μArchiFI: Formal modeling and verification strategies for microarchitetural fault injections

Simon Tollec, Mihail Asavoae, Damien Courousse, Karine Heydemann, Mathieu Jan

μARCHIFI: Formal Modeling and Verification Strategies for Microarchitectural Fault Injections

Simon Tollec*, Mihail Asavoae*, Damien Couroussé§, Karine Heydemann¶‡ and Mathieu Jan*
*Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France – firstname.lastname@cea.fr
§Univ. Grenoble Alpes, CEA, List, F-38000, Grenoble, France – firstname.lastname@cea.fr
¶Thales DIS, France – firstname.lastname@thalesgroup.com
‡Sorbonne Univ., CNRS, LIP6, F-75005, Paris, France

***Abstract*—This paper introduces μARCHIFI, an open-source tool dedicated to the formal modeling and verification of microarchitecture-level fault injections and their effects on complex hardware/software systems. First, we address the problem of the system modeling, and our implementation is integrated into the Yosys toolchain. Second, we introduce verification strategies to evaluate the fault effects for software-level security. We demonstrated the practical use of μARCHIFI on RISC-V use cases using state-of-the-art model-checking tools for hardware verification.**

***Index Terms*—Faulty transition system ; Bounded model checking ; SW/HW co-verification ; Security**

I. INTRODUCTION

Context. Fault Injection (FI) attacks aim at applying abnormal execution conditions to an embedded system, such as high temperature or electromagnetic radiation. These disturbances induce computational errors in the system, leading to undesired behaviors. From a security point of view, these FI attacks can create vulnerabilities, such as the ability of an attacker to retrieve sensitive data or to acquire execution privileges on the platform. Consequently, there is a growing desire to study fault injections and better understand their effects to analyze the system's security or develop countermeasures.

Different abstraction layers are involved in describing faults in a system [1]. Faults initially appear in the circuit, and representing faults at this level permits to describe their initial effect, e.g., bit-flip, bit-reset. The consequences of the fault then propagate in the microarchitecture, can be captured by sequential logic, and induce a different behavior at the software level. An analysis at the hardware level, e.g., [2]–[4], can show that a module is functionally incorrect due to the perturbation induced by fault injections. Such approaches are sufficient for the robustness analysis of standalone components such as cryptographic IPs, but usually, the exploitation of a fault injection, in an attack, involves software. On the other side, a pure software analysis, e.g., [5]–[7], struggles to model many subtle behavioral effects induced by fault injection [8].

Recent research motivates the need to consider both the hardware and the software in the same analysis [8]–[10]. Laurent et al. show how faults on the forwarding mechanism permit to retrieve a previously computed value and reintroduces it in one of the processor pipeline stages [11]. Tollec et al. show how faults on the prefetch buffer can result in various software-level consequences [10], such as: immediate replay of instructions that are alive in the prefetch buffer; execution of instructions in incorrect order; and corruption of the next branch target. However, such effects, induced by faults in the processor microarchitecture, can only be leveraged in an attack by specific software conditions, in particular the sequence of program instructions executed, such that the fault effects propagate until the attack target is reached.

Problem statement. There is a need for modeling and analysis methods to better understand fault effects while considering the software and the hardware together. Such techniques can help to highlight microarchitectural implementation details impacting the system's security. We need to automatically build a model that encompasses both hardware and software implementation details, and fault effects. Such a model needs to be tractable by verification tools in order to leverage automated verification techniques.

Proposal and contributions. We propose a complete workflow for the formal analysis of a full system composed of hardware and software components under fault injection. Faults are modeled at the microarchitectural level to accurately analyze the impact of a fault injection at the hardware level and their effects at the software level. We leverage bounded model checking in order to reason about the impact of fault injections on the system and their possible exploitation by an attacker. This work is a follow-up of [10], and we bring improvements in two directions. First, we formalize the model-checking problem with a transition system including the attacker model. We describe its implementation in an end-to-end formal analysis workflow, named μARCHIFI, based on Yosys [12] and a third-party model checker. μARCHIFI generates the system's formal model from the RTL implementation of the hardware, an input binary program, and the attacker model to analyze the system's robustness under fault injection. The attacker model supports various fault-injection models. Second, we discuss practical strategies to improve the efficiency of the workflow, leveraging well-known optimization techniques from formal methods. We illustrate the use of μARCHIFI on several case studies and evaluate the impact of the proposed strategies. μARCHIFI is open-source and will be publicly available on the GitHub of Yosys[1].

[1]Currently available on https://zenodo.org/record/7958412

Paper outline. Section II introduces microarchitectural fault injection models and hardware transition systems. Section III describes the faulty transition system we introduce to analyze microarchitectural fault consequences at the software level. Section IV introduces our verification problem. The formal model is then automatically generated by the tool detailed in Section V. Section VI evaluates our approach in three case studies. Section VII discusses our contributions wrt. related work, and Section VIII concludes.

II. Background

This section introduces microarchitectural fault models, provides definitions on hardware transition systems modeling, and describes the Yosys framework that can translate a hardware description to formal models.

A. Microarchitectural Faults

Fault Injection (FI) attacks are a powerful threat against embedded systems that cover various physical injection means like clock glitches, electromagnetic pulses, or laser fault injections. In [13], Brockmann et al. propose a unified fault injection model that describes fault effects in the microarchitecture. The authors represent the synchronous digital circuit as a directed graph (V, E) composed of vertices V and edges E. Vertices V represent logic gates, state-holding elements, inputs, and outputs in the circuit, while edges E represent circuit wires connecting two vertices and carrying a digital value. Each vertex representing a logic gate in the graph is associated with a Boolean function describing the gate behavior. By definition, a *fault* occurs in the circuit when a given logic gate is not evaluated with its correct Boolean function.

Microarchitectural faults in the graph are parametrized with the three following attributes: *location*, *effect*, and *number*. The *location* lists the vertices $v \in V$ targeted by the fault injection. The *effect* specifies the fault effect by associating a faulty Boolean function with the targeted vertices. Finally, the *number* describes the maximum number of vertices simultaneously affected by a fault.

B. Transition System for Sequential Hardware Circuits

We model sequential hardware circuit [14, §2.1.2] as a transition system $\mathcal{M} = (S, S_0, X, T)$ where:

- S is the set of circuit states,
- $S_0 \subseteq S$ is the set of initial states,
- X is the set of circuit inputs,
- $T\colon S \times X \to S$ is the transition function of the circuit.

A *system state* $s \in S$ corresponds to a valuation of state-holding elements (e.g., microarchitectural registers, memories) in the hardware design. Assuming there are n state-holding elements, denoted as *registers* in the following, the state s can be seen as a vector of the n register values $s := \langle r_1,...,r_n \rangle$.

An *initial state* $s_0 \in S_0$ is a system state where each register $r_{j,1\leq j\leq n}$ is evaluated with its initial values. Initial states can be determined according to register reset values, for instance. Uninitialized registers imply multiple initial states in the system.

An *input* $x \in X$ is a vector $x := \langle i_1,...,i_m \rangle$ composed with a valuation of the m input variables given to the system. Circuit outputs are not considered in the formalization as they are not useful for the rest of this work.

The *transition function* T describes the valid transitions from a state s_i and an input vector x_i to the state s_{i+1} at the next circuit clock cycle. The function T is determined with the combinational logic of the hardware design and can be decomposed in n *register transition functions* δ_{r_j} where each register next state value is computed by applying δ_{r_j} to the current state and the input vector.

$$s_{i+1} = T(s_i, x_i) = \langle \delta_{r_1}(s_i, x_i), \ldots, \delta_{r_n}(s_i, x_i) \rangle$$

In hardware circuits, intermediate combinational results are often factorized to minimize the number of operations. This optimization avoids duplicating identical operations to reduce hardware costs. We denote *combinational functions* these intermediate results. Consequently, the register transition functions δ_{r_j} can be expressed as a composition of these intermediate combinational functions.

C. Yosys Framework

Yosys [12] is an open-source synthesis tool with a compiler-like infrastructure. Its frontend takes as input a design description using a hardware description language like Verilog. The Yosys intermediate language, called RTLIL, is a netlist composed of gates and wires. The backend converts the RTLIL design into various outputs ranging from technological targets like FPGA or ASIC to formal languages.

In particular, Yosys can transform a hardware design description into a hardware transition system. Supported formal languages are AIGER, SMV, BTOR2 and SMT-LIB. AIGER [15] describes hardware systems at the bit level using an and-inverter graph. The SMV language [16] is provided by the symbolic model checker NUXMV [17] and describes finite and infinite transition systems. NUXMV lifts the verification from the bit level to the word level with data types like bit vectors or memories. BTOR2 [18] is a word-level generalization of AIGER and provides word-level data types, registers, and memories. Finally, the SMT-LIB language [19] is the standard specification to describe SMT problems and is broadly supported by SMT solvers. SMT-LIB can also specify hardware transition systems using the quantifier-free bit-vector theory. Yosys can produce VCD waveform traces of the successive hardware states from model-checker outputs.

Finally, Yosys has built-in options to simulate the design and set the register's initial values. These functionalities allow a user to configure the initial state of the circuit before converting it into a hardware transition system $\mathcal{M}$.

III. Faulty System Modeling

Hardware analysis often relies on equivalence-checking techniques [4], [20] to capture faults that induce a different circuit behavior compared to a reference model. But these

methods can only classify fault effects at the circuit level and cannot determine whether the faults have consequences on the running software. To analyze the consequences of a fault described at the microarchitectural level on the software, we need to observe its propagation in the system. For this purpose, in the following, we perform model checking to capture the successive system states between the fault injection and the fault manifestation. This section defines a faulty transition system comprising the program, the hardware, and the attacker model.

A. Bringing the Software and the Hardware Together

The hardware processor design is modeled as a transition system $\mathcal{M} = (S, S_0, X, T)$, as introduced in Section II. The software program is the sequence of instructions to be executed on the processor. The program is encoded in the initial state of a memory modeled simultaneously with the processor. Accordingly, the initial state S_0 of the system restricts the possible processor executions to the software program under study. The input set X of the system does not represent the program as it is already encoded in the transition system. Instead, system inputs are used to model the fault injections applied during the processor operation. The attacker model and the faulty transition system are introduced in the following of this section.

B. Attacker Model

We define an *attacker model* which specifies how the attacker can perturb the system operation. This model relies on the definition of microarchitectural faults introduced in Section II-A and extends this definition to describe the attacker's capabilities on a hardware transition system. The attacker model comprises *i)* the attacker's goal expressed as a reachability property φ, *ii)* the number of faults N that the attacker can inject into the system, and *iii)* the fault model. The *fault model* is parametrized by the triplet $(\mathcal{L}, \mathcal{T}, \mathcal{E})$ and describes the possible modifications that a fault may induce on the system.

- $\mathcal{L}$ is the set of possible locations of the fault,
- $\mathcal{T}$ is the timing range of the fault injection,
- $\mathcal{E}$ is the set of possible effects of the fault.

The *fault location* $\mathcal{L}$ is a set that denotes the registers targeted by the fault injection, i.e., $L \subseteq \{r_1, ..., r_n\}$.

The *timing range* $\mathcal{T} \subset \mathbb{N}$ of the fault is a set of non-negative integers that specifies when the fault injection can occur in the system. For example, a fault can be injected in the transition system between states s_i and s_{i+1} if $i \in \mathcal{T}$.

The *fault effect* $\mathcal{E} \subset \{set, reset, flips, \dots\}$ is a set of functions that modifies how a register is updated in the next state. For instance, a fault $e \in \mathcal{E}$ injected in register r_j consists in replacing the transition function δ_{r_j} with the faulty transition $\delta^e_{r_j}$ within the same domain, i.e., it produces the same output data type. A non-exhaustive list of possible effects is given below for an 8-bit register:

$e \in \mathcal{E}$	$: \delta_{r_j}(s) \mapsto \delta^e_{r_j}(s)$
$reset$	$: \delta_{r_j}(s) \mapsto 0\text{x}00$
set	$: \delta_{r_j}(s) \mapsto 0\text{xff}$
$flips$	$: \delta_{r_j}(s) \mapsto \neg\delta_{r_j}(s)$
$flip_{\text{lsb}}$	$: \delta_{r_j}(s) \mapsto \delta_{r_j}(s) \oplus 0\text{x}01$

The *number of fault injections* N restricts the possibilities offered by the fault model $(\mathcal{L}, \mathcal{T}, \mathcal{E})$. An attacker can use at most N faulty transitions δ^e_r to compute the next system states.

The *attacker's goal* φ is a reachability property defined on the transition system $\mathcal{M}$. It represents a vulnerability that the attacker wants to reach in order to create an exploit on the system by injecting faults. In the system's normal operation, such an exploit should not exist. In other words, $\neg\varphi$ is a system's invariant that the attacker wants to break.

Let us illustrate some practical instantiations of the attacker model we defined. Laser fault injections are accurate in space and time and can be modeled with only one or two bit-flip. On the other hand, voltage or clock glitches are less accurate and can affect the whole design. We may model them with multiple bit-set and bit-reset.

C. Faulty Hardware Transition System

The *faulty transition system* $\mathcal{M}^F = (S, S_0, X, T)$ results from the modification of the hardware and software transition system $\mathcal{M}$ and the attacker model $((\mathcal{L}, \mathcal{T}, \mathcal{E}), N, \varphi)$. First, a new variable $cnt \in [\![0, N]\!]$ is added in the system model to encode the maximum number of fault injections N. The cnt is incremented each time a faulty transition is applied and cannot be targeted by the fault model, i.e., $cnt \notin \mathcal{L}$. Then, for each targeted register r_l in $\mathcal{L}$, we add a new input x_l to the system to control the fault injection. The input x_l determines whether a fault is injected in register r_l, and hence, if the value of register r_l in the next state should be computed using the normal transition function δ_{r_l} or the faulty transition $\delta^e_{r_l}$.

$$r'_l = \begin{cases} \delta_{r_l}(s) & \text{if } x_l = \text{False} \\ \delta^e_{r_l}(s) & \text{if } x_l = \text{True} \end{cases}$$

A possible extension of this faulty transition system is to expose intermediate *combinational functions* often used in hardware circuits, as introduced in Section II. We can then extend our fault model and the resulting faulty transition system to target these combinational functions with fault injection. This extension is not formalized here but is implemented in the µARCHIFI tool.

IV. TRANSITION SYSTEM VERIFICATION

This section introduces verification techniques on a faulty transition system. In addition, we describe how the knowledge of the running software can be leveraged to refine the transition system verification.

A. Verification Problem Statement

In Section III, we model the system under attack as a transition system $\mathcal{M}^F = (S, S_0, X, T)$. The set of initial states S_0 describes the possible software execution path to analyze, and the inputs X control the possibilities of the attacker to inject faults in the system. The verification problem is then a reachability property verification where an attacker wants to find a sequence of states $(s_0, s_1, ..., s_k) \in S^{k+1}$ such that:

- s_0 is an initial state, i.e., $s_0 \in S_0$,
- transition between states s_i and s_{i+1} is valid, i.e., it exists an input $x_i \in X$ such that $s_{i+1} = T(s_i, x_i)$,
- the number of faults injected in the system does not exceed the attacker capacity, i.e., $cnt \leq N$ and,
- $\varphi(s_k)$ is true.

Such a path in the transition system allows an attacker to identify an instance of the fault model and a software execution trace that verifies the property φ.

Different strategies exist to iterate over the transition model to verify the property. *Unbounded verification techniques* [21]–[24] prove the property in the general case, but the data dependency and the transient nature of faults make these techniques ill-suited [25] on the fault injection problem. *Bounded verification techniques* like Bounded Model Checking (BMC) [26], [27] prove the property from an initial state for a limited number of transitions, fixed a priori with a bound. This *bound* is typically set according to the length of execution trace of the analyzed software. In this work, we rely on bounded verification techniques to address the fault verification problem.

The remainder of this section introduces software-related considerations for applying well-known optimization techniques to speed up our BMC verification.

B. Sandboxing Execution Paths

Sandboxing is a general technique that adds a global constraint on the model to reduce the problem's state space. We apply sandboxing to restrict the Program Counter (PC) to a range of values that a simple static analysis can retrieve from the addresses in the binary, e.g., using objdump-like tool. The verification framework then stops exploring software execution paths that do not satisfy this sandboxing constraint. Consequently, the BMC procedure can also terminate faster when the entire state space has been explored. However, adding such a global constraint on the model may lose the *k*-completeness of the bounded verification procedure. This technique must therefore be used to explore possible vulnerabilities rather than prove the system's robustness.

While only one PC exists at the software level, several microarchitectural registers store its value in the processor design. The fetch stage PC speculates on the next addresses to be read from memory. Applying the sandboxing technique on this register would thus require relaxing the sandboxing constraint. The execute stage PC misses unconditional branches that are resolved directly in the decode stage. We therefore implement sandboxing by constraining the PC of the decode stage of in-order processors, as presented later in Section VI.

Algorithm 1: Bounded model checking (BMC) with concretization

Input: transition system $\mathcal{M} = (S, S_0, X, T)$, reachability property φ, BMC bound k, concretization depth m, number of concretizations L
Output: a path π if a φ is reachable, *None* otherwise

```
1  Function BMC_Concretizing(M, φ, k, m, L) is
       // BMC() checks the reachability of φ up to
          the bound. M, φ, ϕ, i are global variables.
2      Function BMC(bound) is
3          while i < bound do
4              if ϕ ∧ φ(s_i) is SAT then
5                  exit( π := (s_0, ..., s_i))
6              i ← i + 1
7              ϕ ← ϕ ∧ (s_i = T(s_{i-1}, x_{i-1})) // Unrolling
8          end
9      end

       // Initial BMC verification up to step m
10     ϕ ← S_0(s_0) ;   i ← 0
11     BMC(m)
       // Concretization loop
12     ψ ← ϕ ;   set_PC ← ∅ ;   iter ← 1
13     incomplete_enum ← False
14     while ψ is SAT do
15         address ← get_model(ψ)(PC)
16         set_PC ← set_PC ∪ address
17         ψ ← ψ ∧ (PC ≠ address)
18         if iter = L then
19             incomplete_enum ← True; Break
20         iter ← iter + 1
21     end
       // Parallel BMC verifications for each
          concretized path up to bound k
22     for address ∈ set_PC do
23         ϕ ← ϕ ∧ (PC = address)
24         BMC(k) // run BMC on ϕ (concretized paths)
25     end
26     if incomplete_enum then
27         ϕ ← ϕ ∧ (PC ∉ set_PC)
28         BMC(k) // run BMC on ϕ (remaining paths)
29 end
```

C. Concretizing Execution Paths

BMC algorithms [27] typically unroll the transition system in a formula ϕ to check the reachability of a property φ. As a result, solving the formula ϕ suffers from the increasing number of variables and clauses.

We apply the general *concretizing* technique to split ϕ into sub-formulas encoding the different software execution paths. Like *sandboxing*, we rely on the PC to distinguish between these different execution paths. However, a given PC value can refer to several microarchitectural contexts if more than one execution path can reach this address at the same time. This aspect is discussed at the end of this section.

The concretization procedure is detailed in Algorithm 1. After initializing the formula ϕ with an initial state s_0 and performing BMC up to bound m (lines 10-11), a concretization loop enumerates the possible values for the PC (lines 12-21). This loop successively asks an SMT solver to give models

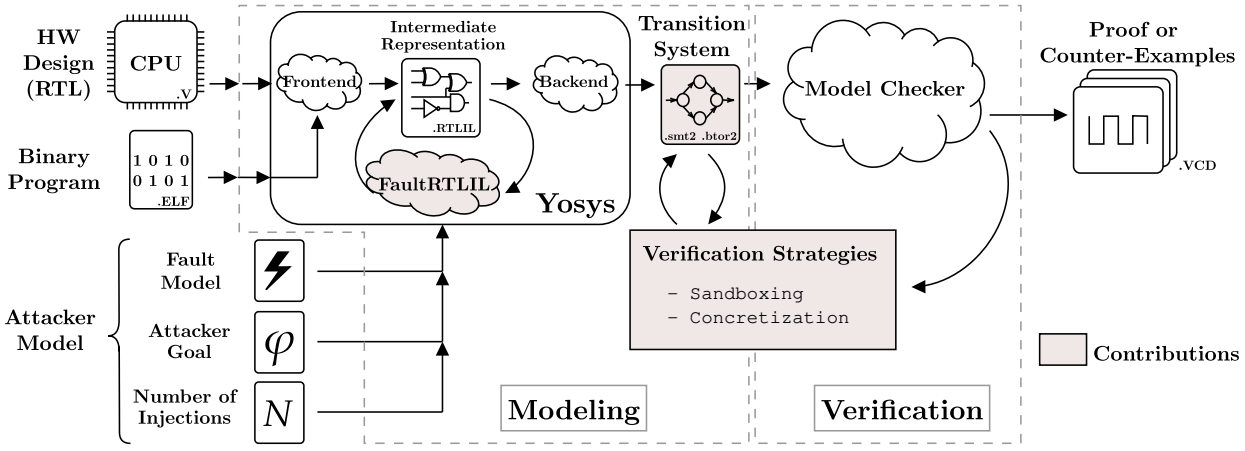

Fig. 1: µARCHIFI architecture and verification tool-chain.

of the system with different PC values. It stops when no more system model exists, i.e., ψ becomes unsatisfiable, or after a given number of concretizations L (lines 18-19). A new BMC procedure is performed for each enumerated PC value until bound k (lines 22-25). When the PC enumeration is incomplete, the remaining paths are encoded within a single formula and checked together (lines 26-28). A program analysis can identify branches' locations in the program and determine the optimal depth m the user should perform the concretization.

D. Discussion

Both the *sandboxing* or the *concretization* techniques reduce the state space to explore by adding terms and clauses to the formula encoding the problem. However, this general approach has some limitations. To speed up verification, a trade-off must be found between eliminating execution paths and the number of additional state variables that increase the complexity of the formula to check. For example, we might want to add constraints ensuring that injected faults lead to a different system behavior than the fault-free reference model. This technique allows to focus on analyzing effective faults while ineffective ones are ignored. However, it requires capturing the complete microarchitectural state twice, leading to an excessively complex system encoding formula. The verification times thus increase.

V. TOOL IMPLEMENTATION

This section introduces the µARCHIFI tool we developed to generate a formal transition system from a hardware description, a program, and an attacker model. First, we detail how a user can use the tool, then we give its implementation details.

A. µARCHIFI Usage

µARCHIFI, illustrated in Fig. 1, takes as input a processor hardware description in Verilog, a binary software program, and an attacker model comprising the fault model. First, the user can simulate the execution of the target program, compiled for the corresponding ISA, on the hardware design to set the initial state of the hardware right before the instruction sequence to analyze formally. Then, the user needs to specify the attacker model comprising the goal φ, the maximal number of faults N, and the fault model (location, timing, and fault effects). This model is automatically integrated into the system through the `FaultRTLIL` pass. The attacker's goal can also be specified into the hardware design using the SystemVerilog Assertion subset supported by Yosys. Finally, the µARCHIFI tool produces a transition system, as introduced in Section III-B, in SMT-LIB or BTOR2 format. The faulty transition system can be verified using external model-checking tools compatible with these input formats, like AVR [25], PONO [28] or BTORMC [18].

When an external model checker finds a counterexample, as illustrated in the *verification* box in Fig. 1, a VCD file reports precisely where the fault is injected and when the attacker's goal is reached. However, understanding the propagation of faults and their consequences requires human expertise, but this task can be facilitated by external tools that perform differential traces comparison against a reference model.

Additional global constraints for the *sandboxing* technique can be specified to the model checker or can be directly included in the input Verilog design parsed by Yosys. The *concretization* technique requires an external model checker to enumerate possible execution paths and is thus not integrated into the µARCHIFI tool.

B. µARCHIFI Architecture

Fig. 1, in the *modeling* box, illustrates the integration of the µARCHIFI tool with the Yosys framework. Yosys can parse and translate a hardware specification into formal languages, allowing us to focus on the automated integration of an attacker model into the system. We work on the RTLIL intermediate representation of Yosys to get the best expressivity and exhaustiveness to specify the fault model. Besides, as the Yosys RTLIL translation preserves all signal and register names from the Verilog processor design, the user can accurately select fault locations based on name pattern matching, cell type, or cell-width filtering. In addition, the

TABLE I: Use Cases description and expected verification result.

	Hardware Design			Software Program		Attacker Model					BMC Results	
	Name	Logic Gates	Flip-Flops	Name	gcc Flag	Attacker goal φ	Location	Timing	Effect	N	k	Reachability
Use Case I	CV32E40P	2842	179	VerifyPIN_V7	`Og`	Bypass authentication	Flip-Flops in Control path	60:75	Symbolic	1	75	φ reachable
Use Case II	Secure Ibex	4422	211	VerifyPIN_V1	`Os`	Bypass authentication	Flip-Flops in Lockstep	*	Symbolic	5	46	φ unreachable
Use Case III	Ibex	1983	114	KeySchedule (AES)	`Os`	Set expanded key to 0	Combinational in EX Stage	*	Reset	2	38	φ unreachable

preservation of names facilitates the generation of comprehensive counterexamples.

Our work extends the Yosys tool by proposing a `FaultRTLIL` translation pass, illustrated in Fig. 1, that takes the attacker's model $(\mathcal{L}, \mathcal{T}, \mathcal{E}), N, \varphi$ in input and integrates it into the system. This integration is achieved in several steps. First, a new RTLIL register is created to encode the maximum number of fault injections N the attacker can inject into the system. Then, a clock is added to control the timing range of the fault injection $\mathcal{T}$. Finally, additional logic functions are inserted into the intermediate representation for each location $l \in \mathcal{L}$ potentially targeted by a fault injection, modeling the possible fault effects $\mathcal{E}$. The targeted elements are then replaced by an *if-then-else* structure controlled by a *fault selector*. Fault selectors are exposed as system inputs and indicate whether the fault should be injected or not. The maximum counter value N and the clock for fault injection timing $\mathcal{T}$, introduced previously, are used to constrain the fault selectors.

VI. Evaluation

This section illustrates the use of µArchiFI in three case studies, applies the verification strategies introduced in Section IV, and discusses the tool's limitations. The µArchiFI implementation, the case studies, and the experimental results are publicly available[2].

All verifications have been executed on an 11th Gen Intel(R) Core(TM) i7-1185G7 CPU platform. Every program presented in this section is compiled with the RISC-V toolchain for the RV32IMC architecture (gcc version 10.2.0). For each verification, the BMC bound k is fixed according to the longest program execution trace plus a 10-percent increment to capture possible modifications in the control flow.

A. Use Case I: Robust Software

Use Case I illustrates the possibility for a user to analyze the robustness of a secure program running on a processor.

Software. We consider a `memcmp`-like authentication mechanism from the FISSC benchmark suite [29]. This collection provides eight versions of the VerifyPIN program embedding software countermeasures against fault injections. The VerifyPIN program compares two 3-digit[3] PIN codes stored in memory: a user and a secret PIN. The user can authenticate when the two codes are identical. In the following, PIN values are symbolic, but we assume that the user PIN and the secret PIN are different in each of their digits. In Use Case I, we target VerifyPIN_V7 with the most software countermeasures. It implements hardened booleans, constant iteration, loop counter check, inline PIN comparison, and duplication of critical tests. VerifyPIN_V7 is compiled with the optimization flag `Og` to prevent the compiler from removing the countermeasure. The program runs in constant time, in 69 clock cycles.

Hardware. We execute the program on the 32-bit, in-order, 4-stage pipeline CV32E40P RISC-V core from the OpenHW group [30]. The version under study does not provide any security countermeasures.

Attacker Model. In this system, the attacker aims to bypass the secure authentication mechanism without triggering the software countermeasures.

$$\varphi_{\mathrm{I}} := (authenticated \wedge \neg software\_alert)$$

The VerifyPIN_V7 program implements the authentication process in two steps. First, a constant-time loop sets a Boolean to True if a difference is detected between the two PINs. Second, a comparison is performed to test the Boolean value and allow the authentication. We evaluate the robustness of the second comparison block against a single fault injected on the sequential logic of the processor control path. The considered fault model targets 102 registers among 179 in the processor. Use Case I is summarized in Table I.

Verification Results. Table II compares verification performance between three model checkers with and without faults. Performing the verification without fault ensures that the attacker goal φ does not hold outside of an attack. The analysis results in Table I highlight that the attacker can bypass the authentication by injecting a single fault. Counterexamples provided by the model checkers permit the user to find the exact location of the fault that leads to the vulnerability φ_{I}. All solvers found the same fault model on this use case, but we can observe that Pono is faster to solve the model-checking problem.

B. Use Case II: Robust Hardware

Use Case II details how a user can determine whether a fault injected into a secure processor can induce a vulnerable behavior on the software without being detected by the hardware countermeasure.

Software. We consider VerifyPIN_V1, the baseline version of the VerifyPIN collection, without any countermeasure. As in Use Case I, the same constraint is applied to user- and secret-PIN, which are still symbolic. VerifyPIN_V1 is compiled with the optimization flag `Os`.

Hardware. The Ibex [31] is a parametrizable open-source 32-bit, in-order processor. We analyze the *small* version of the

[2]https://doi.org/10.5281/zenodo.7958412

[3]VerifyPIN uses 4-digit PINs in its original version.

core [32] in its *secure* configuration. The secure Ibex implements protections against physical attacks like the redundancy-based lockstep mechanism that instantiates the core twice and compares the outputs. The duplicated core is called the *shadow core* and an alert signal is triggered if an attack has been detected during the operation of the processor.

Attacker Model. In this second use case, the attacker still aims to bypass the secure authentication mechanism without triggering the hardware countermeasures.

$$\varphi_{\text{II}} := (authenticated \wedge \neg hardware\_alert)$$

The considered attacker model cannot inject more than five faults into the system. Fault locations are limited to the sequential logic in the shadow core since we do not want to inject the same fault in both cores.

Verification Results. Table I reports that an attacker cannot bypass the secure authentication with the considered fault model. This use case leverages the fact that the secure Ibex implements hardware countermeasures. On the one hand, assuming that the $hardware\_alert$ cannot be triggered makes sense as the attacker wants to bypass the authentication without being detected. On the other hand, it helps the solver simplify the formula during the verification. Table II reports verification performance. BTORMC fails to solve the problem, and we stop the verification after 2 hours.

C. Use Case III: Cryptographic Software

Use Case III details how a user can apply the tool to software implementations of cryptographic algorithms.

Software. Tiny AES [33] is a small software implementation of the encryption algorithm. The key schedule function of the AES program expands the key into several separate keys for each round of AES. We focus here on a round of the key schedule function from the 128-bit AES. The program is compiled with the optimization flag `Os`.

Hardware. We run the key schedule function on the baseline version of the *small* Ibex core without any countermeasure.

Attacker Model. The attacker wants to set to zero a byte in the penultimate round key. An attacker can then use the observation of such an effect to perform differential fault analysis [34], [35]. Fault consequences are observed at the end of the key schedule function to limit the analysis to a small sequence of instruction.

$$\varphi_{\text{III}} := (9^{th}\ Round\_key_{\text{byte}} = 0)$$

To attempt to reach the property φ_{III}, we allow an attacker to inject up to two word-reset faults anywhere in the execute stage of the Ibex.

Verification Results. As reported in Table I, an attacker cannot reach his goal with the considered fault model. Additional verification not described here shows that a more powerful attacker reaches his goal with four fault injections instead of two. We can also note that the verification of φ_{III} on the AES program without fault is faster than both Use Case I and II because the AES key is fixed for while the two 3-digit PINs are symbolic for the VerifyPIN program.

TABLE II: Use-cases verification time with three model checkers.

	Without Fault			With Faults		
	PONO	YOSYS-BMC	BTORMC	PONO	YOSYS-BMC	BTORMC
Use Case I	12.6s	11.1s	1.5s	107s	249s	273s
Use Case II	20.7s	10.6s	3.5s	250s	373s	timeout
Use Case III	0.3s	2.4s	0.1s	313s	1945s	3427s

TABLE III: Verification time improvement with the sandboxing technique wrt. the baseline verification time with faults in Table II.

	PC Sandboxing	PONO	YOSYS-BMC	BTORMC
Use Case I	0x1c4 $\leq$ PC $\leq$ 0x234	110s (+2.8%)	242s (-2.8%)	205s (-24.9%)
Use Case II	0x84 $\leq$ PC $\leq$ 0x114	206s (-17.6%)	297s (-20.4%)	timeout
Use Case III	0x40 $\leq$ PC $\leq$ 0xc0	107s (-65.8%)	1454s (-25.2%)	1659s (-52.0%)

TABLE IV: Verification time improvement with the concretization technique wrt. the baseline verification time with faults from Table II.

			Concretization	
	Concretized step	Baseline	Parallelized	Accumulated
Use Case I	62 (Status comparison)	249s	189s (-24.1%)	509s (+104%)
Use Case II	31 (PIN comparison)	373s	304s (-18.5%)	891s (+139%)
Use Case III	23 (No branch instruction)	1945s	1504s (-22.7%)	2955s (+51%)

D. Influences of Verification Strategies

Sandboxing Execution Paths. For each use case introduced before, we determine the range of possible values for the program counter (PC) by dumping addresses from the binary file. Here, the possible addresses are contiguous, and we add a global constraint on the system to force the PC to stay in this set of values. Table III illustrates that the sandboxing strategy results in an improvement of the performances up to 65%, and these additional constraints do not prevent model checkers from retrieving the vulnerability highlighted in Use Case I.

Such improvements are due to two factors. First, some fault effects are not analyzed if they lead to PC values out of the memory range considered. Secondly, the verification may end before the bound k if all execution paths in the system exit from the considered address range. We also observe that improvements vary between the different solvers even if PONO remains more efficient on the use cases analyzed.

Concretizing Execution Paths. We apply the concretization strategy for each use case with an enumeration bound $L = 3$ to split the bounded verification procedure into $L + 1$ sub-verifications (c.f., Algorithm 1). We arbitrarily set $L = 3$ as it provides the best performance in these practical use cases. A higher value of L increases the *accumulated* verification time without improving the *parallelized* time.

Table IV reports the concretization steps, the baseline verification time from Table II, and the concretization performance. We show each experiment's wall-clock time and accumulated verification time since we can parallelize the executions. Performance is given for the YOSYS-BMC since other evaluated model checkers do not permit to retrieve the SMT formula encoding the unrolled system.

On Use Case I, we concretize the execution at the first branching instruction targeted by fault injection. It corresponds to the PIN-status comparison to allow authentication (step 62).

This results in an improvement of the verification time by 24%. On Use Case II, we apply concretization during a PIN-digit comparison and enumerate PC values associated to different execution paths. However, few performance improvements are observed, especially regarding the accumulated verification time. We believe this is due to the hardware countermeasure that already prevents executing different paths due to the faults. No branching instruction exists on Use Case III. However, many execution paths are possible due to fault injections. Concretization is applied at step 23, at half of the verification time. This results in a 22.7% verification time improvement.

In conclusion, concretization often improves the verification time thanks to the parallelization of the executions. However, these verification times remain higher than the one obtained when using the PONO model checker (Table II).

VII. Related Work

Similar works propose modeling and verification methodologies to study fault injection effects. Classical verification methods like simulation are used [3], [6], [36], [37], but they are often not exhaustive, and it is often difficult to highlight corner cases, like the Prefetch Buffer introduced in Section I. For instance, VERFI [3] needs to set a fixed input test vector to evaluate cryptographic implementation robustness to faults. In the following, we will discuss papers that propose a formal framework to analyze fault effects on the system.

First, some works analyze fault effects on hardware implementation [2], [4], [20]. Formal techniques were first dedicated to analyzing cryptographic circuits with equivalence checking. AutoFAULT tool [2] can parse and transform a small block cipher written in VHDL into a SAT formula to determine if a fault can induce a wrong ciphertext. The FIVER tool [4] translates Verilog netlists to Binary Decision Diagram to compare a fault-free circuit with a faulty copy to determine the fault effects. FIVER symbolically checks every possible input and classifies fault effects according to the expected reference behavior. Faults are classified as *effective*, *ineffective*, or *detected*, depending on whether they induce a different behavior and if the countermeasure (if any) detects them. SYNFI [20] can parse technological netlists to prove the equivalence between golden and faulty circuits to detect if the synthesis step removes countermeasures. However, SYNFI does not handle sequential verification since the design to analyze is unrolled to perform equivalence checking, and thus, the tool cannot analyze software. In comparison, µARCHIFI does not support advanced technological netlists, but we still support any Verilog or SystemVerilog design by plugging our translation pass into the Yosys tool. In addition, we take advantage of a simplified word-level netlist to bridge the gap with the software and facilitate the analysis of the transitional system. We also keep the sequential logic instead of unrolling and flattening the whole design to use model-checking verification techniques.

On the other hand, some additional works model and study faults at the software level and analyze fault effects on the control flow [5], [7], [38]–[40]. These approaches address the binary or the Instruction Set Architecture level and propose methodologies to analyze the robustness of the software programs. SAMVA [40] assesses a binary program against multiple instruction-skip attacks with static analysis. The proposed method by Ducousso et al. [7] permits scaling on large programs like bootloader with up to 10 fault injections. However, these works do not consider the execution platform, and the generic fault models used are sometimes inadequate to model microarchitectural implementation details.

Furthermore, commercial tools offer all the building blocks required for such a fault injection analysis, but their closed nature prevents users from integrating them into the same verification framework. SystemVerilog Assertion (SVA), supported by tools such as Synopsys VC Formal or Siemens QuestaVerify, could define the attacker's goal, but is not suitable for fault modeling. On the other hand, tools such as Cadence JasperGold offer support for fault injection but do not consider software. In short, none of these tools address the verification of software and hardware against fault injection.

Finally, apart from fault injection, some works [41], [42] tackle the problem of hardware-software co-verification using BMC verification. Schmidt et al. [42] propose to separate the control path and the computation in the modeling to cope with system complexity. However, this compositional approach is undermined when the underlying hardware is corrupted by fault injection since data and control are then both impacted.

VIII. Conclusion

In this paper, we propose a faulty transition system to model the hardware implementation of a processor and the software program conjointly. This modeling allows to formally analyze and study the propagation of faults in the microarchitecture and their consequences on the system behavior. The µARCHIFI tool automatically implements this model, from the hardware design description at the RTL level, in Verilog, a binary program, and a specification of the attacker model. µARCHIFI allows to specify a large variety of microarchitectural fault models with high expressiveness. We illustrate the use of µARCHIFI on three use cases encompassing complete microarchitectural designs of RISC-V processors representative of the embedded market and binary programs of hundreds of machine instructions. We discuss possible strategies to improve the verification performance. At this stage, the user of µARCHIFI must find a sweet spot between the size of the hardware design, the size of the analyzed program, and the complexity of the fault model. Future work will focus on combining several verification strategies leveraging software, such as sandboxing and concretization techniques, but also robust hardware embedding countermeasures to analyze fault injections on a larger scale.

REFERENCES

[1] B. Yuce, P. Schaumont, and M. Witteman, "Fault Attacks on Secure Embedded Software: Threats, Design and Evaluation," *Journal of Hardware and Systems Security*, Jun. 2018.

[2] J. Burchard, M. Gay, A.-S. M. Ekossono, J. Horáček, B. Becker, T. Schubert, M. Kreuzer, and I. Polian, "AutoFault: Towards Automatic Construction of Algebraic Fault Attacks," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2017.

[3] V. Arribas, F. Wegener, A. Moradi, and S. Nikova, "Cryptographic Fault Diagnosis using VerFI," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, Dec. 2020.

[4] J. Richter-Brockmann, A. Rezaei Shahmirzadi, P. Sasdrich, A. Moradi, and T. Güneysu, "FIVER – Robust Verification of Countermeasures against Fault Injections," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, Aug. 2021.

[5] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections," in *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*, Mar. 2014.

[6] M. Hoffmann, F. Schellenberg, and C. Paar, "ARMORY: Fully Automated and Exhaustive Fault Simulation on ARM-M Binaries," *IEEE Transactions on Information Forensics and Security*, vol. 16, 2021.

[7] S. Ducousso, S. Bardin, and M.-L. Potet, "Adversarial Reachability for Program-level Security Analysis," in *32nd European Symposium on Programming (ESOP)*, 2023, pp. 59–89.

[8] J. Laurent, C. Deleuze, F. Pebay-Peyroula, and V. Beroulle, "Bridging the Gap between RTL and Software Fault Injection," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 17, no. 3, May 2021.

[9] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software Fault Resistance is Futile: Effective Single-Glitch Attacks," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Aug. 2016.

[10] S. Tollec, M. Asavoae, D. Couroussé, K. Heydemann, and M. Jan, "Exploration of Fault Effects on Formal RISC-V Microarchitecture Models," in *2022 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, Sep. 2022.

[11] J. Laurent, V. Beroulle, C. Deleuze, and F. Pebay-Peyroula, "Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.

[12] C. X. Wolf, "Yosys open synthesis suite," https://github.com/YosysHQ/yosys, 2016.

[13] J. Richter-Brockmann, P. Sasdrich, and T. Guneysu, "Revisiting Fault Adversary Models – Hardware Faults in Theory and Practice," *IEEE Transactions on Computers*, 2022.

[14] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, Apr. 2008.

[15] A. Biere, K. Heljanko, and S. Wieringa, "AIGER 1.9 and beyond," 2011.

[16] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani, "Verilog2SMV: A Tool for Word-level Verification," in *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

[17] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv Symbolic Model Checker," in *Computer Aided Verification*, 2014.

[18] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , BtorMC and Boolector 3.0," in *Computer Aided Verification*, 2018.

[19] C. Barrett, P. Fontaine, and A. Stump, "The SMT-LIB Standard," 2010.

[20] P. Nasahl, M. Osorio, P. Vogel, M. Schaffner, T. Trippel, D. Rizzo, and S. Mangard, "SYNFI: Pre-Silicon Fault Analysis of an Open-Source Secure Element," May 2022.

[21] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," in *Formal Methods in Computer-Aided Design*, 2000, pp. 127–144.

[22] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Computer Aided Verification*, vol. 2725. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[23] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *Verification, Model Checking, and Abstract Interpretation*, 2011.

[24] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, Oct. 2011.

[25] A. Goel and K. Sakallah, "AVR: Abstractly Verifying Reachability," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Biere and D. Parker, Eds., 2020.

[26] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Formal Methods in System Design*, 2001.

[27] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer International Publishing, 2018.

[28] M. Mann, A. Irfan, F. Lonsing, Y. Yang, H. Zhang, K. Brown, A. Gupta, and C. Barrett, "Pono: A Flexible and Extensible SMT-Based Model Checker," in *Computer Aided Verification*, 2021, pp. 461–474.

[29] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, "FISSC: A Fault Injection and Simulation Secure Collection," in *Computer Safety, Reliability, and Security*, 2016.

[30] OpenHW group, "OpenHW Group CV32E40P User Manual," https://cv32e40p.readthedocs.io/en/latest/.

[31] LowRISC, "Ibex: An embedded 32 bit RISC-V CPU core," https://ibex-core.readthedocs.io/en/latest/.

[32] "Ibex RISC-V Core github repository," https://github.com/lowRISC/ibex#configuration.

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

[34] J. Takahashi, T. Fukunaga, and K. Yamakoshi, "DFA Mechanism on the AES Key Schedule," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*. Vienna, Austria: IEEE, Sep. 2007.

[35] S. S. Ali and D. Mukhopadhyay, "A Differential Fault Analysis on AES Key Schedule Using Single Fault," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, Sep. 2011.

[36] J. Grycel and P. Schaumont, "SimpliFI: Hardware Simulation of Embedded Software Fault Attacks," *Cryptography*, vol. 5, no. 2, Jun. 2021.

[37] T. Given-Wilson, N. Jafri, and A. Legay, "Combined software and hardware fault injection vulnerability detection," *Innovations in Systems and Software Engineering*, vol. 16, no. 2, Jun. 2020.

[38] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "SymPLFIED: Symbolic program-level fault injection and error detection framework," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, Jun. 2008.

[39] J.-B. Bréjon, K. Heydemann, E. Encrenaz, Q. Meunier, and S.-T. Vu, "Fault attack vulnerability assessment of binary code," in *Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems*. Valencia Spain: ACM, Jan. 2019.

[40] A. Gicquel, D. Hardy, K. Heydemann, and E. Rohou, "SAMVA: Static Analysis for Multi-fault Attack Paths Determination," in *Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2023, pp. 3–22.

[41] D. Groβe, U. Kühne, and R. Drechsler, "HW/SW co-verification of embedded systems using bounded model checking," in *Proceedings of the 16th ACM Great Lakes Symposium on VLSI - GLSVLSI '06*. Philadelphia, PA, USA: ACM Press, 2006.

[42] B. Schmidt, C. Villarraga, T. Fehmel, J. Bormann, M. Wedler, M. Nguyen, D. Stoffel, and W. Kunz, "A New Formal Verification Approach for Hardware-dependent Embedded System Software," *IPSJ Transactions on System LSI Design Methodology*, pp. 135–145, 2013.