Synthesis of RTL-based Characterization Programs for Fault Injection

Jonah Alle Monne*, Guillaume Bouffard†, Damien Couroussé*, Mathieu Jan‡,
*Univ. Grenoble Alpes, CEA, List, F-38000, Grenoble, France,
†ANSSI, ‡Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

***Abstract*—Fault injection attacks pose a significant threat to the security of embedded devices. While their effects are commonly modeled as instruction skips or data corruption, characterizing these faults requires programs that expose software-visible faulty behavior.**

However, many fault effects originate from microarchitectural elements, making them difficult to identify using existing approaches. On one hand, Register Transfer Level (RTL) analyses provide fine-grained insights but rely on abstract models that may not fully reflect the physical circuit. On the other hand, empirical characterization captures real faults but requires extensive experimentation and often reveals multiple fault models simultaneously, complicating precise identification.

To address this gap, we propose an automated methodology that synthesizes characterization programs specifically designed to expose targeted microarchitectural fault models using a model-checking algorithm. Our methodology also assesses additional fault models revealed by these programs.

Applied to two RISC-V processor cores, CV32E40P and Ibex, our methodology synthesizes programs that expose bit-flip faults for approximately 70 % of microarchitectural signals, using two days of computation on 10 parallel cores. For 25 % of the control signals in CV32E40P, we synthesize programs enabling the precise attribution of a bit-flip to a targeted signal. Such programs could facilitate the use of fault injection to deduce the placement of microarchitectural elements and help design more effective countermeasures.

To the best of our knowledge, this work represents the first systematic methodology for building fault characterization programs, marking a significant step beyond empirical approaches.

I. Introduction

Fault injection attacks have been shown to compromise devices ranging from secure elements [8], [11], [29], specialized security-oriented microcontroller (MCU) designed with built-in countermeasures, to commodity MCU [4], [19], which generally lack hardware protection mechanisms, and even smartphones [24], [28]. These attacks exploit physical vulnerabilities in hardware, triggered through techniques such as glitches [7], electromagnetic [13], or laser [25] injections.

On Central Processing Unit (CPU)-based systems, fault injections are often modeled as alterations of program behavior, such as instruction skips or data corruption [3], [15]. These effects are typically studied using specially designed characterization programs that expose the impact of faults on software execution.

However, some fault effects are inherently tied to the microarchitectural features of the device, highlighting the need for a deeper understanding of fault propagation within the microarchitecture. In 2018, Laurent et al. [20] raised up the interest for microarchitectural faults by demonstrating unanticipated effects by faulting specific microarchitectural components. At Register Transfer Level (RTL), they simulated faults on forwarding logic, speculative execution and general-purpose registers. Their results showed that such faults could bypass the load duplication countermeasure by targeting the forwarding microarchitectural signals. This work showed that injecting faults directly into microarchitectural components can produce effects that are not captured by existing Instruction Set Architecture (ISA) level fault models, such as instruction mutations [22].

Research Gap. Understanding faults effects within a silicon chip is inherently challenging. Many prior works [7], [18], [19], [27] have attempted to characterize CPU behavior using manually crafted programs designed to reveal faults while sweeping a broad range of injection parameters. To enable fault analysis, characterization programs are designed as short instruction sequences, isolated from the surrounding code used to initialize registers and harvest the resulting state [18], [19], thereby reducing ambiguity when attributing observed effects to specific faults. However, the global approach adopted by these previous works primarily focused on exposing faults injected in the most sensitive cells of the design, typically those within the memory hierarchy.

In contrast, Alshaer et al. [4] investigated the impact of instruction alignment on observed fault models. Their methodology differs from earlier studies in that it constructs targeted programs specifically designed to expose anomalous behavior related to a particular CPU feature. Since their analysis focused solely on alignment effects, their programs primarily revealed faults associated with the aligner. Although their methodology enabled the identification of faults in this component, it could in principle be applied to other microarchitectural elements. Nonetheless, doing so would require significant manual effort, as the test programs must be carefully crafted and rely on detailed knowledge of the CPU architecture. Synthesizing characterization programs would reduce both the time and the level of expertise required to reveal the various possible fault effects of a CPU architecture.

This work was funded in part by the French National Research Agency (ANR) under the projects ARSENE (ANR-22-PECY-0004), Forward (ANR-22-PTCC-0001, France 2030), LOTR (ANR-23-CE25-0016), and Programme de recherche à risque Audace (ANR-24-RRII-0004), and in the framework of the *Investissements d'Avenir* program (ANR-10-AIRT-05, IRTNanoElec).

Moreover, the behaviors observed by Alshaer et al. [4] cannot be unambiguously attributed to the aligner alone; interactions with other CPU functionalities may also explain the results. The synthesis of characterization programs must include an evaluation of their effectiveness, in particular to determine which fault models are exposed by a given characterization program. Werner et al. [30] introduced several metrics to support this evaluation, such as the Propagation Rate (PR), quantifying the likelihood that a fault propagates to the program outputs, and the Discrimination Rate (DR), measuring a program's ability to exhibit distinct behaviors for different fault models. Their metrics were however used exclusively for ISA-level characterization, independently from the underlying microarchitecture.

Contributions. We make the following contributions:

- We introduce an automated methodology for synthesizing characterization programs aimed at exposing fault models in a CPU. The proposed approach relies on bounded model checking applied to a CPU RTL model under fault to automatically produce programs capable of revealing the influence of any RTL signal within the system.
- We apply our methodology to two open-source RISC-V processor cores: the CV32E40P from the OpenHW Group and Ibex from LowRISC. Across both case studies, we synthesized more than 1 000 test programs, collectively targeting about 350 distinct points of interest. Our approach thus reaches a coverage of 75 % for the CV32E40P and 65 % for the Ibex.
- We quantify an adapted DR metric for 50 % of the points from which characterization programs are synthesized for the CV32E40P and Ibex processor cores. Interestingly, for the CV32E40P, our synthesized programs allow to precisely isolate faults injected in 25 % of points covered.

Organization. This paper is organized as follows. Section II first highlights the main limitations of existing state-of-the-art approaches that our work aims to address. Section III then introduces the basic notation and outlines the general structure of a characterization program. Section IV presents our synthesis methodology, including its formal objectives and implementation details. Section V reports our results over the two processor cores and related work is presented in Section VII. Finally, Section VIII concludes the article.

II. Motivating Example

Khuat et al. [19] investigated the impact of laser fault injection on the 2-stage ARM Cortex-M0+ System-on-Chip (SoC) to characterize effects on the Flash interface and the processor pipeline. The characterization methodology relies on the program shown in Listing 1. Six injection positions were exploited, producing different fault effects depending on injection timing. Using a single characterization program written by hand, [19] demonstrates how to characterize different fault effects. In particular, for two injection positions skips of `i6` instruction in Listing 1 were attributed to the execute and decode stages respectively, based on cross-comparison of observed effects, injection timings, and an analysis of instruction processing in the microarchitecture. This microarchitectural analysis, however, was based on manual analysis, and on educated guesses since the core implementation was not public at the time.

Listing 1: ARM-based characterization program from [19].

```
i1:  add r0, r0, 0x1
i2:  add r0, r0, 0x2
i3:  add r0, r0, 0x3
i4:  add r0, r0, 0x4
i5:  add r1, r1, 0x1
i6:  add r2, r2, 0x1
i7:  add r3, r3, 0x1
i8:  add r4, r4, 0x1
i9:  add r0, r0, 0x5
i10: add r0, r0, 0x6
i11: add r0, r0, 0x7
i12: add r0, r0, 0x8
```

Listing 2: RISC-V port of Listing 1.

```
i1:  addi x1, x1, 0x1
i2:  addi x1, x1, 0x2
i3:  addi x1, x1, 0x3
i4:  addi x1, x1, 0x4
i5:  addi x2, x2, 0x1
i6:  addi x3, x3, 0x1
i7:  addi x4, x4, 0x1
i8:  addi x5, x5, 0x1
i9:  addi x1, x1, 0x5
i10: addi x1, x1, 0x6
i11: addi x1, x1, 0x7
i12: addi x1, x1, 0x8
```

TABLE I: Faulted positions leading to skip of `i6` instruction in Listing 2.

`i6` location	faulted stages	# positions
Prefetch	Prefetch	3
IF	Prefetch, IF, ID	5
ID	IF, ID, EX	16
EX	ID, EX	5

We evaluate fault effects during the execution of the characterization program illustrated Listing 2 over the 4-stage RISC-V CV32E40P core [2]. We use μArchiFI [26], a pre-silicon tool to assess system robustness against fault injection attacks. The faults are applied at the RTL level, in the processors microarchitecture design, using a single bit-flip fault model. The analysis identifies all the possible fault positions and timings leading to a fault effect equivalent to an instruction skip of `i6` instruction, during execution of program in Listing 1. The analysis results, summarized in Table I, reveal 29 exploitable fault positions. Specifically, we observe that 16 distinct injection positions lead to an instruction skip when `i6` is in the ID (decode) stage, and 5 distinct injection positions when `i6` is in the EX (execute) stage. Furthermore, these $16+5$ (21) positions are spread over three pipeline stages: IF (fetch), ID, and EX. Thus, without precise knowledge of the microarchitecture implementation, it cannot be concluded whether a fault is targeting the ID or the EX stages on a CV32E40P core.

Instead of trying to infer the microachitectural root cause from an observed software fault effect, our methodology proposes another approach by leveraging programs that expose the effect of a targeted fault injection into the processor microarchitecture. Each characterization program targets a distinct fault location; consequently, our methodology aims to provide characterization programs by means of automated synthesis from knowledge of the processor design.

For example, our methodology is able to synthesize the

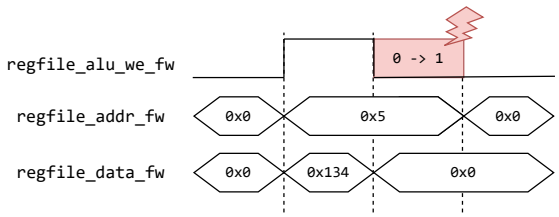


Fig. 1: Chronogram of selected RTL signal transitions in the register file during execution of the synthesized program in Listing 3. The fault, a bit-flip over `regfile_alu_we_fw`, is shown in red.

program shown in Listing 3, exposing a fault injected on the forwarding signal `regfile_alu_we_fw`, which is one of the 16 faults positions from Table I leading to a skip of `i6`. In this listing, the instruction's address is shown before the colon sign. In the reference execution, the RISC-V instruction `jalr x5, x1, 0x25` writes `0x00000134` to register `x5` (`x5 := 0x130+4`). The register file state between the reference and a faulted execution differs only in the value of this register `x5`, indicating that a fault on `regfile_alu_we_fw` is observable through register `x5`. As illustrated in Figure 1, the fault forces data forwarding to remain enabled, causing the Arithmetic Logic Unit (ALU) output (`0x00000000`) to be written into `x5`.

Listing 3: Example of synthesized program when injecting a bit-flip on the RTL signal `regfile_alu_we_fw`.

```
# Address: instruction
    0x128: lw   x9, -0x401(x12)
    0x12C: beq  x1, x9, 0x14
    0x130: jalr x5, x1, 0x25
```

III. Background

This section introduces the modeling of a circuit design under fault injection.

A. Circuit model

An RTL circuit model $\mathcal{C} = (E_c, E_s, W)$ is represented as a set of combinational and sequential elements, resp. denoted E_c and E_s, connected by signals $W \subseteq E \times E$. Let S be the alphabet of circuit states of $\mathcal{C}$. A circuit state $s_i \in S$ is a valuation of the sequential elements E_s in $\mathcal{C}$ at clock cycle i. The program memory defines the part of the circuit state specified by the contents of the program section in the memory design. Let P be the alphabet of circuit programs. We assume that the contents of the program memory are static, such that the circuit program $p \in P$ is a partial value of circuit state s_0.

The execution of a program p in s_0 on circuit $\mathcal{C}$ is captured by function *exec*:

$$exec : \mathcal{C} \times S \to S \quad s_n = exec(\mathcal{C}, s_0)$$

Since p is included in s_0, by abuse of notation we denote program execution by $s_n = exec(\mathcal{C}, p)$ when the value of the initial state is not relevant outside of p.

B. Fault modeling

Let $\mathcal{C} = (E_c, E_s, W)$ be a circuit. A transient fault model for circuit $\mathcal{C}$ is characterized by the tuple $\mathcal{F} = (\alpha, E, I)$.

$$\alpha : \{0,1\}^n \to \{0,1\}^n$$

defines the fault effect applied to a signal of width n, $E \subseteq E_s$ defines the subset of circuit signals that can be faulted (also called fault positions), and $I \subseteq \{\mathbb{N}^+\}$ the set of clock cycles where a fault might be injected.

Evaluating the robustness of a circuit $\mathcal{C}$ to a fault model $\mathcal{F}$ consists in evaluating the consequences of every *fault injection* $f = (\alpha, e, i) \in \mathcal{F}$ over $\mathcal{C}$, $\forall e \in E$, $\forall i \in I$. When the fault injection f is effective over $\mathcal{C}$, it produces a new circuit state s_i^f at clock cycle i, and every subsequent circuit state $s_j^f, j > i$ may differ from the non-faulted circuit states.

By abuse of notation, let us denote $\mathcal{C}_{(\alpha,e,i)}$ the application of fault instance $f = (\alpha, e, i)$ over circuit $\mathcal{C}$. The execution of program p over faulted circuit $\mathcal{C}_{(\alpha,e,i)}$ produces a final faulted state $s^f = exec(\mathcal{C}_{(\alpha,e,i)}, p)$.

Note that security evaluation in the context of fault injection may involve multiple independent injections, possibly using several fault models.

In this work, we restrict ourselves to a single fault injection defined by the fault model $\mathcal{F} = (\alpha, E, I)$, where α is the bit-flip fault model. Let w be a signal of width n, applying a bit-flip on bit position $k \in [0; n-1]$ is defined as:

$$\alpha_k(w) = w \oplus 2^k$$

C. State observation

We are interested in the observation of partial values of circuit states, in particular for the observation of fault effects.

Let $\mathcal{O}_\mathcal{C}$ be the observation of the states of circuit $\mathcal{C}$. An observation monitors circuit states and return values in a domain O:

$$\mathcal{O}_\mathcal{C} : S \to O$$

For example, the observation of the contents of a processor's register file, containing n 32-bit registers, produces a set of 32-bit values:

$$\mathcal{O} : S \to \{\{0,1\}^{32}\} \quad \mathcal{O}(s) = \text{rf}(s)$$

IV. Methodology

We now formalize the design of characterization programs for fault injection, and introduce our methodology for the automated synthesis of such programs. The methodology is composed of two main phases illustrated on Figure 2. At first, RTL modeling prepares the design and injects the specifications that sets up the synthesis. Secondly, the prepared design is given to a solver allowing the synthesis of programs. The next sections detail each step.

A. Problem statement

We target the design of characterization programs suitable for use on a concrete fault-injection bench.

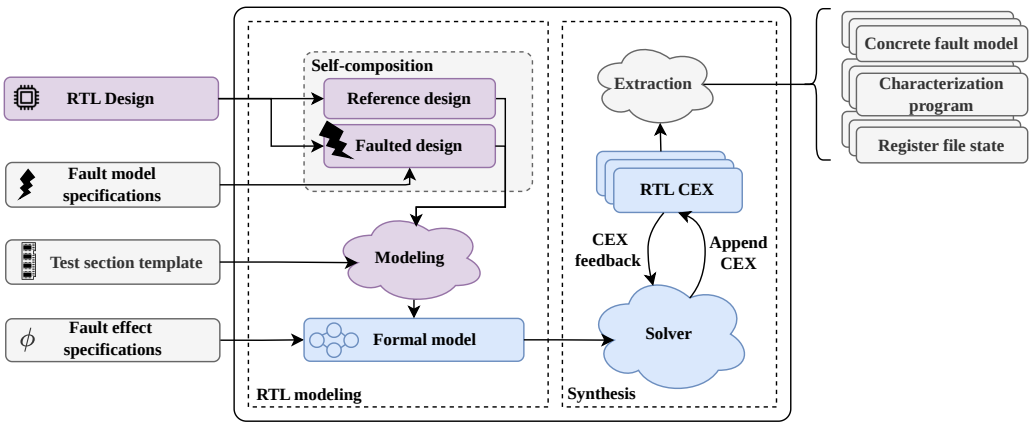


Fig. 2: Methodology overview (RTL modeling in purple, formal modeling in blue, specifications in grey).

1) Observation of fault effects: Without relying on processor instrumentation, such as a debugger, the only way to observe fault effects is through the memory manipulated by programs. Consequently, we assume that the fault injection produces observable effects in the processor's register file.

Let $\mathcal{C}$ be a circuit, $\mathcal{F} = (\alpha, E, I)$ a fault model, and $f = (\alpha, e, i)$ a fault injection in $\mathcal{F}$. Without loss of generality, we assume that the fault effect α is fixed a priori.

Our objective is to synthesize characterization programs p_e. Each program p_e targets the exposure of effects due to a single fault injection (α, e, i) targeting signal e. For each program, the value of injection time i is determined during program synthesis such that $i \in I$. A characterization program p_e exposes a fault injection if it satisfies property $\phi_{(\alpha,e,i)}$:

$$\phi_{(\alpha,e,i)} : \text{rf}(exec(\mathcal{C}, p_e)) \neq \text{rf}(exec(\mathcal{C}_{(\alpha,e,i)}, p_e))$$

We later detail the full synthesis procedure in Section IV-D.

2) Synthesized program termination check: In addition to observability, fault effects in the register file must be captured when synthesized programs have been fully executed. A characterization program thus ends by a set of instructions in charge of recording the state of the register file, thereafter called *dump section* of the characterization program.

Consequently, a characterization program p_e is effective if the fault effects can be observed in its dump section. Assuming that no fault injection is allowed during execution of the dump section, the characterization program is effective if the first instruction of the dump section is executed by the processor. Let $\mathcal{A}_{\text{dump}}$ be the address of the first instruction of the dump section, and Program Counter (PC) the Program Counter of a characterization program p_e, p_e is effective if it satisfies property ϕ_{dump}:

$$\phi_{\text{dump}} : \texttt{PC} = \mathcal{A}_{\text{dump}}$$

3) Fault characterization: A characterization program must satisfy both $\phi_{(\alpha,e,i)}$ and ϕ_{dump}:

$$\phi_{\text{program}} = \phi_{(\alpha,e,i)} \wedge \phi_{\text{dump}}$$

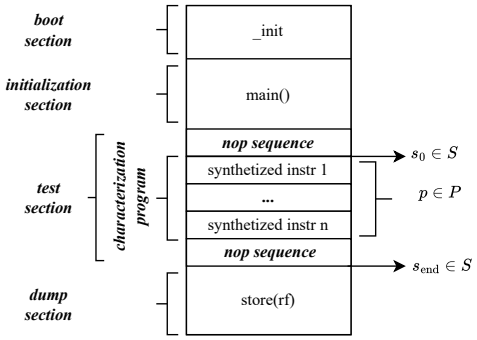


Fig. 3: Characterization program layout.

B. Characterization program layout

Figure 3 illustrates the typical design of a characterization program, organized in four main sections. The boot section starts the processor, initializes and configures peripherals required for data export or for synchronizing the fault injector, and hands over to the `main()` function located in the initialization section.

An important feature is that the characterization program starts from a controlled processor state s_0. In our case, s_0 is fixed and determined by means of simulation. In order to control the value of s_0, the test section starts with a sequence of `nop` instructions, which puts the processor into a deterministic and constant state. The number of instructions in this sequence depends on the length of the pipeline state, such that, when reaching s_0, any machine instruction from the `main()` function has been flushed out from the processor pipeline.

The dump section (Section IV-A2) records the contents of the register file when the execution of characterization program ends. The dump section is located immediately after the test section. The end of the test section is also filled with a sequence of `nop` instructions in order to flush the instructions of the characterization program from the pipeline and to ensure that state s_{end} will not be modified after the execution of first dump instructions.

C. RTL modeling

The modeling phase, illustrated in the left part of Figure 2, prepares the RTL design for transformation into a transition system, a formal representation suitable for analysis by a model checker. We first apply self-composition [9] by duplicating the RTL design. Self-composition enables the comparison of a reference and a faulty circuit state. After duplication, the fault is applied to faulted design. The applied model complies with the fault model specifications.

The *fault model specifications* defines:

- The fault effect α, modeling a single bit flip (Section III-B). When applied to a signal of width $n > 1$, a single bit value is flipped and the targeted bit position is symbolic: it is determined during program synthesis.
- The fault position e, targeting a single signal.
- A time window I.

The *fault effect specifications* are injected into the formal model. The specifications implement the two goals of property ϕ_{program} (Section IV-A):

- they ensure the observability of fault effects in the register file ($\phi_{(\alpha,e,i)}$);
- they ensure the reachability of the dump program after execution of the *test section* (ϕ_{dump}).

Next, a test section template is injected into the RTL model memory and is made symbolic. Setting the program memory as symbolic allows the solver to freely choose values for the test section, enabling the synthesis of program instructions.

Finally, the complete design is converted into a transition system ready for model checking.

D. Program synthesis

Once the modeling is done, the synthesis phase, illustrated in the right part of Figure 2, generates fault characterization programs from formal models of the processor design and of the fault injections.

Our methodology uses Bounded Model Checking (BMC) [12]. But rather than using BMC to prove a system's security property, we leverage the algorithm's efficient state-space exploration to identify states satisfying $\phi_{(\alpha,e,i)}$. The model checking algorithm, also named solver, acts as a Breadth First Search (BFS) algorithm. From an initial state $s_0 \in S$ it explores the state space and assesses $\neg\phi_{\text{program}}$ depth-by-depth. For simplicity, the depth d can be considered as the number of clock cycles analyzed since s_0.

If at a given depth the solver finds a state that breaks $\neg\phi_{\text{program}}$, it returns a counter-example (CEX) tracing the signals that led to the violation of the property. From this CEX, a script extracts a concrete model of the fault injected, the last register file state and the program synthesized. The solver is repeatedly called to produce multiple CEXs. Each new CEX is analysed and translated into additional synthesis constraints to prevent the generation of identical CEX, and in particular to force the solver to generate new programs. This iterative process, by increasing the number of programs exposing the signal e, increases the likelihood of finding a discriminant program that exclusively exposes fault in e.

A maximum depth bound d is defined for the solver's exploration which means that if no CEX is until depth d the property $\neg\phi_{\text{program}}$ will be considered as satisfied up to d cycles and robust to the injected fault model.

However, because the state space of the characterization programs is too vast, we often encounter a state space explosion problem. This leads to cases where we cannot find any CEX due to a timeout, which makes it unable to synthesize a program or conclude that the design is robust to the considered fault model.

To bypass the aforementioned state explosion problem we implement ϕ_{dump} to leverage the solver's optimization capabilities. ϕ_{dump}, when evaluated over two state spaces such $S_{\text{small}} \lll S_{\text{large}}$, must take approximately the same time. Concretely, we identified two candidates signals to implement the property.

- *PC-based*: checks whether the program counter has reached the end of the test section.
- *Cycle-based*: checks that a minimum number of cycles has elapsed during the analysis.

While the PC-based strategy seems more intuitive, the example illustrated in Figure 4 shows that PC-based does not allow the synthesis for programs containing multicycles instructions while the cycle-based strategy can. The property ϕ_{dump} cannot be directly implemented in the solver as it is formulated in Section IV-A2. First, the PC is assessed relatively to the PC at the beginning of synthesis. And secondly we need to put constraints on minimum cycles to be explored by the solver. We therefore define it as follows:

$$\phi_{\text{dump}}^{n,t} = (\text{PC} > \text{PC}_0 + n \times w) \wedge (\text{Cycle} > t),$$

where PC_0 represents the PC value at the start of synthesis, n defines the number of instructions required to consider the program as completed, t defines the minimum number of cycles to be analyzed and w the width in byte of an instruction. In the PC-based strategy $n > t$, otherwise the configuration corresponds to the Cycle-based strategy.

Let consider a fault model impacting the multiplier. Consequently, a fault will be exposed thanks to instructions suite such as Program A $= \{\texttt{mul}, \texttt{addi}, \texttt{addi}\}$. The `mul` instruction needs several cycles to execute which makes Program A execution requiring deeper analysis than Program B, which only contains single cycle instructions. We consider implementations of the property, $\phi_{\text{dump}}^{3,5}$ and $\phi_{\text{dump}}^{3,0}$, associated respectively with the cycle-based and PC-based strategies. Consequently, when we implement $\phi_{\text{dump}} : \phi_{\text{dump}}^{3,0}$, the final state s_{endA} of Program A, due to the `mul` instruction, cannot be reached. This limitation arises because the property $\phi_{\text{dump}}^{3,0}$ can already be satisfied at step 3 by an untractable state space of programs such as Program B, thereby nullifying the effect of the property. However, Program A can be successfully synthesized when $\phi_{\text{dump}} : \phi_{\text{dump}}^{3,5}$ because it forces the solver to unfold more step before assessing $\phi_{(\alpha,e,i)}$. This

example illustrates that with the PC-based strategy, we can only synthesize programs containing single-cycle instructions. In contrast, the cycle-based approach allows us to synthesize both single-cycle and multi-cycle instructions.

V. Validation

Our validation is organized around the following research questions:

- RQ1 Can our synthesis methodology cover all the signals in a processor design?
- RQ2 How long does it take to synthesize programs?
- RQ3 Are the synthesized programs discriminative?
- RQ4 Are synthesis time and coverage sensitive to termination condition parameter t and program size?

A. Experimental Setup

To validate the methodology, we applied it to two RISC-V CPUs: the CV32E40P [2] developed by OpenHW Group and the Ibex [1] core maintained by lowRISC. The CV32E40P features a 4-stage pipeline, whereas the Ibex core has a 3-stage pipeline. Both cores have comparable specifications and implement the IMC extensions of the RISC-V ISA.

All experiments were conducted on a server equipped with 16 Intel Xeon E5-2637 v3 cores (3.50 GHz). Analyses were parallelized across 10 processes. RTL modeling and synthesis used Yosys 0.35; because Yosys's open source frontend does not support SystemVerilog, designs were converted to Verilog with sv2v 0.0.12. BMC employed is Yices2 2.6.5. To simplify control over the encoding of generated instructions and to ease post-analysis of synthesized programs, we added a constraint on a decoder signal of both CPUs to disable the synthesis of compressed instructions. However, without this constraint, the methodology would still be able to synthesize compressed instructions. We also disabled interrupts, along with clock-management features (such as the sleep unit), and excluded their associated signals from the list of targeted fault positions. Please note that these features were not disabled in the design itself, but constraints were added to the design model, to prevent the solver to generate programs that rely on these features. A timeout of 3 hours was applied to the solver to limit the synthesis time of each program.

For the synthesis campaign, all signals with a width of 1 to 2 bits were selected. This choice, while arbitrary, was motivated by our primary goal of generating programs for control signals. As shown in Table II, such signals already account for approximately 55 % of both designs. This selection also reduces the synthesis time, although the methodology can be extended to wider signals, which are beyond the scope of this work.

The solver was configured to synthesize programs capable of revealing $f_{(\alpha,i,e)}$, where α = bit-flip (Section III-B), i ranges from 0 to 10 cycles, and e is the signal to expose. The value e is iterated across all signals within the analysis scope. For each signal, synthesis was limited to 5 programs. 3 memory words were set as symbolic, meaning that with compressed instructions disabled, the test section size for all synthesized programs was fixed at 3 instructions. For the PC-based strategy, the termination condition was $\phi^{7,0}_{\text{dump}}$, allowing for the execution of the test section and the 4 following `nop` instructions to flush the pipeline. For the cycle-based strategy, the termination condition was $\phi^{7,10}_{\text{dump}}$, providing three cycles for synthesized instructions and four cycles for pipeline propagation, plus three additional cycles to support multi-cycle instructions.

TABLE II: Number of signals in $\mathcal{C}$ by width.

	Bit width	0:2	0:16	0:32	all
CV32E40P	# of signals	399	502	665	702
	% of design	56	71	94	100
Ibex	# of signals	475	551	721	786
	% of design	60	70	91	100

TABLE III: Synthesis campaign summary.

Target name	Size	Synthesis time		$\lvert\mathcal{C}[0:2]\rvert$	
	# Cells	$\phi^{7,0}_{\text{dump}}$	$\phi^{7,10}_{\text{dump}}$	*Total*	*Analyzed*
CV32E40P	2498	13h04m10	117h10m14	399	332
Ibex	1749	18h43m01	40h40m31	475	370

Table III summarizes the number of analyzed signals over $\mathcal{C}[0:2]$. We manually discarded 67 signals from the CV32E40P analysis scope and 105 from Ibex analysis scope. All discarded signals were associated with debug features, interrupt inputs, or security features that were disabled during synthesis.

The DR metric (Section V-D) was computed with µArchiFI [26]. The discrimination analysis was performed over all possible signals with widths ranging from 1 to 16 bits, except for the signal targeted by program synthesis. Larger signals were excluded due to combinational loops created by fault instrumentation. For practical reasons, DR was computed on a subset of covered signals. The analysis took 12 days to cover the programs synthesized for 200 signals in CV32E40P, and 9 days to cover the programs synthesized for 137 signals in Ibex.

B. RQ1: coverage

To evaluate how many signals our methodology is able to characterize, we rely on a coverage metric. Let $\mathcal{C}$ be the target circuit, $\mathcal{C}_{\text{found}} \subset \mathcal{C}$ the set of signals for which at least one program was synthesized, $\mathcal{C}_{\text{analyzed}} \subset \mathcal{C}$ the set of all the analyzed design elements, and $\mathcal{C}_{\text{disabled}}$ the signals attributed to disabled features:

$$\mathcal{C}_{\text{analyzed}} = \mathcal{C} - \mathcal{C}_{\text{disabled}}$$

Our coverage metric is defined as follows:

$$\text{coverage\#} = |C_{\text{found}}|$$

Tables IV summarizes the coverage achieved on our test cases by the two strategies. For both cases, the cycle-based strategy achieves higher coverage for almost every module, being twice as high as with the PC-based strategy. However, in CV32E40P's aligner, one additional point is covered by the PC-based approach, which is due to an implementation

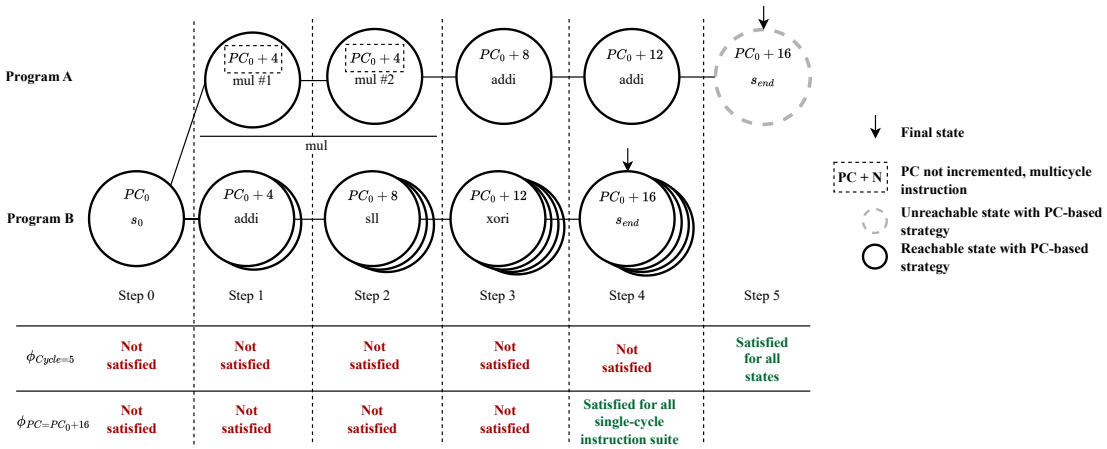


Fig. 4: Reachability of programs depending on the strategy.

TABLE IV: Coverage and size per module for both cases.

Case	Module	Coverage# $\phi^{7,0}_{\text{dump}}$	$\phi^{7,10}_{\text{dump}}$	(Gain)	$\lvert\mathcal{C}_{\text{analyzed}}\rvert$
CV32E40P	**Global**	154	255	**(+101)**	332
	IF Stage	28	49	**(+21)**	58
	IF / Aligner	4	3	**(-1)**	5
	ID Stage	95	139	**(+44)**	186
	EX Stage	10	15	**(+5)**	22
	EX / DIV	0	15	**(+15)**	15
	EX / MULT	2	12	**(+10)**	13
	Load Store	12	16	**(+4)**	22
	CS Registers	3	6	**(+3)**	11
Ibex	**Global**	137	240	**(+103)**	370
	Core Top	21	38	**(+17)**	69
	IF Stage	9	19	**(+10)**	35
	ID Stage	58	121	**(+53)**	168
	EX Stage	8	13	**(+5)**	17
	EX / MulDiv	0	10	**(+10)**	19
	WB Stage	9	14	**(+5)**	14
	Load Store	11	14	**(+3)**	18
	CS Registers	11	11	**(+0)**	30

artifact: the `hardware_loop_pc_set` signal, when faulted, allows the solver to bypass our constraints and resets the PC value. Resetting the PC value breaks the cycle count mechanism of the cycle-based strategy and thus prevents finding a program for this signal.

Because the cycle-based strategy allows for the synthesis of multi-cycle instructions, it significantly improves the coverage for some modules, especially for the multiplier and the divider modules, which are almost exclusively implementing multi-cycle instructions.

To our knowledge, no prior work has attempted to generate programs specifically designed to expose faults injected into microarchitectural control signals. Therefore, our coverage results cannot be directly compared with those of other fault characterization studies. Although we achieved a coverage of 70 %, cases in which no program could be produced are attributable to solver timeouts caused by state-space explosion. Consequently, the absence of a program for a given signal cannot be interpreted as evidence that the register file is insensitive to the considered fault model. The investigation over non-covered points will be part of future work.

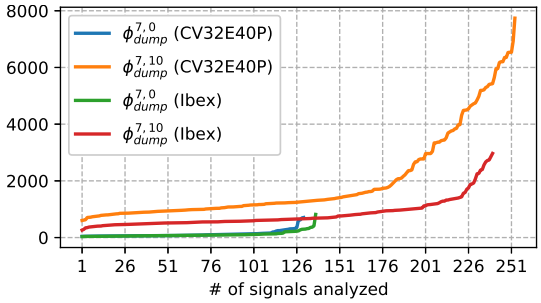


Fig. 5: Average synthesis time (sec.) vs. ordered rank of point.

C. RQ2: synthesis time

Table III summarizes the global results of the synthesis campaign, showing that the cycle-based strategy is much slower than the PC-based strategy. This is also illustrated in Figure 5, which shows the average synthesis time per program for each signal for which at least one point has been found. On average, cycle-based synthesis is approximately ten times as long to synthesize programs on CV32E40P. Synthesis duration is short for most signals with the PC-based strategy for both use cases (Figure 5, green and blue curves). In contrast, synthesis with the cycle-based strategy is systematically longer for both use cases (Figure 5, red and orange curves), with synthesis time being twice as long for CV32E40P wrt. Ibex.

Our automated approach generates such programs in as little as a few minutes to two hours and requires only limited expertise from the evaluator, whereas manual approaches demand a high level of understanding of the target. Additionally, we observe that cycle-based coverage has a substantial impact and tends to increase with design complexity. This finding motivates the combined use of both PC-based and cycle-based strategies to minimize synthesis time while maximizing coverage, which is left for future work.

D. RQ3: discrimination

To evaluate the ability of a program to precisely attribute a fault model to an effect we compute a discrimination rate (DR). Let p denote the program synthesized by the methodology, and let $s^f_{(\alpha,i,e)}$ be the state obtained by $s^f_{(\alpha,i,e)} = \text{exec}(\mathcal{C}_{(\alpha,i,e)}, p)$. The pair $(p, s^f_{(\alpha,i,e)})$ is synthesized in order to identify the fault model $f_{(\alpha,i,e)}$. However, this pair may not uniquely discriminate the fault model $f_{(\alpha,i,e)}$. The DR value counts the number of fault injections on different signals resulting in the same synthesized observable state, as follows:

$$\forall c \in \mathcal{C}, j \in [i - tol; i + tol], \text{DR} = \left| \left\{ s^f_{(\alpha,i,e)} \in S^f_{(\alpha,j,c)} \right\} \right|,$$

where i the fault timing determined during synthesis, and tol a tolerance parameter delimiting a time window around i.

For each signal, between 1 and 5 programs are synthesized, meaning that several DR values can be associated with a signal. Hence we keep the lowest DR value among the synthesized programs. Figure 6 shows histograms of signals associated with their respective DR values for both test cases. We observe that our methodology produces fairly discriminant programs: as shown in Figures 6a and 6b, the first quartile (Q1) and the median are less than 10, with values of 2 and 5, respectively. For the Ibex, for half of studied signals we are able to synthesize a program with DR < 8, close from the results observed on the CV3240P. For 28 signals for CV32E40P and 7 signals for Ibex, we could synthesize programs with DR $= 1$.

Finally, Table VI illustrates that program size and DR appear to vary independently. Considering third-quartile values, synthesized programs of four instructions provide the best trade-off between synthesis time and DR value.

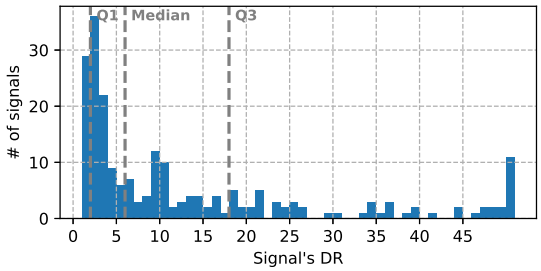
While prior work often relies on manual analysis to associate a faulty state with its root cause, our approach provides a quantifiable metric, based on formal methods, to evaluate a program's DR. Furthermore, signals with DR $= 1$ are uniquely exposed by the proposed methodology, enabling their precise characterization under the assumed fault model. Signals with $1 <$ DR < 10 account for more than half of the analyzed signals. This level of discrimination may enable the design of targeted countermeasures or be exploited to extract information about the circuit implementation through fault injection.

E. RQ4: To what extent are synthesis time and coverage sensitive to termination condition parameter t and program size?

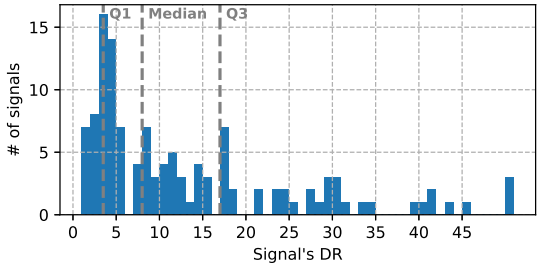
Last, we evaluate the impact of t the number of cycles needed before termination and synthetized programs size fixed during the validation methodology.

1) Impact of cycle-based strategy parameters: The termination condition of the cycle-based strategy is implemented as $\phi^{n,t}_{\text{dump}}$ where $t > n$. To evaluate the impact of parameter t, we ran new synthesis targeting the most timing-sensitive modules of the design, namely the multiplier and the divider, over 28 signals in CV32E40P, and 19 signals in Ibex. All synthesis runs were set with a 1-hour timeout.

Table V presents the coverage achieved for each value of t, illustrating that the value of t directly impacts both achieved coverage and synthesis time. For values below 5 cycles on CV3240P and 6 cycles on Ibex required to execute the targeted operation, program synthesis fails as no program can reach the end of test section in such small numer of cycles. This phenomenon is detailed in Section IV. Conversely, as t increases beyond a certain threshold, synthesis time grows exponentially, ultimately resulting in solver timeouts. Optimal coverage is achieved with $t = 10$ for both use cases, probably because they have similar designs. However, other cores may require different parameter values to maximize coverage.



(a) CV32E40P



(b) Ibex

Fig. 6: Discrimination histograms.

TABLE V: Coverage as function of t, and average synthesis time (sec.) per program, on CV32E40P and Ibex ($\phi^{7,t}_{\text{dump}}$).

	t	≤ 4	5	6	8	10	12	14
CV32E40P	Coverage#	0	16	18	27	27	27	26
	Avg time	-	458	467	623	943	1424	2147
Ibex	Coverage#	0	0	2	9	11	8	10
	Avg time	-	-	128	320	869	1550	1940

2) Impact of program size: Throughout the validation section, the program size l was fixed to 3 instructions. With l the number of instructions in a program we generalize the implementation of ϕ_{dump} as $\phi^{l+4,l+7}_{\text{dump}}$. To assess the impact of this parameter, we conducted a sensitivity study by synthesizing programs ranging from 1 to 8 instructions over 20 randomly selected injection points, illustrated in Table VI.

Our results show that coverage is mostly insensitive to program size. The only exception is related to single-instruction programs, where synthesis fails to cover two signals related to forwarding and branch decoding. The first case requires a preceding instruction to exploit forwarding misbehavior, whereas the second requires at least one branch instruction along with an additional instruction to be skipped. Increasing program size up to 8 instructions does not improve

TABLE VI: Coverage, average synthesis time (sec.) and DR per program as a function of program size, for CV32E40P ($\phi_{\text{dump}}^{l+4,l+7}$). Evaluated over 20 randomly chosen signals of 0 to 2 bits.

l		1	2	3	4	5	6	7	8
Coverage#		8	10	10	10	10	10	10	10
Syn. time		658	888	818	873	1239	1986	1961	2646
DR	Q1	2	2	4	3	4	3	2	3
	med	7	4	9	4	8	7	11	3
	Q3	30	37	26	13	13	22	23	25

coverage, hence supporting our choice of short programs. In contrast, synthesis time increases with program size, as expected due to the enlarged search space.

VI. Discussion

This section discusses the relevance of experimental choices made in the validation section, the duration of a complete synthesis campaign, and our use of a discrimination rate metric.

A. Experimental choices

1) Bit-flip fault model: The proposed methodology aims to bridge the gap between RTL-based simulations and physical fault injection experiments. This requires selecting a fault model that can realistically occur at the silicon level for the synthesis of RTL-based characterization programs. Prior work such as [14] has demonstrated the feasibility of inducing single bit-flips in flip-flops using laser fault injection. For this reason, we adopt the single bit-flip fault model.

Although the fault injection framework used, µArchiFI [26], supports multi-bit fault models, substantial work is required to support multi-bit faults in our methodology. To the best of our knowledge, [23] is the only work supporting multi-bit flip effects at the RTL level, which focuses on identifying reduced sets of sequential elements impacted by laser injections, without considering combinational elements individually. Our methodology considers both combinational and sequential elements and could therefore be used to infer more general models by comparing their effects with those observed in experimental characterization campaigns.

2) Signals size: The validation of the methodology was initially conducted on small-width control signals, which were used to test the effectiveness of the methodology. This choice was motivated by the fact that control signals are typically more challenging to characterize manually than larger data-path signals, making them particularly relevant targets for automated analysis.

However, this restriction is not a limitation of the synthesis in itself. Table VII reports results obtained on signals ranging from 3 to 32 bits for the CV3240P test case. These experiments show that extending the analysis to larger signals yields comparable coverage, with similar synthesis times. However, while the amount of time needed to synthetize programs for such signals seems reasonnable, the current version of DR computation (Section VI-C) would take much more time as it adds up 812 programs to analyze. The substantial time overhead incurred led us to adhere to the initial decision to consider only the $\mathcal{C}[0:2]$ signals during validation (Section V).

TABLE VII: Coverage for CV32E40P ($\phi_{\text{dump}}^{7,10}$), for signals with sizes ranging from 1 to 2, and 3 to 32 bits.

Module	Synthesis time	Coverage#	$\lvert\mathcal{C}_{\text{analyzed}}\rvert$
$\mathcal{C}[0:2]$	117h10m14	255	332
$\mathcal{C}[3:32]$	54h10m07	164	256

3) Register file divergence: The initial evaluation focuses on observing divergences in the register file, in order to remain consistent with practical setup of silicon experiments.

As mentioned in Section IV-D, we do not use the BMC algorithm to prove a system's security property but rather to find CEXs satisfying ϕ_{program}. Hence, extending observability to other parts of the design could simplify the synthesis process by providing additional degrees of freedom to the solver. This can be achieved by introducing relevant properties, similar to $\phi_{(\alpha,e,i)}$, during synthesis.

4) Disabled features: As discussed in the introduction, characterization tests aim to use simple programs that keep the target in a controlled configuration. Moreover, some features may complicate the fault identification process. In this work, we constrained the synthesis process to exclude compressed instructions and interrupt-related behaviors (Section V-A), but we demonstrate that our synthesis methodology can exploit complex modules such as a multiplier.

Enabling compressed instructions would require minimal effort from both the synthesis and DR computation perspectives. However, it would make the interpretation of results more complex due to reduced control over instruction encoding. In contrast, supporting interrupts would require a more substantial extension of the framework, including the definition of appropriate interrupt handling routines and revised termination conditions. Enabling these features could increase the diversity of fault manifestations and potentially improve coverage. A thorough investigation of these aspects is left for future work.

B. Synthesis performance

With the experimental setup used in validation, a full synthesis campaign takes approximately one week using 10 parallel cores for the considered test cases. As the synthesis is performed once per target, this cost remains acceptable, especially compared to manual and empirical approaches. Since the methodology is inherently parallel, substantial speedups can be achieved on modern compute infrastructures commonly available in industrial environments.

However, this non-optimized runtime should be considered a worst-case scenario and can still be significantly be improved before wide adoption. First, we synthesize up to five programs per point to improve coverage, which proportionally increases computation time.

Moreover, the program execution space is too large to be explored exhaustively, so synthesis is performed within a user defined time budget. Once this time budget reached

a timeout happens. This characteristic aligns with our objective of identifying relevant corner cases rather than proving robustness. The time budget directly impacts both runtime and the number of programs discovered, as it sets the solver's timeout, it must therefore be carefully chosen. Performance can be further improved by combining PC-based and cycle-based strategies. A practical approach consists of first applying a PC-based strategy with a short timeout, and falling back to a cycle-based strategy only when necessary.

C. DR computation

The current implementation of the DR computation is intentionally simple and has not yet been optimized. Several improvements can be introduced to enhance scalability. However, the primary objective of this work is to assess the effectiveness of the synthesis methodology rather than to provide a fully optimized DR analysis framework. Designing an efficient and scalable DR computation is therefore left for future work.

Additionally, the current implementation is affected by a combinational loop issue that limits discrimination analysis to signals with widths up to 16 bits. This limitation will be addressed in future versions of the tool. Nevertheless, this limitation can be mitigated by restricting the analysis to sequential signals. In such cases, fault instrumentation does not introduce combinational loops due to the presence of flip-flops.

To validate the robustness of the DR metric, we conducted additional experiments on sequential signals with widths ranging from 1 to 32 bits. These results show that the discrimination of the most effective programs remains consistent across this extended range: over 28 re-analyzed signals, only two of them have their DR incremented by one. However, such sequential signals only represent 25 % of the signals between 17 and 32 bits. Consequently, the combinational loop issue still needs to be addressed to achieve higher coverage.

Moreover, as future work, we identify two approaches to further improve discrimination. The first consists in constraining the solver at synthesis time, while the second relies on using multiple programs and extracting the intersection of the signals exposed.

VII. Related Work

Several articles [4], [19], [27] have studied the link between microarchitectural features and their observed software fault models. However, these approaches rely on empirical methods, manually analyzing case studies to infer the root causes of chip behavior.

Conversely, works such as simpliFI [16] and FaultDetective [21] propose methodologies aimed at attributing observed effects to their underlying causes. SimpliFI, on one side, exploits gate level simulation to observe links between fault effect on program execution and clock glitch injection simulation. On the other side, FaultDetective implements a custom SoC made of six interconnected MSP430 MCUs instrumented with a scan chain to investigate the chain of events that led to an observable fault. While SimpliFI relies solely on simulation-based evaluation, FaultDetective adopts a real-world experimental approach but remains limited to its specifically designed SoC.

Our methodology aims to bridge the gap between the accuracy of characterization-based approaches and the flexibility of RTL-level analysis by synthesizing targeted programs. To the best of our knowledge, this work represents the first attempt to automatically synthesize fault characterization programs capable of isolating and exposing faults in individual RTL signals.

Prior work has also explored program synthesis to generate programs with specific properties. For example, Attias et al. [5] and Blazytko et al. [10] applied synthesis to deobfuscate code by stimulating obfuscated programs with diverse inputs, observing the outputs, and inferring equivalent programs. Program synthesis has also been employed to identify gadget-based exploits [6] or generate loop-free programs [17].

VIII. Conclusion

In this article, we have presented a methodology for synthesizing characterization programs using model checking, specifically designed for CPU RTL models. Our validation demonstrates that the synthesized programs achieve substantial coverage and discrimation of design points. The implementation, validated on two RISC-V CPUs, can be adapted to additional CPU models with an effort comparable to that required to integrate a new CPU into a standard synthesis flow. Overall, our approach provides a scalable and automated framework for CPU characterization, reducing manual effort and paving the way for more rigorous and efficient verification and evaluation flows. Furthermore, the methodology opens several opportunities and constitutes a first step towards exposing subtle microarchitectural faults. In a near future, the methodology may also help for reverse-engineering approaches.

As future work, the next step is to use our synthesized programs in a characterization campaign on an ASIC to fully assess the practical impact of the proposed methodology. The objective of this work is to establish and evaluate the methodology at the RTL level which is required before bridging the gap with a silicon validation. The corresponding physical characterization campaign will require a dedicated experimental setup and a comprehensive analysis of physical injection modalities, which are beyond the scope of this article. New fault models could also be inferred from unanticipated effects observed during these campaign. Additionally, a cone-of-influence analysis could automatically eliminate logic that has no impact on signals involved in fault effect specifications, thereby improving our coverage metric. We also aim to extend our approach to other fault models, such as those induced by clock glitches or electromagnetic injection, and to other circuits such as cryptographic circuits or GPUs. However this generalization would need an adapted formalization as programs would then become sequence of inputs rather than instructions written in memory. Finally, we plan to enhance the discrimination capability of our methodology by directly integrating this metric into the program synthesis process.

References

[1] GitHub - lowRISC/ibex: Ibex is a small 32 bit RISC-V CPU core, previously known as zero-riscy. https://github.com/lowRISC/ibex.

[2] GitHub - openhwgroup/cv32e40p: CV32E40P is an in-order 4-stage RISC-V RV32IMFCXpulp CPU based on RI5CY from PULP-Platform. https://github.com/openhwgroup/cv32e40p.

[3] Ihab Alshaer, Ahmed Al-Kaf, Valentin Egloff, and Vincent Beroulle. Inferred Fault Models for RISC-V and Arm: A Comparative Study. In *37th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, Oxfordshire, United Kingdom, October 2024.

[4] Ihab Alshaer, Brice Colombier, Christophe Deleuze, Vincent Beroulle, and Paolo Maistri. Variable-length instruction set: Feature or bug? In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 464–471, 2022.

[5] Vidal Attias, Nicolas Bellec, Grégoire Menguy, Sébastien Bardin, Jean-Yves Marion, List Université Paris-Saclay, CEA, and LORIA Université de Lorraine, CNRS. Augmenting search-based program synthesis with local inference rules to improve black-box deobfuscation. *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*, page 15, 2025.

[6] Nicolas Bailluet, Isabelle Puaut, Emmanuel Fleury, and Erven Rohou. Nothing is unreachable: automated synthesis of robust code-reuse gadget chains for arbitrary exploitation primitives. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA, 2025. USENIX Association.

[7] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 105–114, 2011.

[8] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application. 9th IFIP WG 8.8/11.2 International Conference*, volume 6035 of *Lecture Notes in Computer Science / Security & Cryptology*, pages 148–163, Passau, Germany, April 2010. Springer.

[9] G. Barthe, P.R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 100–114, 2004.

[10] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 643–659, Vancouver, BC, August 2017. USENIX Association.

[11] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. In Emmanuel Prouff, editor, *Proceedings of the 10th International Conference on Smart Card Research and Advanced Applications (CARDIS)*, volume 7079 of *Lecture Notes in Computer Science*, pages 283–296. Springer, 9 2011.

[12] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer International Publishing, Cham, 2018.

[13] Mathieu Dumont, Mathieu Lisart, and Philippe Maurine. Electromagnetic fault injection : How faults occur. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 9–16, 2019.

[14] Jean-Max Dutertre, Vincent Beroulle, Philippe Candelier, Stephan De Castro, Louis-Barthelemy Faber, Marie-Lise Flottes, Philippe Gendrier, David Hély, Regis Leveugle, Paolo Maistri, Giorgio Di Natale, Athanasios Papadimitriou, and Bruno Rouzeyre. Laser Fault Injection at the CMOS 28 nm Technology Node: an Analysis of the Fault Model. In *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–6, 2018.

[15] Jean-Max Dutertre, Timothé Riom, Olivier Potin, and Jean-Baptiste Rigaud. Experimental Analysis of the Laser-Induced Instruction Skip Fault Model. In *The 24th Nordic Conference on Secure IT Systems, Nordsec 2019*, pages 221–237, Aalborg, Denmark, November 2019.

[16] Jacob Grycel and Patrick Schaumont. SimpliFI: Hardware Simulation of Embedded Software Fault Attacks. *Cryptography*, 5(2), 2021.

[17] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 62–73, New York, NY, USA, 2011. Association for Computing Machinery.

[18] Martin Kelly, Keith Mayes, and John F. Walker. Characterising a CPU fault attack model via run-time data analysis. pages 1–6, 5 2017. IEEE International Symposium on Hardware Oriented Security and Trust (HOST).

[19] Vanthanh Khuat, Jean-Luc Danger, and Jean-Max Dutertre. Laser fault injection in a 32-bit microcontroller: from the flash interface to the execution pipeline. In *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 74–85, 2021.

[20] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. On the importance of analysing microarchitecture for accurate software fault models. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 561–564, 2018.

[21] Zhenyuan Liu, Dillibabu Shanmugam, and Patrick Schaumont. FaultDetective: Explainable to a Fault, from the Design Layout to the Software. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):610–632, Sep. 2024.

[22] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88, 2013.

[23] Athanasios Papadimitriou, David Hély, Vincent Beroulle, Paolo Maistri, and Régis Leveugle. Analysis of laser-induced errors: RTL fault models versus layout locality characteristics. *Microprocess. Microsystems*, 47:64–73, 2016.

[24] Carlton Shepherd, Konstantinos Markantonakis, Nico van Heijningen, Driss Aboulkassimi, Clément Gaine, Thibaut Heckmann, and David Naccache. Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis. *Computers & Security*, 111, 2021.

[25] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems (CHES)*, pages 2–12, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[26] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. μArchiFI: Formal Modeling and Verification Strategies for Mmicroarchitetural Fault Injections. In *FMCAD. 23-Formal Methods in Computer-Aided Design 2023*, pages 101–109. TU Wien Academic Press, 2023.

[27] Thomas Trouchkine, Guillaume Bouffard, and Jessy Clédière. Fault Injection Characterization on Modern CPUs: From the ISA to the Micro-Architecture. In *Information Security Theory and Practice: 13th IFIP WG 11.2 International Conference, WISTP 2019, Paris, France, December 11–12, 2019, Proceedings*, page 123–138, Berlin, Heidelberg, 2019. Springer-Verlag.

[28] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adèle Morisset, and Sébastien Ermeneux. Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot-Extended Version. *IEEE Transactions on Computers*, 69(10):1449–1459, 2020.

[29] Eric Vetillard and Anthony Ferrari. Combined Attacks and Countermeasures. In Dieter Gollmann; Jean-Louis Lanet; Julien Iguchi-Cartigny, editor, *Lecture Notes in Computer Science*, volume LNCS-6035 of *Smart Card Research and Advanced Application*, pages 133–147, Passau, Germany, April 2010. Springer.

[30] Vincent Werner, Laurent Maingault, and Marie-Laure Potet. Toward Better Fault Characterization Tests. HAL ePrint Archive, September 2022. https://hal.science/hal-03790625.