SYNFI: Pre-Silicon Fault Analysis of an Open-Source Secure Element

Pascal Nasahl[†1,3], Miguel Osorio[1], Pirmin Vogel[2], Michael Schaffner[1], Timothy Trippel[1], Dominic Rizzo[1] and Stefan Mangard[3,4]

[1] Google, Mountain View, USA

[2] lowRISC CIC, Cambridge, United Kingdom

[3] Graz University of Technology, Graz, Austria
firstname.lastname@iaik.tugraz.at

[4] Lamarr Security Research, Graz, Austria

Abstract. Fault attacks are active, physical attacks that an adversary can leverage to alter the control-flow of embedded devices to gain access to sensitive information or bypass protection mechanisms. Due to the severity of these attacks, manufacturers deploy hardware-based fault defenses into security-critical systems, such as secure elements. The development of these countermeasures is a challenging task due to the complex interplay of circuit components and because contemporary design automation tools tend to optimize inserted structures away, thereby defeating their purpose. Hence, it is critical that such countermeasures are rigorously verified *post-synthesis*. Since classical functional verification techniques fall short of assessing the effectiveness of countermeasures (due to the circuit being analyzed when no faults are present), developers have to resort to methods capable of injecting faults in a simulation testbench or into a physical chip sample. However, developing test sequences to inject faults in simulation is an error-prone task and performing fault attacks on a chip requires specialized equipment and is incredibly time-consuming. Moreover, identifying the fault-vulnerable circuit is hard in both approaches, and fixing potential design flaws post-silicon is usually infeasible since that would require another tape-out. To that end, this paper introduces SYNFI, a formal pre-silicon fault verification framework that operates on *synthesized* netlists. SYNFI can be used to analyze the general effect of faults on the input-output relationship in a circuit and its fault countermeasures, and thus enables hardware designers to assess and verify the effectiveness of embedded countermeasures in a systematic and semi-automatic way. The framework automatically extracts sensitive parts of the circuit, induces faults into the extracted subcircuit, and analyzes the faults' effects using formal methods. To demonstrate that SYNFI is capable of handling unmodified, industry-grade netlists synthesized with commercial and open tools, we analyze OpenTitan, the first open-source secure element. In our analysis, we identified critical security weaknesses in the unprotected AES block, developed targeted countermeasures, reassessed their security, and contributed these countermeasures back to the OpenTitan project. For other fault-hardened IP, such as the life cycle controller, we used SYNFI to confirm that existing countermeasures provide adequate protection.

Keywords: Secure Root-of-Trust · Fault Injection · Countermeasure Verification · Pre-Silicon Analysis

[†]The work was done while the author was at Google.

1 Introduction

In a fault attack, an adversary induces a fault into a chip to manipulate the execution of the circuit. The physical effect of the fault can then be exploited to hijack the control-flow of a CPU [SD15, NT19], to bypass secure-boot [VTM+17], or to leak sensitive data of the device [BS97, PQ03, DEK+18]. Fault attackers often target secure elements, as they handle highly security-sensitive assets. For this reason, these root-of-trust (RoT) elements, such as OpenTitan [JRR+18], embed several hardware-based fault hardening techniques into the chip. As the resistance of the circuit against faults relies on these countermeasures, it must be assured that they provide the expected security guarantees.

To ensure the correctness of the countermeasures, hardware engineers responsible for designing secure chips must analyze the circuit when influenced by faults in the design phase. This pre-silicon evaluation needs to comprise two central analysis points: First, *can induced faults influence the input-output relation of a security-critical circuit and can the countermeasures detect them?* Here, the hardware designer wants to reveal whether a fault affects the circuit and to verify that the countermeasure achieves the promised security level, *i.e.*, can handle up to a certain number of simultaneously induced faults specified in the threat model. Second, *can the embedded countermeasures hinder an adversary from entering a specific, security-critical circuit state using faults?* An example of such a state is the debug mode of a secure element allowing the adversary to escalate privileges.

Testing the resilience of the circuit and its countermeasures against faults needs to be conducted in the last stage of the front-end design, *i.e.*, at synthesized gate-level netlist. This approach ensures that *(i)* defective countermeasures are detected as early as possible avoiding long design turnaround times. Additionally, at this *(ii)* level of abstraction, the design uses the standard cell library provided by the manufacturer and, therefore, is already close to the final circuit sent to the fab for the tape-out. Furthermore, performing the security assessment at the netlist ensures *(iii)* that flaws introduced by the tooling can be detected. Here, especially the logic synthesis design flow step mapping the register-transfer level (RTL) model to the synthesized gate-level netlist could negatively affect countermeasures using redundancy to detect or mitigate faults. Here, the synthesis optimization passes aiming to meet design constraints, e.g., the area consumption, could be responsible of reducing security guarantees.

One approach of analyzing the resilience of the circuit against faults at the netlist level is to manually induce faults and to analyze their effect in the simulation phase. However, as the names of the wires and cells in the netlist are renamed or mangled by synthesis tools, manually inducing faults in the testbench is an error-prone task. Additionally, the analysis process is very time-consuming since the simulation needs to be restarted for each induced fault. Hence, this process is often at risk of being foregone in the verification phase of the design due to development schedule pressure.

In order to verify the functionality of fault countermeasures embedded into the chip, a framework capable of automatically performing a pre-silicon analysis based on the synthesized gate-level netlist is needed. It is crucial for such a tool to be capable of handling industry-grade netlists using proprietary standard-cell libraries without imposing any restrictions on the netlist and the design. Otherwise, such restrictions would render the tool practically irrelevant, especially for commercial projects that rely on established hardware design flows with a multi-stakeholder design team.

Recently published tools [BGE+17, AWMN20, RBSS+21, BDN08, SKK13] cannot be used to analyze unmodified netlists of industry-driven projects, as these frameworks impose invasive requirements to the design. Tools, such as FIVER [RBSS+21], limit *(a)*, the supported gates in the netlist to a small set, preventing the usage of complex, proprietary standard cell libraries. Furthermore, most of these frameworks [RBSS+21] require *(b)* that the given netlist does not include any cycles, *i.e.*, the hardware designer needs to manually unroll the design before the evaluation. Additionally, related work often demands *(c)* that

the netlist is fully flattened, *i.e.*, does not include any submodules or hierarchy, does not support *(d)* all language features, or is not *(e)* open-source. Finally, as most fault injection frameworks [BGE+17, AWMN20, BDN08] exclusively focus on analyzing cryptographic primitives *(f)*, it remains unclear whether these frameworks also can be used to assess the security of more generic hardware designs consisting of a diverse set of hardware IP components, especially in respect with the two analysis points described above.

Contribution

In this paper, we present SYNFI, a versatile framework capable of performing a pre-silicon fault analysis of synthesized gate-level netlists. SYNFI allows a hardware designer to automatically analyze the resilience of a circuit and its countermeasures against fault attacks with minimal setup overhead. More specifically, SYNFI enables hardware designers and security engineers to study the impact of faults on the circuit, to analyze the functionality of tailored fault countermeasures, and to investigate which cells are the most critical attack targets and need special protection. This information can be used to find logical flaws in the design as well as defects introduced by the hardware design flow tools before the tape-out of the chip.

The SYNFI framework is capable of performing the pre-silicon fault analysis on unmodified netlists generated with proprietary or open design flows and standard cell libraries of designs using common hardware design patterns. For the fault experiment, the security engineer needs to provide information about the circuit to analyze and the fault model. SYNFI supports fault models comprising single and multiple faults injected into various locations in the circuit and different fault effects, *i.e.*, transient or stuck-at effects. With this configuration, SYNFI automatically extracts the circuit to analyze from the netlist and injects faults according to the fault model. In the analysis phase, SYNFI reveals whether a fault affects the input-output relation of the circuit, shows whether the embedded countermeasures can detect faults up to a certain number, and verifies whether a fault could enable an adversary to enter a security-critical state.

To emphasize the importance of conducting a pre-silicon fault analysis before an upcoming tape-out, we utilize SYNFI to analyze components of the OpenTitan secure element. In particular, we focus on analyzing the fault-resiliency of the most security-critical components, such as the AES primitive, the life cycle controller, the lockstep mode of the processor, and several other fault hardened IP. For our assessment, we study the impact of single and multiple faults induced into different parts of the modules for various fault effects. We conduct our analysis on the unmodified netlist generated with the internal, proprietary hardware design flow of OpenTitan including a commercial standard cell library as well as on the netlist synthesized with open-source tools. We utilize SYNFI to *(i)* reveal the impact of faults to unprotected circuits, to *(ii)* verify that the redundancy-based countermeasures are not removed by the synthesis tool, and *(iii)* to verify whether certain security-critical states cannot be entered using faults without triggering the countermeasures. Our in-depth analysis of the tested modules revealed that the AES module is highly susceptible to fault attacks. More concretely, our evaluation disclosed that already a single fault into the AES round counter, the handshake signals, or certain finite-state machines allow an adversary to break the security of the module. To mitigate the encountered security violations, we developed several fault hardening mechanism and integrated them into the OpenTitan project. We ensured the correctness of these countermeasures by reassessing the hardened module using SYNFI. For fault-hardened modules, such as the life cycle controller, we were able to formally verify the expected fault-resiliency.

In summary, our contributions are:

- We present and implement SYNFI, an open-source[1] framework capable of performing a pre-silicon fault analysis at the gate-level. SYNFI allows security engineers to automatically *(i)* reveal whether a fault affects the input-output relation of a circuit and its countermeasures and *(ii)* assess if an adversary can enter a particular circuit state without triggering the countermeasures. In contrast to related work, the SYNFI framework is able to process unmodified netlists of hardware designs making use of a variety of design patterns and synthesized with commercial and open-source synthesis tools.
- We identified several fault attack vectors for the unprotected AES module used in the OpenTitan secure element allowing an adversary to threaten the security of the encryption primitive. To prevent the exploitation of these flaws in the final taped-out chip, we implemented, reassessed, and contributed several fault hardening techniques to the upstream project.
- We verified with SYNFI that a selection of the most security-critical OpenTitan IP blocks hardened against faults provides the expected security. In particular, we verified, among other modules, that an adversary cannot hijack the life cycle controller to enter the RMA debug state from the production state and that a fault into the program counter of the processor is detected by the lockstep mode of the CPU.

2 Background

In this section, we summarize fault attacks and provide detailed background on the OpenTitan project.

2.1 Fault Attacks

Fault attacks are active, physical attacks that are commonly used to threaten the security of embedded devices [DM12, TM17, O'F20, ELG20] and secure elements [Hér, VWWM11, SWUH21]. In these attacks, a fault is induced into the chip causing several effects at the physical level, e.g., transient voltage and current changes as well as timing violations [RBSG21]. These side-effects are then exploited allowing an adversary to bypass security measures [VTM+17, TM17], attack cryptographic primitives [BS97, PQ03, DEK+18], or redirect the control-flow [SD15, NT19] of the executed software. The fault model, which is used to characterize such attacks, comprises the fault methodology, the spatial and temporal properties, and the effect of the fault. The spatial and temporal properties of the fault model define the location, the duration, and the time of the induced fault. Although, depending on these different fault parameters, the effect of a fault varies, commonly bit-flips and stuck-at effects are observed [RU96]. To induce a glitch into a system, various fault methodologies, such as voltage, clock, laser, and EM glitching [KSV13], emerged in the previous years. While these fault methodologies originally only could be performed locally, new fault methodologies even allow inducing faults in software over the network [MOG+20, TSS17, QWLQ19].

2.2 OpenTitan

Secure elements and RoT chips are used in smartphones [Li20], computers [App21, Goo22], and servers [Bro21] to establish a secure anchor point. These elements are trusted by the system and offer various services, such as cryptographic functions, key storage and

[1] https://github.com/lowRISC/synfi

support for secure boot protocols. As a security breach could be fatal, these integrated circuits typically offer a certain level of protection [RLMI21] against fault attacks. RoT chips introduced so far are closed, proprietary designs making it necessary for the system integrator to trust the manufacturer of these devices. The OpenTitan [JRR+18] project aims to obviate this requirement by providing the first open-source root-of-trust chip. However, as the silicon design of the chip is open-source, an attacker also could discover potential attack vectors. Hence, it must be assured that the installed countermeasures work as intended by using a rigorous verification approach.

3 Design and Implementation

This section describes the fundamental concepts of the SYNFI framework along with the design rationale. We first give a high-level overview of the framework and then provide an in-depth description of all the design stages of SYNFI.

3.1 Overview

To analyze the effects of one or multiple faults to the input-output relation of a circuit and its fault countermeasures, the gate-level netlist, the used standard cell library, as well as a fault specification need to be provided to the SYNFI framework.
Netlist & Cell Library: The first input of the SYNFI framework is the unmodified netlist of the module to analyze and the standard cell library that the design is mapped against. As shown in the block diagram in Figure 1, the synthesis design flow step, which

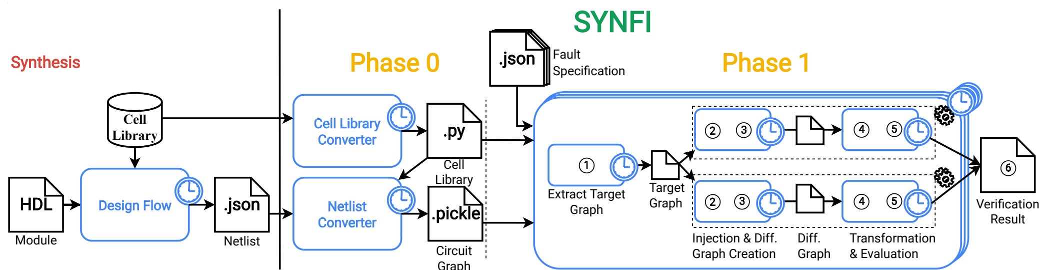


Figure 1: Block diagram of the SYNFI framework.

is not part of the SYNFI framework, is responsible for transforming the RTL design into the netlist using the standard cell library.
Fault Specification: The second input of the framework is the fault specification file responsible for describing the fault experiment the designer wants to perform.

```
1  "Fault Specification": {
2      "Target Circuit": {
3          "inputs": ["in_port1": "2'b00", "in_port2": "2'b01"],
4          "outputs": ["out_port1": "2'b11", "in_port2": "2'b10"]
5      },
6      "Fault Model": {
7          "Simultaneous Faults": 2,
8          "Fault Locations": ["gate1", "gate2", "..."],
9          "Fault Mappings": ["NAND2": ["AND2", "OR2"], "XNOR2": ["XOR2"]]
10     }
11 }
```

Listing 1: Fault specification.

As shown in Listing 1, the fault specification file is split into *(i)* the description of the subcircuit the designer wants to evaluate and *(ii)* the fault model describing the faults injected into this subcircuit. The subcircuit to analyze *(i)* is defined by the user by providing input and output nodes, e.g., input or output ports, cells, or submodules of the design. Furthermore, the user needs to assign inputs and expected output values for the provided nodes. The fault model *(ii)* describes all faults which are induced into the subcircuit. Here, the fault model consists of the *(a)* number of faults injected into the circuit, the *(b)* location, and the *(c)* effects of the faults. With the number of faults *(a)*, the user can specify how many faults are simultaneously injected into the subcircuit. In SYNFI, a fault is injected into *(b)* a certain location, *i.e.*, a gate. Here, the user can either provide a list of gates which are attacked or select an exhaustive approach where SYNFI automatically injects faults into all gates. The last parameter is the *(c)* fault effect. Similar to [RBSG21], we model the effect of a fault induced into a certain gate by replacing the boolean function of the gate type according to a mapping. For example, the mapping `NAND2=[AND2]` replaces a gate of type `NAND2` during the attack phase with an `AND2`. Here, by inverting the boolean function, SYNFI is capable of modeling a transient fault effect. To model a stuck-at 0 or 1 fault, the boolean function of the corresponding gate can be set to a 0 or 1 in the fault mapping. By providing multiple entries in the fault mapping, e.g., `NAND2=[AND2, 0]`, SYNFI can be used to analyze the circuit when influenced by transient or stuck-at faults. Summarized, this mapping enables SYNFI to model transient, stuck-at, or more advanced fault effects. For each subcircuit the user wants to analyze, a new fault specification file needs to be provided and SYNFI needs to be started again.
SYNFI: With the unmodified netlist, the standard cell library, and the fault specification, the tool starts the two-phase transformation and analysis process depicted in Figure 1. In *Phase 0*, the framework transforms the netlist into a directed multigraph and converts the cell library into a format the subsequent steps of the SYNFI framework support. In *Phase 1*, the subcircuit to analyze, *i.e.*, the target graph, is ① extracted from the circuit according to the fault specification file. Afterwards, for each fault location and fault mapping combination, a separate process is started. In these processes, two copies of the target graph are created, *i.e.*, the faulty and non-faulty target graph. SYNFI induces ② faults according to the fault mapping (number of simultaneous faults, location, and mapping) into the faulty target graph by replacing the boolean functions of the target gates according to the mapping. By combining the faulty and non-faulty target and adding an input and output layer responsible for analyzing the effects of the induced faults, the differential graph ③ is created. This differential graph is used by SYNFI to evaluate if a fault is effective, *i.e.*, the fault manipulates the outputs of the faulty target graph and is not detected by the countermeasures. Although the detection of faults by the countermeasures is implementation specific, these countermeasures typically raise an error signal, which SYNFI uses to evaluate whether the fault was detected or not. Finally, the differential graph is converted to a boolean formula ④ and a SAT solver utilizes this mathematical model representing the circuit to reason ⑤ about the effectiveness of the induced faults. In the end, the framework provides a detailed report ⑥ summarizing the outcome of the fault analysis.

3.2 Phase 0 - Cell Library & Netlist Converter

The first step the framework conducts is the transformation of the *(i)* standard cell library and the *(ii)* gate-level netlist. The goal of this transformation step is to support arbitrary netlists generated with different hardware design flows and standard cell libraries.

First *(i)*, SYNFI converts the provided standard cell library from the liberty format to a Python library. For this conversion, SYNFI opens the provided standard cell library and extracts the name, the boolean function, and the input and output pins of the cells, as shown in Listing 2. SYNFI supports all cells with a boolean function, including compound

```
1 "Cell Library": {
2     "AOI21_X2": {
3         "input_pins": ["A1", "B1", "B2"],
4         "output_pins": "ZN",
5         "boolean_function": "ZN = !(A1 & (B1 | B2))
6     }
7 }
```

Listing 2: Cell library entry for an `AOI21_X2` cell.

gates, such as `AOI` cells. Cells that are used due to their electrical rather than for their logical behavior, e.g., filler cells, are not handled by SYNFI as they are not used in the gate-level netlist.

Afterwards *(ii)*, SYNFI transforms the unmodified netlist into a directed multigraph using a Python library [HSSC08].

```
1  "Nodes": {
2      "U1": { "type": "NAND2" },
3      "U2": { "type": "AOI21" }
4  },
5  "Edges": {
6    "1": {
7      "out": { "node": "U1", "port": "ZN" },
8      "in": { "node": "U2", "port": "A1" }
9    }
10 }
```

Listing 3: Graph representation of the netlist.

In this graph, nodes represent ports, cells, and submodules and each of these nodes consists of a name and a type. The type, e.g., a `NAND2` gate or a port, defines the behavior of the node and the corresponding boolean function is provided by the cell library. Similar to gates and ports, submodules are also represented as nodes and the corresponding boolean function needs to be provided by the user. These nodes are connected using edges, which store information about the input and output port. Listing 3 shows an example graph where the output port `ZN` of the gate `U1` is connected with the input port `A1` of the gate `U2`.

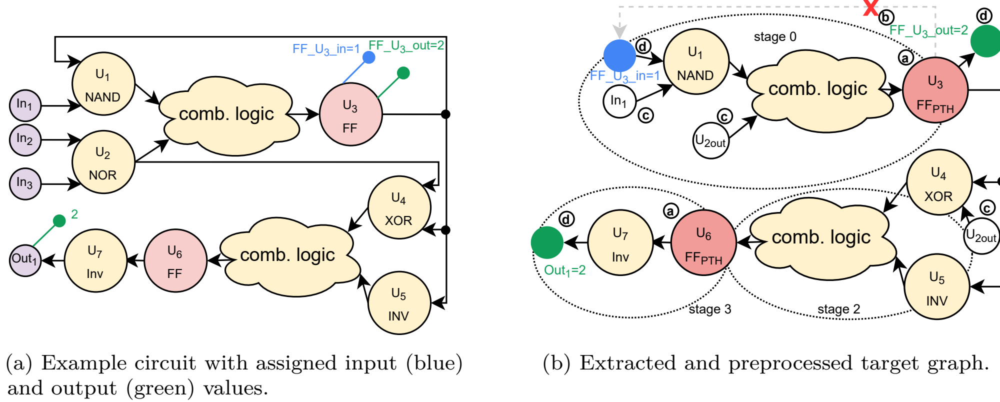
3.3 Phase 1 - Target Graph Extraction

The target graph extraction ① step consists of the *(i)* extraction and *(ii)* preprocessing phase.

3.3.1 Extraction

The goal of the target graph extraction *(i)* is to simplify the subsequent analysis phase by extracting the subcircuit the user wants to analyze with SYNFI from the overall circuit. The definition of the target graph is provided in the fault specification file. Here, the user needs to define input and output nodes and the corresponding input and expected output values in the fault specification file. These nodes can be any cell, port, or submodule in the circuit.

With this information, the SYNFI framework starts the automatic target graph extraction process. Here, the tool finds all paths, consisting of combinational and sequential logic, between the defined inputs and outputs. Due to this extraction step, some nodes, e.g.,



(a) Example circuit with assigned input (blue) and output (green) values.

(b) Extracted and preprocessed target graph.

Figure 2: Target graph extraction.

gates, are missing one or multiple inputs as the corresponding connecting gates are not part of the extracted circuit. For all of these missing inputs, SYNFI introduces auxiliary input nodes.

3.3.2 Preprocessing

The goal of the preprocessing phase *(ii)* is to remove any time-dependencies in the extracted target graph. This is necessary as the graph is converted into a time-independent mathematical model, *i.e.*, a boolean equation, in the last step described in Section 3.6. In SYNFI, we automatically break time-dependencies by *(a)* replacing registers used in pipeline stages with pass-through elements and *(b)* by removing loops and replacing registers in iterative designs and state machines. These pass-through elements are time-independent, *i.e.*, do not have a clock port, and map the input to the (negated) output. SYNFI automatically distinguishes between the two register types by checking whether the register is the start and end of a cycle, *i.e.*, a register used in an iterative design. This preprocessing phase enables SYNFI to handle circuits that were not manually unrolled by the hardware designer. However, when aiming to analyze multiple loop iterations, e.g., multiple rounds in an iterative AES implementation, SYNFI needs to evaluate each round individually.

3.3.3 Preprocessing and Extraction Example

We illustrate the extraction *(i)* and preprocessing *(ii)* phase in the example circuit in Figure 2. Figure 2a depicts a circuit consisting of three input ports ($In_1 ... In_3$), one output port (Out_1), two registers (U_3, U_6), and a set of combinational gates. For the target graph extraction step ①, the user needs to provide input and output nodes and a corresponding circuit state, *i.e.*, values for these nodes. In this example, we set the register $U_3 = 1$ (blue) as the input and the register $U_3 = 2$ and $Out_1 = 2$ (green) as the output in the fault specification file.

As the circuit contains a register used in a pipeline stage (U_6) and a register used in a sequential loop (U_3), SYNFI removes these time-dependencies in the graph. The registers are, as shown in Figure 2b, replaced ⓐ with pass-through elements and the loop between U_3 and U_1 is removed ⓑ. Then, SYNFI finds all paths between the input node (U_3) and the output nodes (U_3, Out_1), *i.e.*, all nodes except U_2, In_1, In_2, and In_3. To avoid that certain gates have unconnected inputs, *i.e.*, U_1, the frameworks adds auxiliary input nodes

ⓒ and connects them with the corresponding nodes. Finally, the framework adds input and output nodes ⓓ for the user-defined input and output values.

3.4 Phase 1 - Fault Injection

After transforming the netlist into a graph and extracting the target graph, the injection ② phase starts. For each fault combination, *i.e.*, number of simultaneous faults, fault locations, and fault mappings, defined in the fault model, SYNFI starts a new process. Inside these processes, SYNFI creates two copies of the extracted target graph - the faulty and non-faulty target graph. While the non-faulty target graph is used as a reference circuit in the subsequent steps, SYNFI induces faults into the faulty target graph. Here, the framework replaces the boolean function at a fault location according to the fault mapping.

To limit the configuration effort, SYNFI already provides a default fault mapping for a large set of gates. Furthermore, as the fault location is an optional parameter allowing the security engineer to attack specific gates, SYNFI supports an exhaustive injection approach automatically targeting all available gates in the target graph. Hence, at a minimum, the user only needs to specify the number of simultaneous faults injected into the gate-level netlist in the fault model.

3.5 Phase 1 - Differential Graph Creation

For each fault combination process, SYNFI creates a differential graph ③ consisting of a faulty and non-faulty target graph. These differential graphs are responsible for evaluating the impact of faults on the circuit. As depicted in Figure 3, the differential graph consists

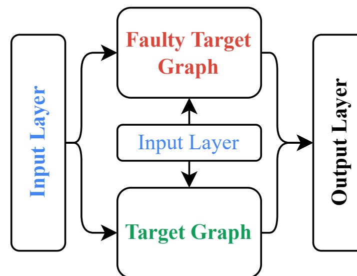


Figure 3: Differential graph.

of the faulty and non-faulty target graph. To this differential graph, we add an input layer and an output layer. In the input layer, we assign the input nodes added in the extraction phase ① the values provided by the user in the fault specification. As the user does not need to provide all possible input values of the analyzed circuit, SYNFI automatically connects the non-defined inputs of the faulty target graph with the non-defined inputs of the non-faulty target graph. The SAT solver, which is used in the evaluation step and described in Section 3.6, then automatically assigns values to these non-defined inputs. The output layer is used to analyze the effect of a fault. This layer consists of a logic comparing the output values produced by the faulty and non-faulty target graph with the output values provided by the user in the fault specification. Depending on the attack objectives and the implemented countermeasures, SYNFI allows the hardware designer to define two different types of effective faults, which are defined by the impact of the fault to the output. The detection of these two different effective fault types is implemented in the output layer.

3.5.1 Unspecific Fault Effects

This type of effective fault enables SYNFI to generically reveal whether a fault influences the input-output relation of a circuit. For circuits without countermeasures, SYNFI defines a fault to be **effective**, *iff* this fault manipulates one or multiple output bits of the analyzed subcircuit, *i.e.*, the non-faulty and faulty target graph produce different output values.

$$OutputLogic = (O_{NF}{==}O_E) \land (O_F{!=}O_E) \quad (1)$$

Equation 1 depicts the logic in the output layer used to detect this type of fault effect. The first part of the equation ensures that the output O_{NF} of the non-faulty (NF) graph produces the expected output value O_E defined in the fault specification. More specifically, this part of the equation ensures that the SAT solver only assigns values to the non-defined inputs (cf. Section 3.5) of the differential graph which generate the expected output circuit state O_E. The second part of the equation is responsible for ensuring that the output O_F of the faulty (F) circuit does not match the expected output value. If the output logic produces a logical 1, an **effective** fault is found.

$$OutputLogic = ((O_{NF}{==}O_E) \land (O_{NFA}{==}0)) \land ((O_F{!=}O_E) \land (O_{FA}{==}0)) \quad (2)$$

For circuits with dedicated fault countermeasures, SYNFI considers a fault to be **effective**, *iff* this fault manipulates one or multiple output bits of the analyzed subcircuit *and* the alert signal of the countermeasure was not triggered. If the alert signal was triggered, the countermeasure works as intended and the fault is considered to be **ineffective**. The output logic in Equation 2 models this behavior by ensuring that the alert signal O_{FA} was not triggered in both parts of the formula.

3.5.2 Specific Fault Effects

This type of fault effect allows SYNFI to check whether a fault enables the adversary to enter a specific circuit state. Here, SYNFI considers a fault to be **effective**, *iff* the output of the faulty target graph matches the expected output value defined in the fault specification.

$$OutputLogic = (O_{NF}{==}O_E) \land (O_F{==}O_{EF}) \quad (3)$$

Equation 3 shows the logic in the output layer capable of detecting this fault effect type. Here, the first part of the equation ensures that the outputs O_{NF} produced by the non-faulty target graph match the expected output values O_E provided in the fault specification. The second part of the equation ensures that the outputs of the faulty graph match the expected fault output value O_{EF} specified in the fault specification.

$$OutputLogic = ((O_{NF}{==}O_E) \land (O_{NFA}{==}O_{EA})) \land ((O_F{==}O_{EF}) \land (O_{FA}{==}O_{EA})) \quad (4)$$

For circuits consisting of a fault countermeasure designed to detect a fault, the alert signal O_{EA} is also incorporated in the output logic, as shown in Equation 4. Here, the output layer ensures that the non-faulty circuit produces the expected output values ($O_{NF}{==}O_{E_0}$) and that the alert was not triggered in the reference circuit, *i.e.*, ($O_{NFA}{==}O_{EA}$). For the faulty target graph, the equation ensures that the output value matches the expected fault output value ($O_F{==}O_{EF}$), and the alert was not triggered.

3.6 Phase 1 - Transformation & Evaluation

After creating the differential graph, the SYNFI framework converts this graph ④ into a mathematical model. As each node is assigned a boolean function, the tool uses the Tseitin transformation [Tse83] to automatically transform the differential graph into a boolean formula in conjunctive normal form (CNF). The extracted boolean formula then is handed over to a SAT solver for the evaluation ⑤. As shown in the differential graph in Figure 3, the inputs of the boolean formula are either set in the input layer to values provided by the user in the fault specification or are left unconnected. For these unconnected input values, which are shared by the faulty and non-fault target graph, the SAT solver can set these values freely as long as the reference circuit produces the expected output values. This is ensured by the output layer of the differential graph (cf. Section 3.5). If the logic in the output layer produces a logical 1, an effective fault is found. For the report ⑥, the framework collects the number of effective faults, their location, and fault mapping.

3.6.1 Selection of the SAT Solver

For our Python-based tool, we use the PySAT [IMM18] framework as an interface to the SAT solver. To determine the fastest solver for our purpose, we executed several fault injection verification experiments with the provided solvers [BFFH20, ES03, Bie17, AS18, LM21, L+18] as a custom benchmark and decided to use MiniSAT22 [ES03] in the end.

3.7 SYNFI Guarantees

SYNFI provides hard security guarantees for a specific fault experiment conducted on the analyzed circuit. This fault experiment is defined by the security engineer analyzing the circuit in the fault specification and is in line with the threat model of the design. The fault specification consists of the *(i)* definition of the fault model and the *(ii)* description of the target circuit.

In the fault model *(i)*, the SYNFI user defines the fault capabilities of the attacker, which are specified in the threat model of the analyzed circuit. This definition comprises the number of faults the attacker can simultaneously inject into the circuit, the effects, and the locations of the faults. For the fault locations, the security engineer can either target specific gates or instrument SYNFI to exhaustively inject faults into all gates of the circuit. SYNFI injects a fault into these targeted gates by replacing the boolean function of the gate according to the fault mapping specified in the fault model. Here, SYNFI supports transient or permanent fault effects.

The target circuit *(ii)* is the subpart of the overall circuit containing the security-critical logic and the corresponding fault countermeasure the security engineer aims to analyze with SYNFI. This circuit is defined in the fault specification by providing the names of input and output ports of a module or certain gates. SYNFI then automatically extracts the target circuit between these inputs and outputs. In addition to the names of these ports or gates, the SYNFI user needs to specify a specific circuit state, *i.e.*, values for the inputs and outputs.

Depending on the configuration, SYNFI can reveal whether a fault has *(a)* an unspecific (cf. Section 3.5.1) or *(b)* a specific (cf. Section 3.5.2) effect. More concretely, SYNFI can formally verify *(a)* whether or not any fault specified in the fault model can change the input-output relation of the target circuit without triggering the fault countermeasures. Additionally, SYNFI can formally show *(b)* whether or not it is possible to enter a specific circuit state from a given circuit state without triggering the countermeasures using a fault.

Note that SYNFI is designed to detect faults manipulating the input-output relation of the analyzed circuit. Hence, classes of fault attacks not impacting this relation, e.g., safe error [YJ00] or ineffective attacks [DEK+18], are not in the scope of SYNFI.

When the input circuit state space is small, e.g., a counter logic, multiple fault experiments for each possible circuit state can be conducted. Then, SYNFI provides comprehensive security guarantees for the analyzed circuit. For larger circuit state spaces, the security engineer needs to focus on verifying specific states which are particularly security-sensitive or are a representative of the possible states.

False-positive Results. SYNFI can produce false-positive results when the target circuit is too loosely specified. As described in Section 3.5, the SYNFI user does not need to provide the enire input circuit state in the fault specification. The non-defined inputs provide more freedom to the SAT solver and the solver can freely set these inputs as long as the non-faulty target graph produces the specified output circuit state.

However, in some circuits, the SAT solver could find a circuit state which cannot occur during normal operation. Then, a false-positive result is returned, requiring a manual inspection of unexpected effective faults.

Note that the approach of SYNFI is to consider states that occur during normal operation and analyze how a fault changes the behavior. Using a faulty starting state means analyzing how a fault can change a faulty starting state. This is a fault that is beyond the defined fault model and actually corresponds to a stronger fault model. When a false-positive like this occurs, the security engineer can simply manually exclude it or constrain SYNFI more tightly to avoid the need for manual inspection. In some cases, the false positive may also provide a hint to the security engineer about faults that can occur with stronger fault models and this can be an input for an extended analysis with a stronger fault model.

Overall, in our analysis, fault positives have not turned out to be a severe limitation as effective faults have occurred rarely in fault-hardened circuits and it was possible to handle them by constraining SYNFI more tightly to specific circuit states.

False-negative Results. SYNFI cannot produce false-negative results within the bounds of the fault specification. The security engineer only needs to ensure that the fault specification matches the threat model of the analyzed circuit. For example, when the threat model considers an attacker capable of injecting faults with permanent or transient effects, the fault model also needs to model these faults in the fault mappings.

4 Analysis of OpenTitan

The OpenTitan chip will be deployed in hostile environments allowing an adversary to gain physical access and attempt to inject faults into the device to break its security. Therefore, in this chapter, we utilize SYNFI to actively contribute to the security of the OpenTitan chip before the tape-out by performing a pre-silicon fault analysis. As analyzing the entire chip consisting of a wide variety of IP blocks is far beyond the scope of this paper and not all modules actually need to provide fault-resiliency, we selected, together with the OpenTitan project team, the most security-critical modules for our analysis. In particular, we focused on analyzing *(i)* the unprotected AES module and *(ii)* the protected life cycle controller, the lockstep mode of the CPU, and generic, fault-hardened building blocks. We utilized SYNFI to **(FE)** reveal the faults' effect to an unprotected module, to **(FD)** check whether faults can be detected by the countermeasures, and to **(FS)** verify that faults cannot enable an adversary to enter a specific state without triggering the countermeasures. For all experiments, we injected up to a certain number of simultaneous faults specified in the threat model of each module into the circuit. Our analysis is conducted on the unmodified netlist synthesized with the internal OpenTitan hardware design flow consisting of the Synopsys DC synthesis tool and a proprietary standard cell library.

Results. With SYNFI, we revealed that the *(i)* AES module is susceptible to single faults

Table 1: Verification results for the AES round counter performed on a 16-core machine.

Target	Setting	Simult. Faults	Effective [%]	Total [#]	Execution [s]	Circuit [GE]
① Unprotected Round Counter	**FE**	1	55.56	18	4.4	20.25
② Unprotected Round Counter	**FS**	2	2.65	302	4.47	20.25
③ Protected Round Counter	**FD**	2	0.13	34,652	366.81	156

enabling an adversary to perform attacks on a round-reduced AES, extract temporary encryption results over the software interface, or hijack the execution-flow of the AES FSM. For the other analyzed modules *(ii)*, our analysis showed that they provide adequate protection, *i.e.*, all modules can withstand or detect at least single fault attacks.

4.1 AES

The AES module of OpenTitan is a hardware accelerator providing a secure encryption and decryption mechanism for protocols used by the chip. As this IP block is one of the most crucial elements of the RoT element, we analyze in detail the behavior of the most security-critical parts of the module when influenced by faults. In comparison to related work (cf. Section 5), we focus on assessing generic hardware primitives, such as state machines and counters, instead of performing specific cryptographic data-flow attacks, such as SIFA [DEK+18] or DFA [PQ03].

Results. Our analysis revealed several fault attack vectors for the unprotected AES module. In particular, SYNFI showed that single faults into the AES round counter, handshake signals, and certain FSMs could enable an adversary to break the security of the module. Based on these verification results, we developed several fault hardening techniques, reassessed their security, and contributed them to the OpenTitan project.

4.1.1 AES Round Counter

The AES [DR99] block cipher performs, depending on the mode of operation, a certain number of encryption rounds. This round counter, which is generated in an FSM, is security-critical, as a fault hijacking the counter value could weaken the cryptographic strength of the AES [Bir04].

Unprotected round counter. To analyze the resilience of the round counter against faults, we first ① utilize SYNFI to reveal if the round counter circuit is generally susceptible to faults, *i.e.*, it is possible to arbitrarily manipulate the counter value. Then ②, we determine how many simultaneous faults are required to manipulate the counter to a specific value.

```
1 "Fault Specification":
2     "Target Circuit":
3         "inputs": ["rnd_ctr_q": "4'b0001"],
4         "outputs": ["rnd_ctr_d": "4'b0010"]
5     "Fault Model":
6         "Sim. Faults": 1 or 2, "Fault Locations": ["*"]
```

Listing 4: Fault specification for the round counter.

To conduct this analysis, we describe the circuit of interest and the fault model in the fault specification file as shown in Listing 4. We configure SYNFI to analyze the logic in between the `rnd_ctr` register responsible for incrementing the value and set the input value of the counter circuit to 1 and the expected output value to 2. For the fault model, we instrument SYNFI to exhaustively induce one or two simultaneous faults into all available gates of the

circuit. We provide and describe the used fault model configuration to verify the round counter in more detail in Appendix A.1.

Table 1 shows the evaluation report generated by SYNFI. The setting column in the table specifies how SYNFI considers a fault to be effective. In the **(FE)** mode, any fault having an arbitrary effect to the input-output relation of the circuit is considered to be an effective fault. In **(FD)**, an effective fault is a fault manipulating the circuit's output and the fault countermeasures did not detect, *i.e.*, did not trigger the alert signal, this fault. Finally, **(FS)** refers to a fault changing the output of the circuit to a specific state without triggering the countermeasures. Moreover, the table shows the total number of injected faults and the percentage of the effective faults. The effective fault percentage number indicates how many of the total number of injected faults SYNFI considers to be effective. Finally, the table highlights the execution time of SYNFI and the circuit size in gate equivalent (GE). Note that the circuit size refers to the fault affected target circuit extracted by the SYNFI framework, which is a subcircuit of the whole circuit.

As shown in the first row ①, a single fault into the circuit enables a fault attacker to manipulate the round counter value. To manipulate the round counter to a specific value, SYNFI reveals in the second row ② that an adversary needs to induce at least two simultaneous faults.

Hardened round counter. To enhance the resilience of the counter against faults, we extend the FSM to generate an up counting (the round counter) and a redundant down counting counter value. We redundantly instantiate this FSM, combine the generated counters, and add an error logic capable of detecting an ongoing fault attack.

```
1  // Instantiate redundant FSMs.
2  for (genvar i = 0; i < 3; i++) begin : gen_fsm
3    aes_cipher_control_fsm u_aes_cipher_control_fsm_i (
4      .rnd_ctr_q_i     ( rnd_ctr_q          ),
5      .rnd_ctr_d_o     ( mr_rnd_ctr_d[i]    ),
6      .rnd_ctr_rem_q_i ( rnd_ctr_rem_q      ),
7      .rnd_ctr_rem_d_o ( mr_rnd_ctr_rem_d[i] ),
8      ...);
9  end
10 // Combine counter signals.
11 always_comb begin : combine_counter_signals
12   for (int i = 0; i < 3; i++) begin
13     rnd_ctr_d     |= mr_rnd_ctr_d[i];
14     rnd_ctr_rem_d |= mr_rnd_ctr_rem_d[i];
15   end
16 end
17 // Generate sum.
18 assign rnd_ctr_sum = rnd_ctr_q + rnd_ctr_rem_q;
19 assign rnd_ctr_err = (rnd_ctr_sum != num_rounds_q) ? 1'b1 : 1'b0;
```

Listing 5: Round counter protection in the `aes_cipher_control` module.

To ensure that the synthesis tool does not weaken the redundancy-based protection mechanism shown in Listing 5, we reassess its security using SYNFI. In particular, we utilize the framework to evaluate whether the circuit is capable of detecting a single fault arbitrarily manipulating the counter value **(FD)**.

SYNFI could formally verify that, in a specific circuit state, a single fault cannot manipulate the counter value without triggering the alert signal. This specific circuit state comprises a fixed counter input value of 1 and a counter output value of 2, all other non-defined inputs of the circuit are automatically set by the SAT solver SYNFI internally uses. We argue that testing a single circuit state, *i.e.*, an input-output pair, is sufficient to verify that the tooling of the design flow does not remove the redundancy-based countermeasures. For two simultaneous faults, SYNFI reveals in Row ③ in Table 1, that

Table 2: Verification results for the AES handshake signal on a 16-core machine.

Target	Setting	Simult. Faults	Effective [%]	Total [#]	Execution [s]	Circuit [GE]
① Unprotected Handshake Signal	**FS**	1	83.83	31	4.49	32
② Protected Handshake Signal	**FS**	3	15.32	8436	270.68	38

at least one fault into the error logic and one fault into the input or output shared round counter register are required to tamper the counter value without raising the alert.

4.1.2 AES Handshake Signals

Internally, the AES IP consists of a variety of handshake signals responsible for influencing the data- and control-flow of the encryption. As manipulating the `out_valid_o` signal would allow an adversary to leak temporary encryption data to the software interface of the AES, we exemplarily focus on analyzing this signal. More specifically, we instrument SYNFI to show whether it is possible to manipulate this signal to a specific value **(FS)**, *i.e.*, from a logical 0 to a logical 1. Here, we configure SYNFI to inject a single fault into the FSM circuit responsible for driving this signal. The verification result in Row ① in Table 2 shows that already a single fault induced into the circuit enables an adversary to tamper the handshake signal.

```
1  module aes_cipher_control_fsm (
2    output logic             out_valid_o,
3    input  logic [3:0]       rnd_ctr_q_i,
4    ...
5  );
6    assign num_rounds_regular = num_rounds_q_i - 4'd1;
7    unique case (aes_cipher_ctrl_cs)
8      ROUND: begin
9        advance = (dec_key_gen_q_i | sub_bytes_out_req_i) & key_expand_out_req_i;
10       if (advance) begin
11         // Are we doing the last regular round?
12         if (rnd_ctr_q_i == num_rounds_regular) begin
13           if (dec_key_gen_q_i) begin
14             out_valid_o = 1'b1;
15           ...
```

Listing 6: `out_valid_o` signal generation in the `aes_cipher_control_fsm` module.

The detailed verification summary reporting the fault-affected cells shows that the adversary can induce faults either *(i)* directly into the `out_valid_o` signal (Line 14 in Listing 6), the *(ii)* comparisons in the output logic (Line 12 in Listing 6), or the *(iii)* control signals (Line 13 in Listing 6) of the FSM. Hence, to comprehensively protect the handshake signal, we must consider all three attack vectors.

Multi-bit encoding. To protect critical handshake signals *(i)*, we extend the AES IP to adopt the multi-bit encoding the OpenTitan project uses in other hardware modules.

```
1  typedef enum logic [2:0] { SP2V_HIGH = 3'b011, SP2V_LOW  = 3'b100 } sp2v_e;
```

Listing 7: Encoded multi-bit signals.

In the multi-bit encoding approach shown in Listing 7, a 1-bit signal is encoded into a 3-bit signal resulting in a Hamming distance of three. Here, the first two bits represent the logical value and the third bit is the inverse of the value to encode.

To verify that the synthesis step does not weaken the security guarantees of multi-bit signals by simplifying the encoding in the optimization phase, we use the SYNFI framework

to inject faults into the encoded `out_valid_o` signal. In particular, we use SYNFI to reveal if it is possible to manipulate the encoded signal to a specific value **(FS)**, *i.e.*, from `SP2V_LOW` to `SP2V_HIGH`. The verification result in Row ② in Table 2 confirms the expected security bound of three, *i.e.*, SYNFI could not find an effective fault when inducing one or two simultaneous faults.

Multi-rail FSM. As a single fault into the output logic *(ii)* of the FSM, e.g., the comparison in Line 12 in Listing 6, is enough to tamper the out valid signal, we design and deploy a redundant multi-rail FSM scheme.

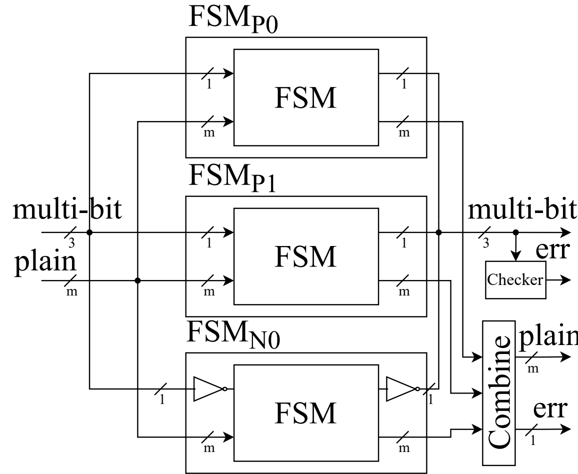


Figure 4: Multi-rail FSM approach.

The multi-rail scheme, as shown in Figure 4, instantiates the unmodified FSM in a triple modular redundancy mode. Unencoded m-bit signals are independently processed by the three state machines and an output logic is responsible for combining and checking the resulting signals. On a comparison mismatch, an alert signal is triggered. For multi-bit encoded signals (cf. Listing 7), the first two bits are processed by the positive FSM_{P0} and FSM_{P1} rail. The inverted third bit is processed by the negative FSM_{N0} rail. Combining these signals at the output again produces an encoded multi-bit signal. If a fault in one or two FSMs modifies the signal, an invalid code word is produced, which is detectable by a checker unit.

Listing 8 shows the multi-rail FSM approach integrated into the AES module. In this approach, three redundant FSMs are instantiated where the two `aes_cipher_control_fsm_p` FSMs produce a positive output and the `aes_cipher_control_fsm_n` FSM a negated output. Combined, they form a multi-bit signal with a Hamming distance of three. As the multi-rail approach requires to instantiate FSMs redundantly, the area increases from 211.15 GE for the `aes_cipher_control_fsm_p` FSM to 908.81 GE for the whole `aes_cipher_control` module including the redundant FSMs, the combination logic, as well as other countermeasures, such as the counter error logic introduced in Section 4.1.1.

```
1 assign sp_dec_key_gen_q = {dec_key_gen_q}
2 // For every bit in the Sp2V signals, one separate rail is instantiated.
3 for (genvar i = 0; i < 3; i++) begin : gen_fsm
4   if (SP2V_LOGIC_HIGH[i] == 1'b1) begin : gen_fsm_p
5     aes_cipher_control_fsm_p u_aes_cipher_control_fsm_i (
6       .rnd_ctr_q_i     ( rnd_ctr_q           ),
7       .num_rounds_q_i  ( num_rounds_q        ),
8       .dec_key_gen_q_i ( sp_dec_key_gen_q[i] ),
9       .out_valid_o     ( sp_out_valid[i]     ),
10      ...
11    );
12  end else begin: gen_fsm_n
13    aes_cipher_control_fsm_n u_aes_cipher_control_fsm_i (
14      .rnd_ctr_q_i     ( rnd_ctr_q           ),
15      .num_rounds_q_i  ( num_rounds_q        ),
16      .dec_key_gen_q_i ( sp_dec_key_gen_q[i] ),
17      .out_valid_o     ( sp_out_valid[i]     ),
18      ...
19    );
20  end
21 end
22 // Convert sparsified outputs to sp2v_e type.
23 assign out_valid_o = sp2v_e'(sp_out_valid);
```

Listing 8: Fault resistant multi-rail FSM in the `aes_cipher_control` module.

```
1 "Fault Specification":
2     "Target Circuit":
3         "inputs": ["rnd_ctr": "2"],
4         "outputs": ["out_valid_o": "SP2V_LOW"],
5         "expected fault outputs": ["out_valid_o": "SP2V_HIGH"],
6         "alerts": ["rnd_ctr_err": "0"]
```

Listing 9: Fault specification for the multi-rail FSM.

Table 3: Verification results for the AES multi-rail FSM on a 16- or 72-core* machine.

	Target	Setting	Simult. Faults	Effective [%]	Total [#]	Execution	Circuit [GE]
①	Multi-Rail FSM loose config	**FS**	1	2.46	122	8.2 s	96.5
②	Multi-Rail FSM tight config	**FS**	1	0	573	266.87 s	355.75
③	Multi-Rail FSM tight config	**FS**	2	0	170,982	1.01 h	355.75
④	*Multi-Rail FSM tight config	**FS**	3	0.02	35,222,293	38.27 h	355.75

To verify that the synthesis tool does not remove the redundant FSMs, we utilize SYNFI to analyze the resilience of the multi-rail approach against faults. In particular, we instrument the framework to reveal whether there exists a fault enabling an adversary to manipulate the `out_valid_o` signal to a specific value **(FS)**, *i.e.*, from `SP2V_LOW` to `SP2V_HIGH`, without triggering the fault countermeasure. Using the fault specification file shown in Listing 9, SYNFI automatically extracts the circuit between the round counter

register and the out valid signal, *i.e.*, the multi-rail FSM including the input, output, and error logic. By setting the counter value to 2, we force the circuit in a state where the out valid signal is set to a logical 0 in the fault-free setting. As shown in Row ① in Table 3, in this SYNFI configuration, already a single fault can be sufficient to produce an effective fault, *i.e.*, the out valid signal is set to a logical 1 and the error was not triggered. The framework reports that all of these four effective faults occur when faulting the `num_rounds` register value. Since the value of this register is used for the comparison in Line 12 in Listing 6 in all redundant FSMs, the out valid signal is set to a logical 1. Nevertheless, this verification result can only provide a limited statement about the security of the multi-rail approach as the SYNFI framework was minimally constrained. By only defining the input value of the round counter register, the SAT solver is *loosely* constrained (cf. Section 3.7) and automatically sets the `dec_key_gen_q_i` and `advance` signals in the boolean formula of the differential graph to a logical 1. Setting these variables is possible, as in the non-faulty reference circuit the `out_valid_o` always stays at `SP2V_LOW` when the round counter value is 2. Hence, as all redundant FSMs set the `out_valid_o` to the same value, the error signal is not set.

To avoid these false-positive results, we more *tightly* configure SYNFI by further defining the inputs $dec_key_gen_q_i = 0$ and $key_expand_out_req_i = 0$ in the fault specification file. In this configuration, the extracted circuit SYNFI analyzes increases from $96.5\,GE$ to $355.75\,GE$, as the tool finds more paths from the defined inputs to the outputs. Now, as expected and depicted in Row ② and ③ in Table 3, one or two simultaneous faults cannot manipulate the out valid signal. Starting with three simultaneous faults into the circuit, we observe effective faults (cf. Row ④ in Table 3). These effective faults manipulating the out valid signal are caused by inducing bit-flips into variables used by the redundant FSMs. To demonstrate the possibility of scaling SYNFI to the cloud and as the number of possible fault combinations for three simultaneous faults, *i.e.*, fault location and fault mapping, explodes, we conducted this experiment on a 72-core server. We measured a total run time of $38.27\,h$ and a maximum memory consumption of less than $8\,GB$ for injecting $35,222,292$ faults into the circuit of a size of $355.75\,GE$.

Shadow registers. As discussed in the initial experiment in Section 4.1.2, handshake signals also can be tampered by faulting *(iii)* control signals used by the FSM. To protect security-critical control signals, which are provided by the software over a register interface, the AES modules stores them in dedicated shadow registers. These registers constantly compare the two values and on a comparison mismatch caused by, for example a fault, an alert is raised forcing the AES module in a terminal state.

4.1.3 Sparsely Encoded State Machines

Finite-state machines are, as seen in Section 4.1.2, security-critical hardware elements as they are responsible for setting control signals used by the data path.

```
1 typedef enum logic [3:0] { IDLE, INIT, ROUND, ..., ERROR } aes_cipher_ctrl_e;
2 always_comb begin : aes_cipher_ctrl_fsm
3   unique case (aes_cipher_ctrl_cs)
4     IDLE: begin
5       if <condition>:
6         aes_cipher_ctrl_ns = INIT;
7       end
8     end
9     INIT: begin
10    ...
```

Listing 10: Finite-state machine with a state encoding vulnerable to faults.

```
1 typedef enum logic [5:0] {
2   IDLE       = 6'b001001, INIT        = 6'b100011, ROUND       = 6'b111101,
3   FINISH     = 6'b010000, PRNG_RESEED = 6'b100100, CLEAR_S     = 6'b111010,
4   CLEAR_KD   = 6'b001110, ERROR       = 6'b010111
5 } aes_cipher_ctrl_e;
```

Listing 11: Sparsely encoded FSM state.

Table 4: Verification results for the AES FSM state encoding on a 16-core machine.

Target	**Setting**	**Simult. Faults**	**Effective [%]**	**Total [#]**	**Execution [s]**	**Circuit [GE]**
Encoded FSM states	**FS**	3	15	29	16.09	90.5

In a state machine, the next-state logic derives the next state from the current state and a set of inputs. As seen in Listing 10, the state variable stored in the state register is typically represented as a simple enum. However, as the minimum Hamming distance between two states is 1, a single fault into the state registers would allow an adversary to hijack the control-flow of the FSM, *i.e.*, skip or enter a normally non-reachable state.

In order to mitigate this threat, we deploy the sparse FSM state encoding technique used by different OpenTitan modules into the AES. The encoding, which is shown in Listing 11, assures a minimum Hamming distance between the states of 3, increasing the resistance against faults. Additionally, we introduce a default error state, which is entered when the state value does not match the `aes_cipher_ctrl_e` enum. Now, if a fault flips bits in the state variable, with a high probability, the terminal error state is entered.

To verify that an aggressive synthesis setting does not reduce the security by altering the state encoding, we utilize SYNFI to analyze the `aes_cipher_control_fsm` FSM. In particular, we determine, how many faults are required to hijack the control-flow of the FSM by skipping a certain state and directly enter a normally not reachable state, *i.e.*, (**FS**). For this experiment, we instrument SYNFI to analyze the next-state logic and to inject faults directly into the state registers. SYNFI shows that one or two simultaneous bit-flips into the state registers triggers the alert signal, *i.e.*, the FSM enters the error state. When inducing three simultaneous faults, as shown in Table 4, the attacker is able to redirect the control-flow of the FSM.

FSM optimizations. Several synthesis tools also apply optimization passes to state machines. Yosys, for example, removes unused control signals, merges states, and recodes the FSM state variables stored in the state registers [Wol]. To analyze the impact of these optimization on the security of the sparsely encoded states, we synthesize the `aes_cipher_control_fsm` module with Yosys using an aggressive optimization strategy. Similar to the previous experiment, we configure SYNFI to skip a state and directly enter a typically non-reachable state. Our result shows that, in comparison to the verification in Table 4, now 2 instead of 3 simultaneous faults are already sufficient to skip the FSM state, *i.e.*, the FSM optimization weakens the encoding. To prevent these optimizations, Yosys can be parameterized with the `nofsm` flag. In summary, this experiment shows that synthesis optimizations configured by different stakeholders, e.g., trying to minimize the area of the design, could have fatal security implications.

4.2 Life Cycle Controller

OpenTitan can be transferred into different operational states depending on where the device is deployed, *i.e.*, at the customer or the manufacturer. The state switching mechanism is implemented in hardware in the life cycle controller module. As certain states, e.g., the return material authorization (RMA) state, enable security-sensitive features, such as

Table 5: Verification results for entering the RMA state on a 16-core machine.

	Target	Setting	Simult. Faults	Effective [%]	Total [#]	Execution [s]	Circuit [GE]
①	Token comparison in `TokenHashSt`	**FS**	1	0	313	32.56	215
②	Token comparison in `TokenCheck0St`	**FS**	1	0	349	35.87	248.25
③	Token comparison in `TokenCheck1St`	**FS**	1	0	310	28.26	204.75
④	Skip token check states	**FS**	7	100	7	17.44	214.5

access to the debug port, the life cycle controller is hardened against fault attacks.
Results. SYNFI verified that the analyzed fault-hardened primitives of the life cycle controller provide adequate protection. Specifically, we could confirm that the countermeasures prevent the exploitation of single or double faults for all critical attack vectors.

4.2.1 Entering the RMA State

The core mechanism of the life cycle controller IP is an FSM responsible for determining the life cycle state of OpenTitan. To hinder an adversary from entering the security-sensitive RMA state, which is used to debug the RoT chip when returned to the manufacturer, this state transition is only permitted when possessing a 128-bit unlock token. Internally, the state machine checks the validity of the token in three different FSM states. This redundancy-based mechanism and the state encoding technique guaranteeing a minimum Hamming distance of 7 between the state symbols form the fault protection strategy of the controller. Based on this description of the fault-hardening mechanisms, we identified two major attack vectors for a fault attacker: *(i)* glitch the token comparisons three times or *(ii)* hijack the execution of the FSM by glitching the state symbols.
Glitching the comparisons. Glitching the three token checks requires a strong adversary capable of injecting three faults in three clock cycles. We utilize SYNFI to test whether these three comparisons are susceptible to a single fault each. For this verification, we configure SYNFI to analyze if it is possible to induce a fault into the next-state logic of the FSM changing the next valid state **(FS)**. In the fault specification file, we instrument the tool to exhaustively induce a single fault into all gates of the next-state logic for each of the three fault experiments. Row ①-③ in Table 5 shows that SYNFI could not find a single fault allowing the attacker to enter the next state (from `TokenHashSt` to `TokenCheck0St`, $\cdots$) without possessing the required token.
Skip the token check states. In order to verify that the synthesis step does not weaken the 16-bit FSM state encoding, we utilize SYNFI to check whether it is possible to induce faults into the state register allowing the attacker to directly enter the RMA state **(FS)**. Here, we configure SYNFI to analyze the FSM and to inject 1 to 7 simultaneous faults into the state register. As shown in Row ④ in Table 5, at least 7 simultaneous faults are required to enter the target state. This matches the security expectation, *i.e.*, a Hamming distance of 7 for the state symbol encoding.

4.2.2 Flash Erase Mechanism

Before entering RMA, the life cycle controller erases the flash to hinder an adversary from accessing previously created data. Although entering RMA only is possible when knowing a secret, device dependent token, this hardware-backed flash erasing mechanism is meant to be a second line of defense. Internally, the flash erasing command is directly triggered in the FSM of the life cycle controller. To ensure that the flash was erased before entering

Table 6: Verification results for glitching the locking mechanism performed on a 16- or 72-core* machine.

Target	Setting	Simult. Faults	Effective [%]	Total [#]	Execution	Circuit [GE]
① Prevent counter incr.	**FS**	1	17.81	853	261.76 s	1170
② Reset counter value*	**FS**	3	0	1,000,000	7 h	1170
③ Skip `CntProgSt`	**FS**	7	100	7	17.53 s	228.75

RMA, the acknowledgment sent by the flash controller is checked three times in the FSM. If one of the acknowledgements was not received, e.g., due to a fault, the FSM remains in the current state.

Glitching the Encoded Flash Handshake Signal. An attacker with access to a valid RMA token aiming to bypass the flash erasing mechanisms needs to suppress the flash erasing command as well as the acknowledgement signal or the corresponding check three times. However, as the initiate and acknowledgement signal is encoded with a Hamming distance of 4, the adversary theoretically needs to flip 4-bits four times. To confirm this behavior, *i.e.*, the synthesis tool did not tamper the encoding, we exemplary analyze the resilience of the flash erase initiate signal. In particular, we instrument SYNFI to reveal whether it is possible to induce faults manipulating this signal to a specific value **(FS)**, *i.e.*, from an encoded $On = 4'b1001$ to an encoded $Off = 4'b0110$. This analysis showed that already two simultaneous faults injected into the encoded signal allow an adversary to hijack the initiate signal. Since bit 0 and 3 as well as 1 and 2 in the encoding are always the same, the synthesis tool decided to merge these signals and only instantiate two instead of four registers driving the bits of the signal. Hence, the security of the encoding is reduced to a Hamming distance of 2.

In order to prevent that the four registers instantiated in the HDL code are merged in the synthesized netlist, we augmented the design flow with the `set_dont_touch` parameter. Now, as shown in Row ④ in Table 5, the encoding works as expected and an attacker needs at least four simultaneous faults to tamper the encoded signal.

4.2.3 Locking Mechanism

OpenTitan limits the number of state transitions and transition attempts to 24. Once this number is reached, the life cycle controller rejects further attempts, effectively locking the device into its current state. In the life cycle hardware IP block, the counter increment is conducted in the `lc_ctrl_state_transition` module and the counter value is programmed into the one time programmable (OTP) memory of OpenTitan in the `lc_ctrl_fsm` FSM. Hence, an adversary aiming to increase the number of state transitions attempts either needs to fault the counter increment *(i)* or the programming *(ii)* of the value into the OTP.

Skip the Counter Increment. Inside the `lc_ctrl_state_transition` module, an FSM is responsible for updating the counter value. As illustrated in Listing 12, this FSM uses a strong state encoding technique to mitigate fault attacks. Each state of type `lc_cnt_e` consists of 384-bit with a Hamming distance of 269 between `LcCnt0` and `LcCnt24` and a Hamming of 3 between `LcCnt23` and `LcCnt24`. To verify that the synthesis flow does not weaken the encoding, we utilize SYNFI to verify that a fault cannot manipulate the `next_lc_cnt_o` variable to a specific value **(FS)**. More concretely, we aim to bypass the counter increment from $LcCnt23$ to $LcCnt24$. In this scenario, the attacker aims to avoid that the counter increments to the final $LcCnt24$ value and locks the life cycle controller.

As shown in Row ① in Table 6, already a single fault allows the adversary to avoid that the counter is incremented. Supported by the generated analysis results, we were able to track back the single point of failure responsible for enabling an attacker flipping the

```
1 module lc_ctrl_state_transition (
2   input  lc_cnt_e           lc_cnt_i,
3   output lc_cnt_e           next_lc_cnt_o,
4   ...
5 );
6   unique case (lc_cnt_i)
7     LcCnt0:   next_lc_cnt_o = LcCnt1;
8     LcCnt1:   next_lc_cnt_o = LcCnt2;
9     ...
10    LcCnt23:  next_lc_cnt_o = LcCnt24;
11 endcase
```

Listing 12: Life cycle controller counter increment FSM.

three bits, *i.e.*, the Hamming distance between `LcCnt23` and `LcCnt24`, with a single fault. Since the three gates driving the three targeted bits of the `next_lc_cnt_o` output port are driven by a single gate, attacking this gate or drivers of this gate allow an adversary to manipulate the output counter value. However, as the counter increment is only prevented once, an attacker only could initiate one additional state transition, making this attack impractical in reality.

Since resetting the counter value to `LcCnt0` enables the attacker to initiate more additional state transitions, the encoding is also stronger, *i.e.*, a Hamming distance of 269 between `LcCnt23` and `LcCnt0`. To ensure that this strong encoding between these two counter values is correct after the synthesis, we test with SYNFI whether it is possible to switch to the specific `LcCnt0` value from `LcCnt23` with faults **(FS)**. Similar to the previous experiment, we configure the framework to exhaustively inject 1, 2, and 3 simultaneous faults into the circuit responsible for incrementing the counter value. Since the possible fault combinations explode, we limited the number of injected faults to 1 M and performed the experiment on a 72-core server in the cloud. Within this fault threat model, SYNFI could not find a single, effective fault, as shown in Row ② in Table 6.

Prevent the Programming of the Counter. The programming of the counter value into OTP is initiated in the `CntProgSt` state in the life cycle controller FSM. We utilize SYNFI to validate that an adversary cannot bypass this state and directly switch to the next state **(FS)**. As shown in Row ③ in Table 6, at least 7 faults, *i.e.*, the Hamming distance the encoding guarantees, are required to skip the state.

4.2.4 Life Cycle Escalation Signal

When the life cycle controller detects an ongoing attack, the 4-bit encoded `lc_escalate_en` signal is triggered. This signal is consumed by other hardware modules, e.g., the AES IP, and transfers them into an non-escapable error state. To validate that the optimization passes in the synthesis does not weaken the encoding of the signal, we inject faults into the escalation signal driven in the `lc_ctrl_signal_decode` module. In the fault model used by SYNFI, we set the input value to $lc_escalate_en = On = 4'b1001$, the expected output value to $lc_escalate_en_o = On = 4'b1001$, and the fault output to $lc_escalate_en_o = Off = 4'b0110$ **(FS)**. Without constraining the synthesis flow with the `set_dont_touch` parameter (cf. Section 4.2.2), the security of the encoding is reduced to a Hamming distance of 2, as Synopsys removes the redundant flip-flops. When applying this constraint to the `lc_escalate_en` flip-flop, at least four simultaneous faults are required to suppress the escalation signal.

Table 7: Verification results for the Ibex processor on a 16-core machine.

Target	Setting	Simult. Faults	Effective [%]	Total	Execution	Circuit [GE]
① Glitch the PC	**FE**	1	78.1	557	185.07 s	488
② Glitch the PC	**FS**	2	0.02	309,500	0.72 h	488
③ Lockstep mode	**FS**	1	6.31	111	10.22 s	165.34

4.2.5 Privilege Escalation in the PROD State

When OpenTitan is shipped to the customer, the device is put into the PROD state. In this state, certain features are activated, such as the CPU, and security-critical features, such as debug functionalities, are disabled. Instead of directly hijacking the life cycle state of OpenTitan, an adversary also could aim to switch on such features in the PROD state. All features are activated in the `lc_ctrl_signal_decode` module by setting the corresponding signal from `Off` to `On`. This signal then is transmitted to the corresponding hardware module responsible for activating or deactivating the feature. Similar to the escalation signal described in Section 4.2.4, the OpenTitan project uses a 4-bit encoding technique with a Hamming distance of 4 between `Off` and `On`. For the fault verification of the encoded signal, we configure the input to $lc_hw_debug_en = Off = 4'b0110$, the expected output value to $lc_hw_debug_en_o = Off = 4'b0110$, and the fault output to $lc_hw_debug_en_o = On = 4'b1001$ **(FS)**. Similar to the previous experiments, the `set_dont_touch` constraint needs to be applied to the registers responsible for driving the `lc_hw_debug_en` signal to maintain a Hamming distance of 4. Then, at least four simultaneous faults are required to enable the debug mode. While a transient fault only can active the debug functionality for a brief moment, a permanent stuck-at fault could allow an adversary to enable this feature permanently.

4.3 Ibex

The RISC-V Ibex processor is the core element of the OpenTitan chip. In this section, we utilize SYNFI to analyze the behavior of the CPU when injecting faults. To demonstrate the ability of SYNFI to handle different netlists, we, contrary to the previous verification setups, analyze the netlist synthesized with the open-source Yosys synthesis tool and the open Nangate45 cell library.

Results. Our analysis showed that the error logic of the Ibex lockstep mode is capable of detecting a fault into the program counter.

4.3.1 Glitching the Program Counter

A fault into the program counter (PC) allows an attacker to arbitrarily redirect the control-flow of the program executed on the processor [TSW16]. We utilize SYNFI to ① analyze whether the instruction fetch stage of the Ibex is generally susceptible to a fault arbitrarily changing the PC **(FE)**. Row ① in Table 7 shows that a single fault already is sufficient to manipulate the PC and to redirect the control-flow. Although targeting a `NOP` slide does not require an adversary to accurately manipulate the PC, randomly glitching the PC makes it hard for the attacker to exploit the induced fault. Therefore, we analyze ② if it is possible to change the program counter to a specific PC, *i.e.*, from the boot address to 32'h40400, using a fault **(FS)**. The analysis of SYNFI in Row ② in Table 7 shows that, with two simultaneously induced faults, glitching the PC to a specific value is hard. More specifically, for this fault specification, SYNFI shows that only 62 (0.02 %) out of $309,500$ injected faults manipulate the program counter to the specified value.

4.3.2 Lockstep Mode

To protect the execution of software on Ibex from faults, OpenTitan instantiates the CPU redundantly in a dual-core lockstep mode. In this approach, the input used for the main core is delayed, provided to the redundant core, and the delayed output is compared to the output of the main core. On a mismatch, a hardware monitor raises an alert. Similar to the verification setup in Section 4.3.1, we consider an adversary aiming to redirect the control-flow by glitching the program counter. For the verification, we assume that the attacker already managed to flip a bit in the instruction address generated by the main core but not in the shadow core. As the error detection logic should raise an error due to the mismatch, a fault attacker needs to additionally inject a fault into this error detection circuit. SYNFI reveals that *(i)* the error detection logic actually raises an error, *i.e.*, the synthesis tool did not remove the redundant logic, and that *(ii)* one fault could enable an attacker to suppress the error flag **(FS)**, as shown in Row ③ in Table 7.

4.4 Generic Primitives

The OpenTitan project provides a set of generic hardware building blocks which are reused in several hardware modules. In this section, we analyze the fault protected generic primitives, *i.e.*, the counter and the LFSR, using SYNFI.
Results. Our analysis with SYNFI confirms that the inspected primitives provide the expected security, *i.e.*, a single fault into the protected counter or the LFSR triggers the alert signal of the countermeasures.

4.4.1 Counter

The `prim_count` module provides a fault protected generic counter primitive which is used by different modules. This module offers a flexible parameterization interface allowing the hardware designer to define the mode, the start value, and the bit width of the counter. In order to mitigate faults manipulating the counter value, the `prim_count` module implements the double count or cross count protection mode. While in the double count mode two redundant counters are compared, in the cross count mode the values of the up counting counter and the down counting counter are added and compared to a constant. On a comparison mismatch, a fault is detected and an error is triggered.

To ensure that the synthesis does not remove the redundant counter, we use the SYNFI framework to test the resilience of the module against faults. In particular, we check whether the countermeasure can detect a fault arbitrarily changing the output of the counter value **(FD)**. For this, we configure SYNFI to inject faults into the counter logic, the counter registers, and the comparison logic. With this description, SYNFI automatically extracts the target circuit (29.75 GE) from the overall counter circuit (32.75 GE).

When injecting one fault into the netlist, SYNFI finds two effective faults manipulating the output counter value without triggering the error logic. These effective faults occur when faulting the counter increment signal or the counter clear signal, which are used by both counter instances. Since this behavior is documented in the description of the module, we further analyze the effect of two faults into the counter. In this setting, our tool shows

Table 8: Verification results for the `prim_double_lfsr` and `prim_count` modules.

	Target	Setting	Simult. Faults	Effective [%]	Total	Execution [s]	Circuit [GE]
①	`prim_count`	**FD**	2	10.82	1552	37.46	29.75
②	`prim_double_lfsr`	**FD**	2	0.05	7827	74.44	116.75
③	`prim_double_lfsr`	**FD**	3	0.09	340,692	2114.69	116.75

in Row ① in Table 8, that 10.82 % of all injected faults are effective, *i.e.*, manipulate the output counter value to an arbitrary value but do not trigger the error signal.

4.4.2 Double LFSR

As the linear-feedback shift registers (LSFRs) are used in OpenTitan as the primary source of randomness, they require a strong protection against fault attacks. The `prim_double_lfsr` module, which is used by several hardware IP blocks in the project, instantiates an LFSR twice and triggers an exception if the comparison of the two generated values mismatches.

In order to verify that a potentially aggressive synthesis setup does not remove the redundancy used as a fault protection, we use SYNFI to induce faults into the netlist and observe the behavior of the circuit. Here, we are interested if the error detection logic is capable of detecting a fault arbitrarily manipulating the output of the LFSR **(FD)**. SYNFI confirms that a fault into the circuit manipulating the LFSR value is detected by the error logic. When inducing two simultaneous faults into the netlist, SYNFI finds, as shown in Row ② in Table 8, 4 effective faults either suppressing the error signal and changing the output LSFR value or manipulating the LFSR value in both LFSR modules. Increasing the number of simultaneous faults to three increases the number of faults injected into the circuit of a size of $116.75\,GE$ to $340,692$, which takes $35\,min$ on a 16-core setup. Based on these results, forging the LFSR output to an attacker controllable value with three or less simultaneous faults is hard to achieve.

5 Related Work

Fault injection verification frameworks can be categorized into simulation- or verification-based approaches operating either on the RTL model or on the gate-level netlist. As indicated in Section 1, frameworks [Gei20, JAR+95] working on the HDL description of a module only can provide security assumptions for this level of abstraction. In particular, the transformation of the RTL model into the gate-level netlist, *i.e.*, the synthesis, can be responsible for inducing flaws into redundancy-based fault countermeasures by applying optimization passes. To also detect flaws potentially introduced in this design phase, various frameworks conduct their fault experiments at the netlist level [BGE+17, AWMN20, RBSS+21, BDN08, SKK13]. The disadvantage of simulation-based frameworks [BDN08, AWMN20, SKK13] is that they require an input stimuli covering all inputs of the circuit. Verification-based frameworks, such as SYNFI, FIVER [RBSS+21], and AutoFault [BGE+17], can achieve higher fault coverage as a SAT solver is responsible for probing all the undefined inputs. Similar to SYNFI, AutoFault and FIVER transform the gate-level netlist into a different representation and extract the equation of the circuit. FIVER first transforms the circuit into a DAG and then converts this graph into a binary decision diagram to perform the symbolic fault injection. As fault attacks originally focused on breaking cryptographic primitives, most fault injection frameworks [BGE+17, AWMN20, BDN08], including FIVER, concentrate on analyzing such schemes. However, when using these frameworks to analyze more generic circuits, such as a silicon design of a root-of-trust element including a broad range of fault countermeasures, there are some limitations to overcome. First, some tools only provide support for a subset of VHDL descriptions [BGE+17] and others limit the number of supported gates [RBSS+21] to a small set. Especially for industry-grade designs using long-established digital design flows, this constraint is severe as it is unlikely to adapt these hardware design flows. SYNFI overcomes these limitations by automatically processing and including arbitrary standard cell libraries into the framework and by translating the Verilog netlist into a unified model, *i.e.*, a directed multigraph. This approach allows the framework to also

support submodules when the boolean formula is provided. Second, FIVER requires that the given netlist does not include any cycles, *i.e.*, the hardware designer needs to manually unroll the design before the evaluation. As described in Section 3.3, SYNFI is able to also handle such designs and automatically unfolds cycles found in the graph. To overcome computational limitations for larger circuits, the architecture of SYNFI makes heavy usage of multiprocessing, allowing the distribution of large workloads into the cloud. Finally, and in contrast to other frameworks [BGE+17, Gei20, JAR+95], we release an open-source version of SYNFI to encourage the verification of other security-critical designs.

Similar to related work [RBSS+21], SYNFI can also be used to analyze the resilience of cryptographic primitives against fault attacks. For example, when analyzing a round of the LED block cipher [GPPR11] protected by a detection-based countermeasure, the SYNFI user needs to provide a plaintext-ciphertext pair in the fault configuration file. Then, depending on the configuration, SYNFI can detect *(i)* whether it is possible to induce a fault with any effect on the ciphertext without triggering the countermeasure or *(ii)* whether it is possible to flip certain bits in the ciphertext without triggering the countermeasure.

6 Limitations and Future Work

This section summarizes current limitations of SYNFI and highlights potential future work.

Fault Specification. In the current prototype implementation of SYNFI, the user needs to manually specify input and expected output values in the fault model configuration. A possible future work could be to automate this process by parsing these values from traces generated by the simulation tools. This parser fetches the values of the circuit of interest for a specific amount of clock cycles and automatically writes these values into a separate fault model for each clock cycle. As SYNFI is already capable of successively analyzing multiple fault models (cf. Appendix A.1), no additional changes in the existing framework would be required.

To assist the security engineer to specify the target circuit in the fault specification file, the SYNFI repository[2] contains an experimental feature automatically creating this file. When using this feature, the SYNFI user directly can specify the input and output nodes and their values of the target circuit in the HDL code using code annotation. The experimental tool then extracts this annotated information from the netlist and automatically describes the target circuit in the fault specification file, *i.e.*, the tool defines the inputs and outputs and the state of the circuit.

Preprocessing. SYNFI automatically extracts a time-independent mathematical model of the circuit to analyze in the preprocessing phase. This time-independent model is created by replacing registers used in pipeline stages with pass-through elements and by removing cycles introduced by sequential logic. Hence, SYNFI is capable of analyzing the effect of a fault in multiple clock cycles. In addition, by automatically processing register stages and sequential loops, the framework can handle designs that the designer did not manually unroll. However, when aiming to analyze multiple loop iterations, e.g., multiple rounds in an iterative AES implementation, SYNFI needs to be configured for each round individually. Nonetheless, as SYNFI allows using multiple fault configurations in a single fault model specification file executed in one verification run, this is only a minor limitation. Nevertheless, a possible extension of the framework could automatically unroll the circuit instead of removing the loop.

[2] https://github.com/lowRISC/synfi

Fault Effects & Layout. SYNFI and related frameworks [BGE+17, AWMN20, BDN08] model a fault at the logical and not at the electrical level. Consequently, these frameworks cannot analyze transient faults occurring within a clock cycle and they also cannot consider the propagation delay between gates. Additionally, these tools, including SYNFI, operate on the gate-level netlist after the synthesis step and not on the layout after place and route. As some backend tools provide the possibility to simplify and optimize the netlist before the actual place and route step, SYNFI needs to be reapplied to this netlist to confirm the evaluation results. A future work could extend SYNFI to operate on the layout to also take the position of the gates into account for the analysis.

Performance. One of the main performance impact factors is the extraction and preprocessing phase (cf. Section 3.3). In this phase, SYNFI extracts *(i)* the target graph by finding all paths from the input and output nodes specified in the fault specification and handles *(ii)* registers used in iterative designs by finding cycles including a register. These operations on the graph could be improved by switching to a faster Python graph library or by porting SYNFI to C or C++.

Another performance limitation is the number of fault combinations, *i.e.*, fault locations and fault effects. For an exhaustive fault analysis over all gates and multiple simultaneous injected faults, the number of fault combinations explodes. As SYNFI, for each fault combination, needs to create the differential graph, convert this graph into a boolean formula, and uses a SAT solver to evaluate the effectiveness of the faults, the number of fault combinations primarily affects the runtime. To improve this evaluation performance, the optimizations proposed by FIVER [RBSS+21] could be integrated. Here, FIVER uses a fault propagation path analysis and a clustering technique to minimize the computational overhead.

7 Conclusion

In this paper, we presented SYNFI, a pre-silicon framework capable of inducing faults and analyzing their effects on the gate-level netlist. The framework enables hardware designers and security engineers to analyze the resilience of designs against fault attacks. As SYNFI conducts the security assessment directly on the unmodified netlist, the framework assures that *(i)* the same netlist is used for the evaluation as for the next steps in the digital hardware design flow with the final tape-out step and that *(ii)* potential security weaknesses still can be fixed before the chip gets manufactured. For the evaluation, SYNFI extracts the circuit of interest and injects fault into this circuit according to the fault model. To evaluate the effect of induced faults, the framework constructs a differential graph, transforms this graph into a mathematical model, and uses a SAT solver to study the behavior of the circuit when affected by faults. SYNFI is capable of *(i)* revealing whether faults affect the input-output relation of a circuit and its countermeasures and *(ii)* showing whether it is possible to enter a security-critical state using a fault without triggering the countermeasures. We utilized the framework to assess the security of several hardware modules of OpenTitan, a secure RoT chip. Our evaluation results presented in Section 4 showed that the unprotected AES module is highly susceptible to single faults, our proposed fault-hardening techniques increased the security, and that the other protected hardware blocks provide a strong resilience against fault attacks.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their valuable feedback. Furthermore, we would like to thank Vedad Hadzic, Bettina Könighofer, and

Roderick Bloem from Graz University of Technology for the initial discussion and Alphan Ulusoy from Google for the help with the implementation.

A Appendix

A.1 Round Counter Fault Model File

Listing 13 shows the fault specification used for the verification of the cipher control round counter FSM discussed in Section 4.1.1. The `simultaneous_faults` parameter, which also can be overwritten by a command line argument, defines the number of simultaneous faults injected into the extracted circuit. To specify the extracted target circuit, the input and output elements need to be defined using the `stages` parameter. These elements can be any gate, register, or input and output port of the circuit. For the example in Listing 13, the circuit between the `rnd_ctr` register with the input `Q` and the output port `D` is defined as the circuit of interest. The SYNFI tool then uses this information to automatically extract the target circuit by finding all paths between the defined input and output element. This circuit can consist of combinational and sequential logic. If multiple stages are provided, SYNFI automatically connects them. To constrain the SAT solver, the user needs to provide input values with the `input_values` parameter. In order to allow the output layer to evaluate the effect of a fault, the fault model also provides information about the expected, expected fault output value, and the alert value. The `node_fault_mapping` parameter defines the mapping function of a target gate. During the fault injection process, the boolean function of the target gate is replaced according to this mapping. The target gate can be defined using the `fault_locations` entry. If the fault evaluator does not have an intuition about the critical gates which need to be analyzed, the SYNFI tool is also capable of exhaustively targeting all gates in the extracted circuit.

```
1  {
2      "fimodels": {
3          "aes_cipher_control_fsm_rnd_cntr_target_value": {
4              "simultaneous_faults": 1,
5              "stages": {
6                  "stage_cntr": {
7                      "inputs": [
8                          "rnd_ctr_q_i"
9                      ],
10                     "outputs": [
11                         "rnd_ctr_d_o"
12                     ],
13                     "type": "inout"
14                 }
15             },
16             "input_values": {
17                 "rnd_ctr_q_i": {
18                     "i": {
19                         "0": 1, "1": 0, "2": 0, "3": 0
20                     }
21                 }
22             },
23             "output_values": {
24                 "rnd_ctr_d_o": {
25                     "o": {
26                         "0": 0, "1": 1, "2": 0, "3": 0
```

```
27                      }
28                  }
29              },
30              "output_fault_values": {
31                  "rnd_ctr_d_o": {
32                      "o": {
33                          "0": 0, "1": 0, "2": 1, "3": 0
34                      }
35                  }
36              },
37              "alert_values": { },
38              "node_fault_mapping": {
39                  "NAND": [
40                      "AND"
41                  ]
42              },
43              "fault_locations": {
44                  "Gate_189": ["stage_cntr"]
45              }
46          }
47      }
48  }
```

Listing 13: Fault specification file for the `aes_cipher_control_fsm` round counter experiment.

References

[App21] Apple. Mac models with the Apple T2 Security Chip, 2021. https://support.apple.com/en-us/HT208862.

[AS18] Gilles Audemard and Laurent Simon. On the glucose SAT solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001, 2018.

[AWMN20] Victor Arribas, Felix Wegener, Amir Moradi, and Svetla Nikova. Cryptographic fault diagnosis using VerFI. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 229–240. IEEE, 2020.

[BDN08] Alberto Bosio and Giorgio Di Natale. LIFTING: A flexible open-source fault simulator. In *2008 17th Asian Test Symposium*, pages 35–40. IEEE, 2008.

[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.

[BGE+17] Jan Burchard, Manl Gay, Ange-Salomé Messeng Ekossono, Jan Horáček, Bernd Becker, Tobias Schubert, Martin Kreuzer, and Ilia Polian. Autofault: towards automatic construction of algebraic fault attacks. In *2017 Workshop*

on Fault Diagnosis and Tolerance in Cryptography (FDTC), pages 65–72. IEEE, 2017.

[Bie17] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[Bir04] Alex Biryukov. The boomerang attack on 5 and 6-round reduced AES. In *International Conference on Advanced Encryption Standard*, pages 11–15. Springer, 2004.

[Bro21] David Brown. Confidential computing: an AWS perspective, 2021. https://aws.amazon.com/blogs/security/confidential-computing-an-aws-perspective/.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual international cryptology conference*, pages 513–525. Springer, 1997.

[DEK+18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: exploiting ineffective fault inductions on symmetric cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 547–572, 2018.

[DM12] Eric DeBusschere and Mike McCambridge. Modern game console exploitation. *CSc 566 2012: Research Presentations*, pages 466–566, 2012.

[DR99] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. 1999.

[ELG20] Mahmoud A Elmohr, Haohao Liao, and Catherine H Gebotys. EM fault injection on ARM and RISC-V. In *2020 21st International Symposium on Quality Electronic Design (ISQED)*, pages 206–212. IEEE, 2020.

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[Gei20] Johannes Geier. Fast RTL-based fault injection framework for RISC-V cores. 2020.

[Goo22] Google. Titan C - the nucleus of trust, 2022. https://showcase.withgoogle.com/titan-c/.

[GPPR11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The LED block cipher. In *International workshop on cryptographic hardware and embedded systems*, pages 326–341. Springer, 2011.

[Hér] Olivier Hériveaux. Defeating a secure element with multiple laser fault injections.

[HSSC08] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[IMM18] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

[JAR+95] Eric Jenn, Jean Arlat, Marcus Rimen, Joakim Ohlsson, and Johan Karlsson. Fault injection into VHDL models: the MEFISTO tool. In *Predictably Dependable Computing Systems*, pages 329–346. Springer, 1995.

[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.

[KSV13] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, 2013.

[L+18] Jia Hui Liang et al. MapleSAT: Combining machine learning and deduction in SAT solvers. *Retrieved Junary*, 8:2021, 2018.

[Li20] Abner Li. Google Tensor explained: Why the Pixel 6 has a custom chip, specs, and what it does. Oct 2020. https://9to5google.com/2021/10/20/google-tensor-chip/.

[LM21] Mark Liffiton and Jordyn Maglalang. MiniCARD, 2021. https://github.com/liffiton/minicard.

[MOG+20] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482. IEEE, 2020.

[NT19] Pascal Nasahl and Niek Timmers. Attacking AUTOSAR using software and hardware attacks. *escar USA*, 2019.

[O'F20] Colin O'Flynn. BAM BAM!! on reliability of EMFI for in-situ automotive ECU attacks. *Embedded Security in Cars Europe*, 2020, 2020.

[PQ03] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In *International workshop on cryptographic hardware and embedded systems*, pages 77–88. Springer, 2003.

[QWLQ19] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2019.

[RBSG21] Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. Revisiting fault adversary models-hardware faults in theory and practice. *Cryptology ePrint Archive*, 2021.

[RBSS+21] Jan Richter-Brockmann, Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. FIVER–robust verification of countermeasures against fault injections. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 447–473, 2021.

[RLMI21] Thomas Roche, Victor Lomné, Camille Mutschler, and Laurent Imbert. A side journey to Titan. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 231–248, 2021.

[RU96] Teresa Riesgo and Javier Uceda. A fault model for VHDL descriptions at the register transfer level. In *Proceedings EURO-DAC'96. European Design Automation Conference with EURO-VHDL'96 and Exhibition*, pages 462–467. IEEE, 1996.

[SD15] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat*, 15:71, 2015.

[SKK13] Aleksandar Simevski, Rolf Kraemer, and Milos Krstic. Automated integration of fault injection into the ASIC design flow. In *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 255–260. IEEE, 2013.

[SWUH21] Marc Schink, Alexander Wagner, Florian Unterstein, and Johann Heyszl. Security and trust in open source security tokens. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 176–201, 2021.

[TM17] Niek Timmers and Cristofaro Mune. Escalating privileges in linux using voltage fault injection. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–8. IEEE, 2017.

[Tse83] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. {CLKSCREW}: exposing the perils of security-oblivious energy management. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1057–1074, 2017.

[TSW16] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. IEEE, 2016.

[VTM+17] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adele Morisset, and Sebastien Ermeneux. Laser-induced fault injection on smartphone bypassing the secure boot. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 41–48. IEEE, 2017.

[VWWM11] Jasper GJ Van Woudenberg, Marc F Witteman, and Federico Menarini. Practical optical fault injection on secure microcontrollers. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 91–99. IEEE, 2011.

[Wol] C. Wolf. Yosys manual. https://github.com/YosysHQ/yosys-manual-build/releases/download/manual/manual.pdf.

[YJ00] Sung-Ming Yen and Marc Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on computers*, 49(9):967–970, 2000.