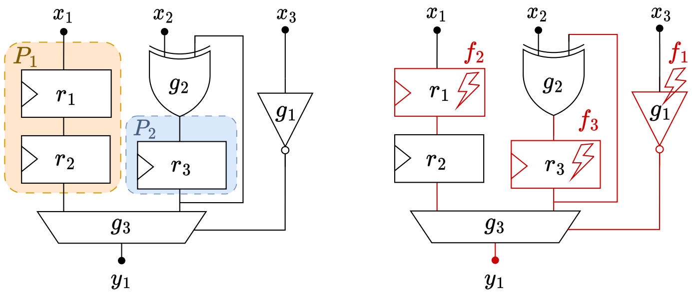
2.2 Fault Injection Attacks

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* $u \in U$.

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 $J \subseteq [1, n]$.



(a) Circuit partitioning. **(b)** Fault propagation.

Figure 1: Simple circuit examples.

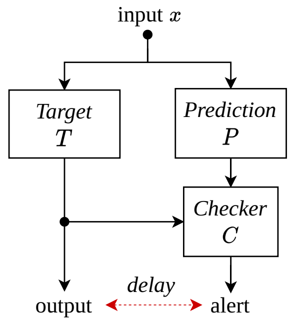


Figure 2: CED scheme.

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 $\varphi\left(\sigma_n^{\mathbf{F}}\right) = 1$.

2.3 Concurrent Error Detection Schemes

Concurrent Error Detection schemes (CEDs) attempt to protect a system against fault attacks using spatial redundancy and checking mechanisms. Figure 2 depicts such a scheme where *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. Alternatively, P can also be implemented with error detection codes [RSBG20]. Some implementations also introduce a *delay* between the operation of the target and the prediction functions to increase the practical difficulty of faulting both [VM02, MP23]. We formalize CED schemes in Definitions 8 and 9.

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 $A(\sigma_i) \neq 0$, i.e., $\exists\, a \in A$ such that $a(\sigma_i) \neq 0$.

Definition 9 (k-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

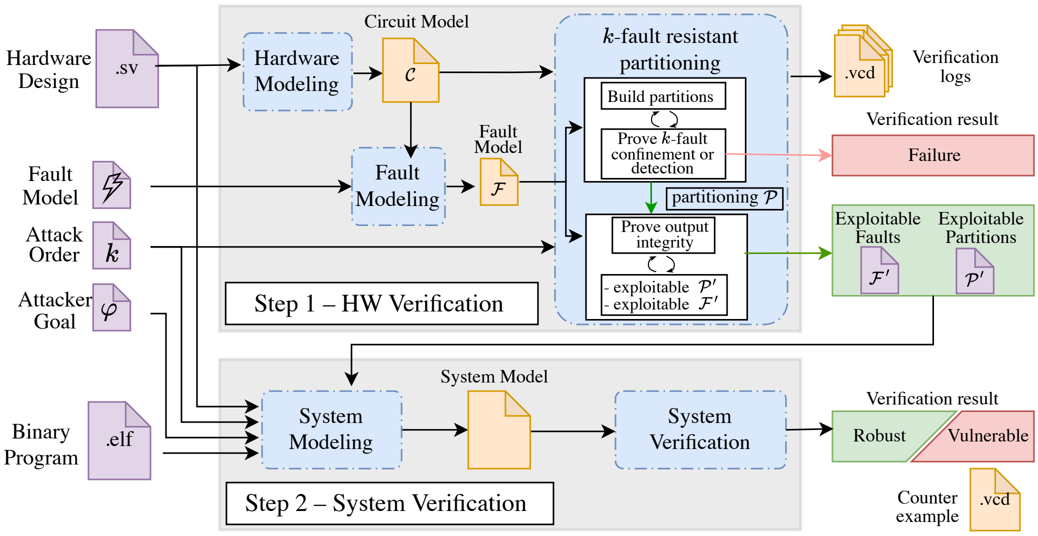


Figure 3: Co-verification methodology for fault injection analysis on combined software/hardware systems.

and k be the attack order. We say that the (d, A)-CED is k-*secure* against the fault model $\mathcal{F}$ if and only if, $\forall n \in \mathbb{N}^*$,

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

Intuitively, Definition 9 says that k-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-security of a (d, A)-CED against the fault model $\mathcal{F}$ often relies on bounded equivalence checking [RSS$^+$21, NOV$^+$22]. As depicted 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-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-security in the general case, and may struggle as the checking complexity increases with n.

3 Co-Verification Methodology

This section introduces our hardware/software co-verification methodology, which is capable of analyzing the system's robustness against fault injections. In our work, we focus on CPU-based systems implementing one or several CED schemes. The proposed methodology, illustrated in Figure 3, is composed of two steps. In the first step, labeled Step 1 and highlighted in the top box of Figure 3, we evaluate the robustness of the implemented

CED schemes against faults at order k. The second step, labeled Step 2 and depicted in the bottom box of Figure 3, is dedicated to the full system verification.

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-outed circuit. 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-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-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.

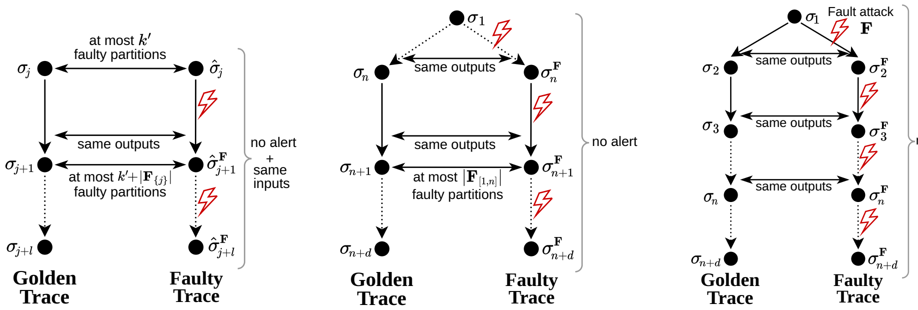
The verification of the k-fault resistant partitioning is performed by first building iteratively a partitioning that ensures fault confinement or detection for the attack order k and the fault model $\mathcal{F}$. When this construction fails, the user can inspect the verification logs to understand 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 $\mathcal{P}'$.

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



(a) k-fault resistant partitioning. **(b)** Inductive property. **(c)** k-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)**.

fault locations derived from $\mathcal{F}'$ and $\mathcal{P}'$ help to select the best-suited level of abstraction 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. A counterexample helps the user to understand where the fault was injected and how it propagates in the system to create the vulnerability.

4 k-Fault Resistant Partitioning

This section first formally defines the notion of *k-fault resistant partitioning* before proving that k-fault resistant partitioning of a circuit $\mathcal{C}$ implementing a (d, A)-CED scheme implies its k-security. 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-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-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-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

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

Similar to k-security, the definition of k-fault resistant partitioning also considers two execution traces $\left(\sigma_i\right)_{i=j}^{j+l}$ and $\left(\hat{\sigma}_i\right)_{i=j}^{j+l}$ 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 $\left(\hat{\sigma}_i\right)_{i=j}^{j+l}$ 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 $\left|\mathbf{F}_{\{j\}}\right|$ 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-secure. We facilitate the proof through a stronger inductive property, as depicted in Figure 4b.

Theorem 1 (k-fault resistant partitioning implies k-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-secure.*

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 $n \in \mathbb{N}^*$:

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

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 $\left(\sigma_i\right)_{i=1}^{n+d}$.

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

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

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

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

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

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

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 $\left(O'(\sigma_{n+1}) = O'\left(\sigma_{n+1}^{\mathbf{F}_{[1,n+1]}}\right)\right) = \top$. Joining the previous facts about the output into $\left(\bigwedge_{i=1}^{n+1} O'(\sigma_i) = O'\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)\right)$, we have proven the implication in (5). After generalization, we get (5) itself. □

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

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 $\mathcal{F} \subseteq G \times U$, and an attack order k.

Output: 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 $\mathcal{P}' \subseteq \mathcal{P}$.

Init:
- Initialize the partitioning $\mathcal{P}$, by default $\mathcal{P} \leftarrow \left\{\{r_1\}, \{r_2\}, \ldots, \{r_{|R|}\}\right\}$
- Create $(\sigma_i)_{i=1}^{d+1}$ and $(\hat{\sigma}_i)_{i=1}^{d+1}$, two arbitrary execution traces of $\mathcal{C}$ with $d+1$ states each
- Create an arbitrary fault attack $\mathbf{F} \subseteq \mathcal{F} \times [1, d+1]$

Global assumption:
- $\psi_{LeftHandSideEq2} := \left(\bigwedge_{i=1}^{d+1} I(\sigma_i) = I(\hat{\sigma}_i)\right) \wedge \left(\bigwedge_{i=1}^{d+1} A\left(\hat{\sigma}_i^{\mathbf{F}}\right) = 0\right) \wedge$ $(\Delta_{\mathcal{P}}(\sigma_1, \hat{\sigma}_1) \leq k') \wedge (|\mathbf{F}| \leq k'') \wedge (k' + k'' \leq k)$, left-hand side of Equation (2)

(1) **Procedure to build and prove k-fault confinement or detection:**
- **Local assumption:**
  - $\psi_{kFaultConfinement} := \left(\Delta_{\mathcal{P}}(\sigma_2, \hat{\sigma}_2) \leq k' + \left|\mathbf{F}_{\{1\}}\right|\right)$, at most $k' + \left|\mathbf{F}_{\{1\}}\right|$ faulty partitions at the next clock cycle
- **while** $(\psi_{LeftHandSideEq2} \wedge \neg\psi_{kFaultConfinement})$ *is SAT* **do**
  - $\mathcal{P}_{init} \leftarrow \{P \in \mathcal{P} \mid P(\sigma_1) \neq P(\hat{\sigma}_1)\}$, corrupted partitions at the initial state
  - $\mathcal{P}_{next} \leftarrow \left\{P \in \mathcal{P} \mid P(\sigma_2) \neq P\left(\hat{\sigma}_2^{\mathbf{F}}\right)\right\}$, corrupted partitions at the next state
  - $\mathcal{P} \leftarrow$ `merge_strategy`$(\mathcal{P}, \mathcal{P}_{init}, \mathcal{P}_{next})$, update the partitioning $\mathcal{P}$
  - **if** $|\mathcal{P}| \leq k$ **then**
    - **return** $\{\}, \{\}, \{\}$ `/* fail to identify a` k`-fault resistant partitioning */`

(2) **Procedure to check outputs' integrity and list exploitable fault locations:**
- **Init:**
  - Initialize empty sets $\mathcal{F}' \leftarrow \{\}$ and $\mathcal{P}' \leftarrow \{\}$ for the exploitable fault locations
- **Local assumption:**
  - $\psi_{NoFaultOn\mathcal{P}'} := \left(\bigwedge_{P \in \mathcal{P}'} P(\sigma_1) = P(\hat{\sigma}_1)\right)$, no fault on exploitable partitions $\mathcal{P}'$
  - $\psi_{NoFaultOn\mathcal{F}'} := (\mathbf{F} \cap \mathcal{F}' \times [1, d+1] = \emptyset)$, no fault on expoitable gate locations $\mathcal{F}'$
  - $\psi_{OutputIntegrity} := \left(O'(\sigma_1) = O'\left(\hat{\sigma}_1^{\mathbf{F}}\right)\right)$, outputs are equal at the initial state
- **while** $(\psi_{LeftHandSideEq2} \wedge \psi_{NoFaultOn\mathcal{P}'} \wedge \psi_{NoFaultOn\mathcal{F}'} \wedge \neg\psi_{OutputIntegrity})$ *is SAT* **do**
  - $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{P \in \mathcal{P} \mid P(\sigma_1) \neq P(\hat{\sigma}_1)\}$, add new exploitable partitions identified
  - $\mathcal{F}' \leftarrow \mathcal{F}' \cup \{(g, u) \in G \times U \mid \exists j, (g, u, j) \in \mathbf{F}\}$ add new exploitable fault injections
- `/*` $\mathcal{P}$ `is` k`-fault resistant assuming` $\psi_{NoFaultOn\mathcal{P}'}$ `and` $\psi_{NoFaultOn\mathcal{F}'}$ `*/`
- **return** $\mathcal{P}, \mathcal{P}', \mathcal{F}'$

resistant partitioning is only a sufficient condition for k-security, it significantly simplifies the endeavor of proving a circuit k-secure because the circuit is only unrolled $max(1, d)$ times, compared to the bounded equivalence checking approach. The following section provides an algorithm to build and prove such a 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 based on SAT solving techniques. It takes as input a circuit model $\mathcal{C}$, a fault model $\mathcal{F}$, and an attack order k. Algorithm 1 comprises procedure (1) to build a partitioning ensuring k-fault confinement or detection and procedure (2) to enumerate fault injections compromising outputs' integrity. Eventually, the algorithm either returns a k-fault resistant partitioning $\mathcal{P}$ with a set of assumptions under which the circuit is k-secure or fails to find such a partitioning and provides counterexamples to understand why.

Initially, we create a circuit partitioning $\mathcal{P}$ with sets of individual registers by default. It can also be set to a previously computed partitioning. In addition, we create two execution traces $(\sigma_i)_{i=1}^{d+1}$ and $(\hat{\sigma}_i)_{i=1}^{d+1}$ of length $d+1$ and consider a fault attack $\mathbf{F}$ included in the range of attacker capabilities $\mathcal{F}$. The fault attack $\mathbf{F}$ and the execution traces are symbolic objects whose behaviors are constrained with the global assumption $\psi_{LeftHandSideEq2}$ (line 5). This assumption corresponds to the implication's premise in Definition 10.

Procedure 1 is an iterative process. Given the current partitioning $\mathcal{P}$, a query is provided to a SAT solver to check whether faults can propagate to more partitions than allowed (assumption $\psi_{kFaultConfinement}$, line 9) after one clock cycle. When $\psi_{LeftHandSideEq2} \wedge \neg\psi_{kFaultConfinement}$ is satisfiable, the procedure collects corrupted partitions $\mathcal{P}_{init}$ at the initial state and corrupted partitions $\mathcal{P}_{next}$ at the next state from the SAT assignment returned by the solver. A `merge_strategy`() function then updates $\mathcal{P}$. This merging aims to gather partitions for which a fault can propagate from one to another without raising the alert. For $k = 1$, the merging groups all the partitions in $\mathcal{P}_{init}$ and $\mathcal{P}_{next}$. When $k > 1$, the merging is more complex as it is often a combination of multiple faults that leads to the corruption of partitions in $\mathcal{P}_{next}$. Consequently, a random strategy is applied while eliminating every impossible merge if the partitions are not connected. As $|\mathcal{P}|$ decreases at each iteration, Procedure 1 converges to a fixed point where partitioning $\mathcal{P}$ fulfills assumption $\psi_{kFaultConfinement}$. However, the procedure fails as soon as the resulting partitioning has less than k partitions, as such a partitioning cannot guarantee the outputs' integrity for the second procedure (cf. Section 3). Procedure 1 may fail for one of the three following reasons: *i)* the circuit $\mathcal{C}$ has some flaws and is not k-secure, *ii)* there is no k-resistant partitioning even if $\mathcal{C}$ is k-secure (cf. Section 4.1), or *iii)* the merging heuristics does not allow to build a k-resistant partitioning. Log files are generated at each iteration to provide the model returned by the SAT solver and the partition merging performed. It may help understand why the procedure fails to find a partitioning $\mathcal{P}$. However, this analysis must be performed manually.

For higher-order fault attacks, a user should start with $k = 1$ to build a 1-fault resistant partitioning $\mathcal{P}_1$ before increasing iteratively k up to the desired security level as a k-fault resistant partitioning must be $(k-1)$-fault resistant. Partitioning $\mathcal{P}_k$ is then used to initialize the next iteration with $k+1$.

Procedure 2 initializes two empty sets $\mathcal{P}'$ and $\mathcal{F}'$ to enumerate exploitable fault locations. $\mathcal{P}'$ contains register locations while $\mathcal{F}'$ contains gate locations. Then, it iteratively verifies if the partitioning $\mathcal{P}$ guarantees outputs' integrity in the presence of k faults while not targeting the exploitable fault locations identified in previous iterations. When $\psi_{LeftHandSideEq2} \wedge \psi_{NoFaultOn\mathcal{F}'} \wedge \psi_{NoFaultOn\mathcal{P}'} \wedge \neg\psi_{OutputIntegrity}$ is satisfiable, the procedure collects the fault locations from the assignment returned by the SAT solver. The sets of exploitable fault locations $\mathcal{P}'$ and $\mathcal{F}'$ are updated accordingly. At the end, procedure 2 returns the partitioning $\mathcal{P}$ proven k-secure considering no fault on the $\mathcal{P}'$ and $\mathcal{F}'$ that are also returned.

For attack order $k = 1$, some combinational faults can be optimized away from the considered faults. Since Equation (2) assumes there is no output corruption at the initial state, we can remove fault locations not combinatorially connected to the circuit's primary outputs as they cannot have immediate consequences on them.

5 Implementation

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

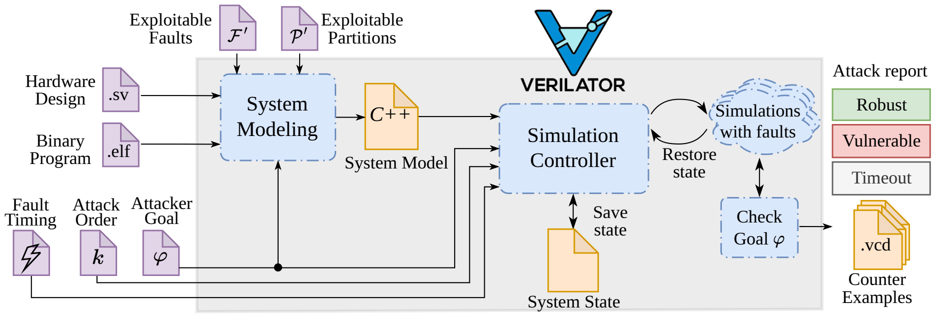


Figure 5: Software/Hardware co-verification flow (Step 2) using Verilator.

5.1 Step 1 — Hardware Verification Flow

When provided at the RTL level, the hardware design is first converted to a bit-level netlist using the synthesis tool Yosys [Wol23] to match the circuit model given in Definition 1. In addition, we also implemented a feature to extract a (d, A)-CED according to its input, output, and alert interface. This feature helps to analyze the security of a complete circuit implementing multiple CED schemes by dividing it into multiple subcircuits.

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 $(\sigma_i)_{i=1}^{d+1}$ and $(\hat{\sigma}_i)_{i=1}^{d+1}$ are modeled unrolling the circuit d times. Fault injections are applied to execution traces using new Boolean variables to 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.

[3] https://github.com/lowRISC/ibex/pull/2117

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.

6 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 whether the identified vulnerabilities can be exploited in different security-critical programs. Third, we provide a hardware fix for the vulnerability discovered and re-evaluate the security using our verification approach. Although this paper focuses on verifying the Secure Ibex processor, we demonstrate that our methodology can also be applied to other hardware, such as cryptographic circuits, in Appendix A.

OpenTitan threat model. The OpenTitan project [JRR+18] provides an open-source[4] secure element design. Internally, OpenTitan consists of the 32-bit RISC-V Ibex processor, a rich set of security features and peripherals including an AES accelerator and a big number accelerator, but also hardened software such as a reference firmware and a secure boot. Globally, these countermeasures aim to protect the chip's confidentiality, integrity, and authenticity[5]. A malicious attacker uses fault injection to violate the security of the chip (*attacker goal*): we consider an attacker having physical access to the platform capable of interfering with its operation by performing fault injection attacks. For our analysis, we consider a single transient bit-flip everywhere in the microarchitecture (*fault model*).

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 to protect code and data against fault attacks (Figure 6). The *dual-core lockstep* (DCLS) mechanism instantiates the Ibex core twice and compares the outputs between the main core and the shadow core. An alert signal is triggered on a mismatch caused by a fault attack, and the system enters a well-defined error state. To increase the protection against fault injections, the shadow core inputs are delayed for d cycles, d being fixed at synthesis time. In our evaluation, we use the default value, $d = 2$. Both core instances share the register file and use Error Detection Codes (EDC) to detect bit-flips. Additionally, the shared register file uses a write-enable glitch detection mechanism, and dummy instructions are randomly inserted to increase the difficulty of the fault injection timing.

6.1 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 1 summarizes

[4] https://github.com/lowRISC/opentitan
[5] https://opentitan.org/book/doc/security/threat_model/index.html

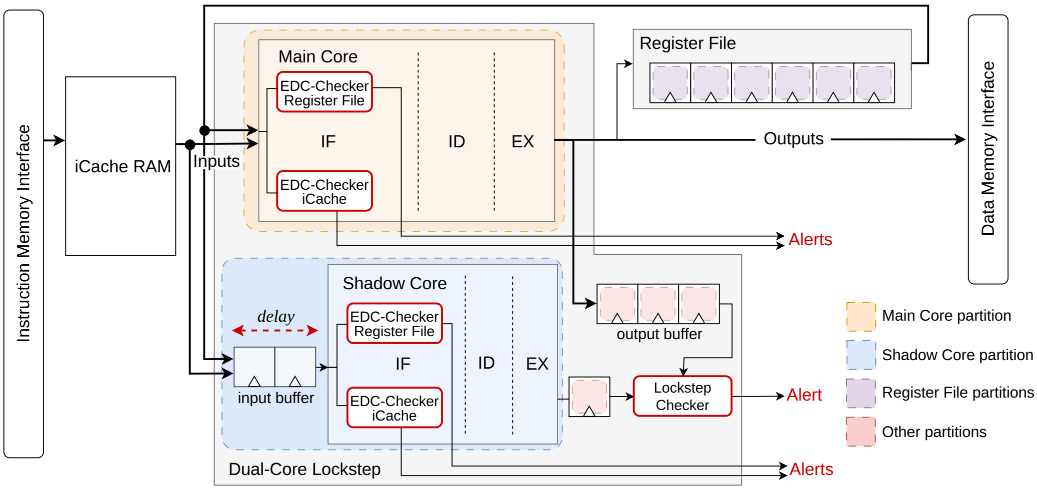


Figure 6: Secure Ibex countermeasures.

the area in gate equivalent (GE) for each circuit, provides the number of possible fault locations, and reports verification results and performance. 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 (cf. Definition 1). We focus on $k = 1$ since the countermeasures of Secure Ibex aim to mitigate a single fault.

Register file analysis. The register file consists of thirty-two 32-bit registers, each protected by a 7-bit EDC. Countermeasures inside the register file 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 (1) has proven that a fault injected in the circuit cannot propagate to multiple registers without being detected by the protections. Table 1 reports that each register is an independent partition except for the 39 bits of the dummy instruction register that belong to the same partition as the register file countermeasures do not protect them.

However, procedure (2) 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

Table 1: Circuit characteristics and performance for k-fault resistant partitioning analysis (with k=1) applied to different Secure Ibex modules. Verifications have been executed on an Intel(R) Xeon(R) Gold 6154 CPU platform.

Circuit Characteristics			Faults		Partitioning Performance			Results		
Name	Size (GE)	Registers (#)	Locations (#)	Order k	Procedure (1)		Procedure (2) (s)	Partitions (#)	Exploitable Faults	
					Iterations (#)	Time (s)			$\mathcal{P}'$ (#)	$\mathcal{F}'$ (#)
Register File	12 075	1 326	8 392	1	53	306.46	37.6	1 288	0	172
DCLS	117 998	5 918	116 561	1	508	72 743 (20h12)	18 619 (5h10)	1 108	0	0
Secure Ibex (no iCache)	130 194	7 248	125 080	1	1	38 684 (10h45)	110 931 (30h50)	2 400	0	0 (+172)

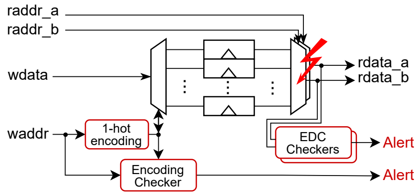


Figure 7: Register file protections and the identified vulnerability.

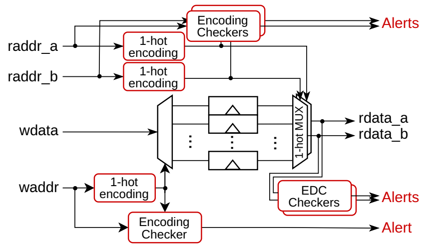


Figure 8: Register file with fixed vulnerability.

we designed, which got integrated into the upstream OpenTitan repository, in Section 6.3.

Dual-Core Lockstep (DCLS) analysis. At first, procedure (1) 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 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 failing to prove system 1-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[6] into the project.

Assuming `enable_cmp_q` $= 1$, our analysis builds 1,108 partitions. Two of them are the main core and the shadow core, while the others are registers that faults cannot corrupt without raising an alert. Figure 6 denotes them as *main core*, *shadow core*, and *other partitions*. Building $\mathcal{P}$ takes 508 iterations in 11 hours, and, then, proving the faults confinement in $\mathcal{P}$ takes 9h20 (Table 1). Finally, procedure (2) proves that the DCLS can detect any single bit-flip in one of the two cores and in its internal comparison logic.

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, so that procedure (1) only needs one iteration to prove the fault confinement (Table 1). As a result, our methodology proves the 1-security of the full Ibex processor against a single fault injection.

6.2 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. Our evaluation results are reported in Table 2.

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

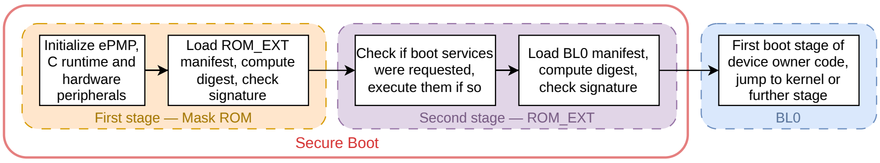


Figure 9: OpenTitan secure boot flow.

6.2.1 Secure Boot

The secure boot process guarantees the *integrity and authenticity of the code running on the device* after a system reset, illustrated Figure 9. The first stage configures the peripherals, sets up the software environment, and also verifies the integrity of the second boot stage, `ROM_EXT`, stored in Flash memory before booting on it. The second stage of the secure boot code provides boot services and also 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 (ROM) 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 whole `rom_verify` function, assuming that the signature is already computed (OTBN module). 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.

6.2.2 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 [kok19] 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 aims to corrupt one byte in the first column of the 9^{th} round key (φ_{key_sched}) of the *key schedule* function [TFY07, KQ08]. The second attack targets the *AES algorithm* itself 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[7] and compare them against the precomputed reference values to determine if the fault induced a single-byte corruption. Table 2 summarizes evaluation results for each

[7] Actually, we observe values on the data memory interface as OpenTitan implements memory scrambling.

Table 2: Co-verification results on software use cases. Verifications have been executed on an Intel(R) Xeon(R) Gold 6154 CPU platform.

Program Characteristics			Fault Characteritics			Analysis Results			Performance	
Name	**Function/ Version**	**Length (# instructions)**	**Attacker Goal φ**	**Timing (clock cycles)**	**Locations (#)**	**Success**	**Fail**	**Timeout**	**Parallelized Runs**	**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

attack. Our analysis reported 532 successful fault injections over the 5760 possibilities to fulfill φ_{key_sched}. Similarly, 4084 successful fault injections were identified over the 38912 configurations tested to reach φ_{aes}. Inspecting the analysis reports shows that successful fault injections are mainly applied to memory load and store operations.

6.2.3 Analysis of 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 higher number of countermeasures. It implements hardened booleans, constant iteration, loop counter check, inline function call, and duplication of critical tests.

Attacker Goal. The attacker aims to *i)* bypass the secure authentication, i.e., φ_{authen} : ($\texttt{g\_authenticated} = \mathit{true}$), or *ii)* manipulate the maximum number of authentication tries, i.e., φ_{ptc} : ($\texttt{g\_ptc} \geq 3$).

For each evaluation, the attacker's goal is resolved 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 2 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.

6.3 Fixing Register File Vulnerability

As demonstrated in Section 6.1, 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. Furthermore, checkers ensure the integrity of these encoded signals. Finally, the one-hot encoded read addresses are each fed into a mux directly operating on these one-hot encoded signals. Internally, the one-hot mux selects each output bit individually by performing an AND and an OR reduction on the one-hot encoded address and the register file.

A fault-induced bit-flip into the read addresses *raddr_a* and *raddr_b* is detected, as the DCLS mechanism constantly checks these signals. The one-hot encoding checkers immediately raise an alert if the signals are not forming valid one-hot codewords or there is a mismatch between the encoded and plain read addresses. Although a fault inside the mux enables the adversary to select a bit from the wrong register file entry, this bit error on *rdata_a* or *rdata_b* is then detected by the EDC checkers.

We formally verify the 1-security of the fixed register file by re-evaluating it using our hardware verification flow. This implies that any single bit-flip induced into the analyzed hardware cannot be exploited in any software. Our patch was integrated[8] into the OpenTitan project.

7 Related Work

In this section, we compare our work to software/hardware fault verification approaches.

Hardware Fault Verification. FIVER [RSS+21] transforms the circuit to analyze into a Binary Decision Diagram (BDD) and compares the outputs of the golden model to the faulty one to reveal the effects of faults on cryptographic circuits. Similarly, SYNFI [NOV+22] is a pre-silicon fault analysis framework allowing hardware designers to evaluate the resilience of a circuit and its countermeasures against faults. The circuit needs to be manually unrolled for analysis over multiple cycles.

Summarized, related work [RSS+21, NOV+22] either proposes techniques to evaluate cryptographic circuits or uses bounded verification methods that are not suitable to verify processors. In contrast, our methodology can fully verify larger CPUs, i.e., Secure Ibex, and provide unbounded guarantees. Moreover, as we demonstrate in Appendix A, our approach also can be utilized to analyze cryptographic circuits against multiple fault injections achieving the same performance level as FIVER [RSS+21].

Software Fault Verification. ARMORY [HSP21] is a framework capable of analyzing the effects of faults on a program. Hereby, ARMORY provides the possibility of automatically injecting faults during the execution of a binary in an ARMv7-M emulator. 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 [Ris13] 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 has jointly modeled hardware and software in a framework based on the Yosys infrastructure [TAC+23]. However, modeling all these components together inside the same formal model leads to scalability issues as state-of-the-art hardware model checkers [NPWB18, MIL+21] cannot cope with the complex generated models. As shown in their previous paper [TAC+22], this monolithic approach can only verify small programs, e.g., a hundred instructions in 25 hours on similar CPUs in size. However, the authors consider a restricted fault model targeting only a subset of the possible fault locations. They highlight the consequences of microarchitectural

[8] https://github.com/lowRISC/ibex/pull/2117

faults on the running software and only provide security guarantees for this restricted fault model. In contrast, with our hardware verification methodology, we were able to prove the 1-security of the Secure Ibex with a 130 kGE circuit considering all possible bit-flips on any circuit gate. Moreover, by integrating hardware verification results into a co-verification framework, we can address previously intractable software verification, e.g., 2526 instructions in 2.6 hours (Table 2) on the OpenTitan platform. Note that this performance number does not include the time needed for the hardware verification. Furthermore, in comparison to the existing framework [TAC+23], our approach requires conducting the verification of the hardware only once, and the verification result, i.e., the reduced fault model, can be used to verify any program running on the same hardware.

8 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. As demonstrated in our paper, we provide unbounded fault verification proofs for the k-security of a circuit using *k-fault resistant partitioning*. If our methodology identifies a security vulnerability in hardware, we are incorporating the results from the hardware verification stage into the software verification step. 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.

A main future work is to extend the notion of k-fault resistant partitioning to support non-separable CED-based hardware countermeasures or mixed hardware/software protections, where software-only or hardware-only verification techniques cannot be used.

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 project (ANR-22-PECY-0004), 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.

A Impeccable Circuits Verification Results

We apply our methodology to hardware implementations of cryptographic primitives protected by code-based CEDs, as introduced by the Impeccable Circuits work [AMR+20]. This section generalizes our methodology and formally analyzes the k-security of these circuits.

Even if identifying an inductive invariant on cryptographic circuits is not as crucial as on a CPU since its operation time is generally bounded, working on such use cases provides us a reference for comparing our method with existing bounded verification techniques like FIVER [RSS+21] addressing multiple fault injections. In the following, we study four versions of the Skinny-64 symmetric block cipher implementing protections for up to 3 single bit-flips. Analysis results are reported in Table 3.

Table 3: Circuit characteristics and performance for k-fault resistant partitioning analysis applied to Impeccable Circuits [AMR+20]. Verifications have been executed on an Intel(R) Core(TM) i7-1185G7 CPU.

Circuit Characteristics				Faults		Partitioning		Analysis Results	
Name	**Code** [n, k, d]	**Size (GE)**	**Registers (#)**	**Locations (#)**	**Order** k	**Procedure 1 (s)**	**Procedure 2 (s)**	**Partitions (#)**	**Exploitable Faults (#)**
Skinny-64 red-1	$[5, 4, 2]$	3 270	235	2 097	1	1.18	0.043	235	128 inputs + 64 outputs
Skinny-64 red-2	$[6, 4, 2]$	3 718	269	2 234	1	1.48	0.056	269	
Skinny-64 red-3	$[7, 4, 3]$	4 163	305	2 371	1	1.32	0.052	305	
					2	9.06	0.324	305	
Skinny-64 red-4	$[8, 4, 4]$	6 316	341	4 142	1	2.48	0.127	341	
					2	10.23	0.404	341	
					3	57.89	0.693	341	+ 441 others

Our algorithm proves the 2-security of Skinny-64 implementations in less than 10 seconds. In addition, circuit inputs and outputs can be faulted as they are not protected with EDC. We achieve the same level of performance as state-of-the-art bounded verification tools like FIVER [RSS+21] applied on the size-equivalent circuit CRAFT.

However, our fault analysis fails to build a circuit partitioning with an attack order $k = 3$. Investigating the logs produced during procedure 1 shows that a fault in the Sbox, which is the only non-linear operation used in Skinny, diffuses the fault to multiple bits. Also, a fault injected in the checker mechanism can prevent a fault from being detected. As a result, collisions are found between the target and the prediction functions. Understanding if these collisions identified by our approach are spurious vulnerabilities or can be reproduced in actual encryption is left as future work. Assuming that faults cannot be injected in these circuit elements (i.e., 441 bits as reported by Table 3) is a sufficient condition to prove the 3-security of the circuit.

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+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.

[BDL] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. In *Advances in Cryptology – EUROCRYPT'*.

[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*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 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 *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, 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.

[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 - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*, 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 *Secure IT Systems - 24th Nordic Conference, NordSec 2019, Aalborg, Denmark, November 18-20, 2019, Proceedings*, 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.

[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.

[kok19] kokke. Tiny AES. https://github.com/kokke/tiny-AES-c, 2019.

[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.

[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 - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *LNCS*, pages 461–474, 2021.

[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 *Proceedings of the Great Lakes Symposium on VLSI 2023, GLSVLSI 2023, Knoxville, TN, USA, June 5-7, 2023*, 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 - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, 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.

[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 *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 213–222, 2014.

[Ris13] Riscure. Fisim. https://github.com/Riscure/FiSim, 2013.

[RSBG20] Jan Richter-Brockmann, Pascal Sasdrich, Florian Bache, and Tim Güneysu. Concurrent error detection revisited: hardware protection against fault and side-channel attacks. In *Availability, Reliability and Security – ARES'20*, pages 20:1–20:11, 2020.

[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/.

[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 2023, Ames, IA, USA, October 24-27, 2023*, 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 *17th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2002), 6-8 November 2002, Vancouver, BC, Canada, Proceedings*, 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.

[Wol23] Claire Wolf. Yosys open synthesis suite. https://yosyshq.net/yosys, 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.