Exploration of fault effects on formal RISC-V microarchitecture models

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

Exploration of Fault Effects on Formal RISC-V Microarchitecture Models

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
‡Sorbonne Univ., CNRS, LIP6, F-75005, Paris, France – firstname.lastname@lip6.fr

***Abstract*—This paper introduces a formal workflow for modeling software/hardware systems in order to explore the effects of fault injections and evaluate the robustness to fault injection attacks. We illustrate this workflow on four versions of a PIN authentication code, embedding different software countermeasures. The code is symbolically evaluated on two implementations of the RISC-V CV32E40P core: the original implementation from the OpenHW group and an implementation that integrates protection of the pipeline control signals. On the original, unprotected core, our formal workflow exposes various vulnerabilities, including previously unknown ones, whereas, on the protected core, it confirms the effectiveness of the proposed countermeasures.**

***Keywords*-Secure Embedded Systems; Fault Attack; Microarchitecture; Formal Verification**

I. Introduction

Fault Injection (FI) attack is a powerful threat against embedded systems [1]. Many fault injection techniques, such as clock or voltage glitches [2], [3], electromagnetic radiations [4], or laser pulses [5], can be used to physically disturb the circuit and induce incorrect values in the hardware. The fault then propagates at the microarchitecture level and emerges at the software level when incorrect instructions are executed or wrong data is manipulated. These faults can be exploited, for example, to retrieve sensitive data or to bypass security mechanisms.

For more than ten years, a lot of work has been addressing the characterization of fault effects given specific equipment and a given target processor [3], [4], [6]–[9]. Such characterization is conducted following a black-box or grey-box approach as proper documentation is not publicly available. Moreover, only limited information can be retrieved after a fault injection, typically the content of the general purpose registers and memory. Some specifically designed codes are attacked to help the analysis of fault effects. These effects are eventually expressed at the Instruction Set Architecture (ISA) level and thus known as *ISA-level fault models*. These comprise instruction or operand corruption [4], [6], [8], instruction skip [3], [4], [9], test inversion, instruction replay [3], [10], etc.

The ISA-level fault model is convenient for designing software protections or performing vulnerability analyses because it effectively abstracts the target hardware and still encompasses a broad set of faulty behaviors. Many fault effects can often be observed for a given processor target by varying the test code. However, some other effects escape this ISA-level modeling and thus remain unexplained. As an example, Proy et al. [11] classified as *magic edges* some effects observed on a real use case but undetected in their fault characterization based on specific test codes. Some faults inside the microarchitecture can not be explained at the ISA level, as shown by Laurent et al. [12]. For example, corrupting the write signal of the register file prevents the result of a calculation from being written back. This may result in an instruction skip but also in a more complex effect, as the correct computed value is available and forwarded to the following dependent instructions in the pipeline. This illustrates that fault effects depend on the program and cannot be explained without the knowledge of the microarchitecture implementation, e.g., forwarding mechanisms vary with pipeline depth and implementation choices. For security evaluation purposes, there is thus a need for vulnerability analyses that consider both software and hardware (i.e., microarchitecture) in order to analyze the fault effects. Processor source code is now available through open hardware initiatives making such white-box vulnerability analyses possible.

Vulnerability analyses can be performed using simulation tools. Nevertheless, due to the extensive domain of the data input and possible faults, simulation requires techniques for pruning the search space to eventually find a vulnerability. Formal verification (e.g., model checking) is advantageous because it allows abstracting the input data or the fault value by using symbolic evaluation to cover a wider range of analyses. Besides, the verification process is guided towards identifying counterexamples (i.e., vulnerabilities), whereas simulation requires an exhaustive search through all possibilities. Thus, we argue that formal verification provides a strong foundation to systematically explore hardware/software systems in order to comprehensively characterize faults. While some formal approaches at binary code or ISA level have been proposed [13]–[15], to the best of our knowledge, there exists no formal approach for studying the fault injection effects considering both software and hardware.

This paper presents a formal verification-based workflow that combines software and hardware models to explore fault effects and evaluate the system's robustness to FIs. Our workflow starts from a binary program (an .ELF file), a Register Transfer Level (RTL) implementation of a processor core and memory, and configuration files to specify the fault injection settings and the security property of interest. Whenever a fault vulnerability is identified, the workflow generates a counterexample as a Value Change Dump (VCD) file, presenting the processor signals and the corresponding hardware-level trace. We also propose a comprehensive illustration of this workflow, using protected and unprotected software and hardware implementations. On the software side, we consider several versions of a PIN authentication code [16] augmented with different software countermeasures. On the hardware side, we consider two versions of the CV32E40P RISC-V processor from the OpenHW group, the original CV32E40P [17] and a protected variant [18]. First, we study the robustness of the different PIN code versions running on the original CV32E40P. We analyze their vulnerabilities and report new fault effects to the best of our knowledge. We also verify that the protected CV32E40P core, i.e., hardened against microarchitectural faults, protects the system effectively.

This paper is organized as follows. Section II first briefly introduces formal verification at RTL level and associated tools, then describes related work analyzing software and hardware parts of systems jointly. Section III introduces our formal verification-based workflow. Section IV presents some case studies, and Section V reports experimental results, focusing on new fault effects and on the evaluation of the system's robustness. The conclusions are in Section VI.

II. Background and Related Work

Formal verification of hardware and software systems covers both deductive (e.g., theorem proving) and algorithmic approaches (e.g., model checking). The verification process is driven by logical inferences in the former case, and by checking patterns specified by logic formulas in finite state graphs in the latter case. Without being exhaustive, we focus next on hardware verification techniques and applications. Deductive hardware verification addresses a wide range of issues: formalization of HDLs like Bluespec [19], functional verification of micro-architectures like pipelines [20] or full-fledged verification frameworks like Kami [21]. Model checking for hardware systems, powered by SAT or SMT solving, also targets similar applications, supported by a number of model checkers: ABC [22] for bit-level verification, CoSA [23] to assist in hardware design or AVR [24] for word-level reasoning.

Several tools, e.g., Cadence SMV [25] and EBMC [26] automatically generate a formal model from a description of a hardware circuit. Cadence SMV relies on compositional verification techniques (path splitting, symmetry reduction, refinement etc.) to verify large RTL designs. EBMC is also applied to hardware design verification as it supports a fragment of the assertion language of SystemVerilog to specify properties and Bounded Model Checking (BMC) to verify these properties. The open-source synthesis tool Yosys [27] also produces an abstract representation of a hardware design to generate a formal model in SMT or NuSMV [28] formats. Through Yosys, the model checkers CoSA and AVR can also handle RTL-level designs, providing access to complementary techniques to BMC (e.g., interpolation, data abstraction etc.).

Some works [29], [30] propose formal frameworks to verify the effect of faults on a hardware circuit, but these are restricted to cryptographic circuits. To the best of our knowledge, there is no formal verification approach to address faults and their effects by considering the processor and the program running on it. Meanwhile, other works have analyzed the effect of faults at a higher level by abstracting from the hardware implementation. They address the ISA level [14], [31] or the LLVM-IR level [32]. No work brings together both software and hardware in the same formal analysis.

However, recent work has raised the need to consider microarchitecture details when exploring the effect of fault injections. Yuce et al. [33] illustrate that strictly software analysis and countermeasure are sometimes inadequate and do not consider the diversity of hardware implementations, such as pipeline size. Laurent et al. [12] also reported the importance of considering both the software and the hardware in the analysis. They reported new vulnerabilities in the microarchitecture [34] and went further by proposing a methodology to bridge the gap between hardware and software considerations [35]. However, it highlighted the need for a methodology to automate the analysis of the effect of faults on joint SW and HW systems. Some works have addressed this question. Zhang et al. [36] identify bugs in processors' hardware implementations and reconstruct the instruction sequence that leads to bugs using a backward symbolic execution but have not directly addressed the problem of transient fault injection in hardware and their effects on software security.

Even if there is a great diversity of techniques to study software or hardware systems, to our knowledge, no work has yet considered exploring and analyzing fault injection effects with formal techniques on processors comprising both the software and the hardware.

III. Workflow

We combine software and hardware descriptions into a single verification process to study the effects of FIs on the microarchitecture of processors and the security issues involved. Our workflow extends hardware models, generated from an RTL design of processors and memories, with microarchitectural fault models. On the software side, a vulnerability property is expressed over a software representation of binary programs. Finally, we apply a BMC verification process over the resulting software/hardware model. Figure 1 illustrates this workflow and is detailed in the remainder of this section.

A. Hardware Modeling

Our workflow requires a hardware description at the RTL level of both a processor and a memory, denoted respectively CPU and RAM in Figure 1. Since our analysis currently focuses on the processor microarchitecture, we exploit a simplified memory model. The memory allows the proper operation of the processor and is not analyzed against fault injection attacks. We rely on Yosys to automatically translate both the CPU and the RAM, described in the Verilog language (IEEE 1364-2005), to a formal model in the Satisfiability

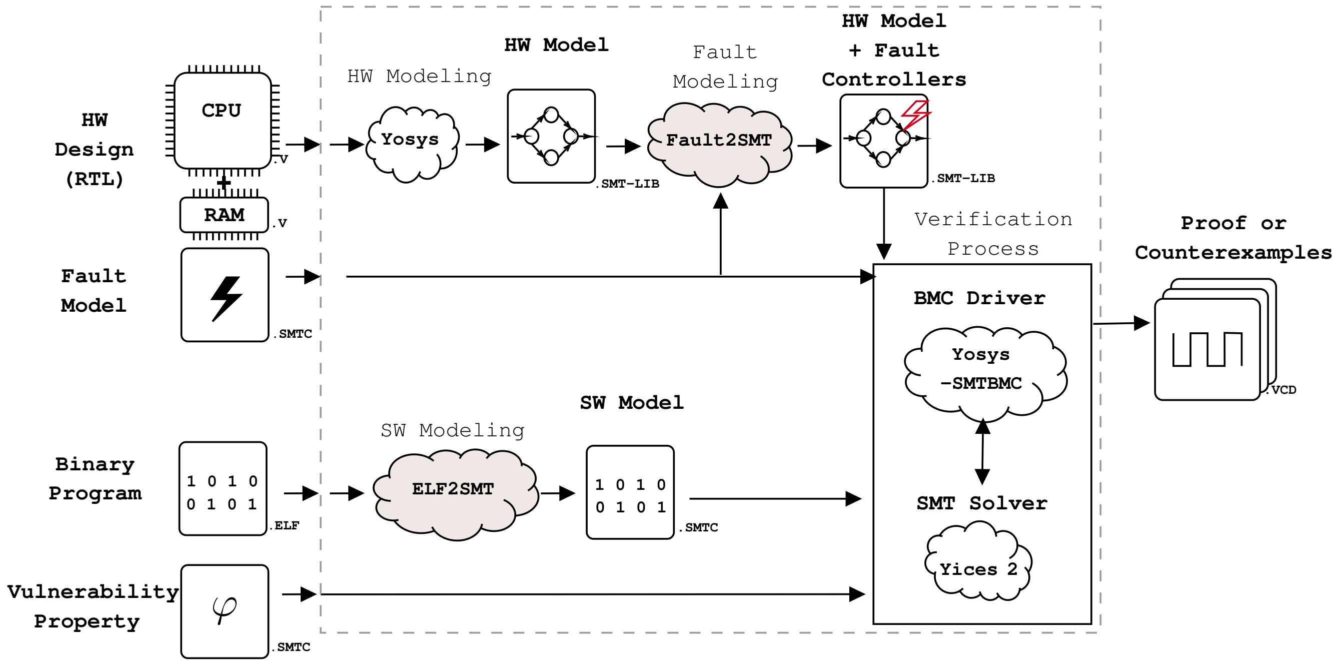

Fig. 1: Illustration of the proposed workflow. Bounded Model Checking (BMC) checks a vulnerability property on a model composed of hardware, software, and fault models.

Modulo Theory Language [37] (SMT-LIB). The SMT backend of Yosys comes with the SMTC language, derived from SMT-LIB, which we use to express properties at each step of the BMC.

The generated SMT model (HW Model in Figure 1) represents both the combinatorial and sequential logics of the CPU and RAM at the RTL level. The SMT model is later manipulated as a transition system during model checking (BMC driver in Figure 1), where each synchronized transition (i.e., a depth in the BMC) corresponds to the update of the sequential logic in the hardware design. To describe the SMT model, Yosys uses the `QF_AUFBV` theory which stands for quantifier-free formulas with data structures adapted to numerical systems such as arrays, uninterpreted functions, and bitvectors. Moreover, Yosys preserves the complete correspondence between the RTL signal naming and the SMT variables. In the remainder of this paper, we thus refer to SMT variables related to the hardware model by their corresponding RTL signal names.

B. Fault Modeling

The workflow modifies the SMT model generated by Yosys to model fault injection effects in the processor. This modeling is a two steps process. First, the SMT model of the CPU is instrumented with fault controllers on any variable (corresponding to RTL signals) specified in the fault model configuration file (Fault2SMT box and Fault Model input in Figure 1). Second, the verification process (BMC Driver) drives the fault controllers according to the constraints expressed in the SMTC language.

The workflow relies on a fault configuration file (Fault Model input in Figure 1) that implements fault models according to four characteristics:

i) the timing constraints of fault injections, expressed as clock cycles;
ii) the hardware elements (SMT variables referring to RTL signals) targeted by fault injection;
iii) the effect of fault injections (e.g., bit-set, bit-reset, bit-flip, or random);
iv) the number of fault injections allowed during a verification.

Listing 1: Example of SMTC syntax for bit-random fault injections on the RTL signal `fw_mux`.

```
1 state 10:90
2 assume (= [fw_mux_fault] bit-random)
3 assume (= [fw_mux_cnt] 1)
```

As an example, Listing 1 describes a bit-random fault model applied on the RTL signal `fw_mux`. From cycle 10 to cycle 90 (line 1), the bits of the `fw_mux_fault` signal can take any value (line 2). The constraint at line 3 specifies that at most one fault injection is allowed during this time interval using a dedicated counter named `fw_mux_cnt`.

Figure 2 represents a logical view of a fault controller. It targets the signal `sig`, e.g., `fw_mux` in Listing 1. The signal `sig_sel` indicates if a faulty value is outputted instead of the golden value. Signals `sig_fault` and `sig_cnt` respectively control the faulty value and the number of injected faults for this fault controller. Signal `sig_fault` can be left symbolic to explore all the possible faulty values and all the

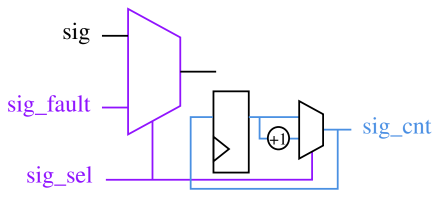

Fig. 2: Logical representation of a fault controller for signal `sig` added in the SMT model of the CPU.

Listing 2: Example of SMTC constraints automatically generated from an ELF file to describe the analyzed input program.

```
state 0
assume (= (select [ram] #x00) #x97)
assume (= (select [ram] #x01) #x11)
assume (= (select [ram] #x02) #x00)
assume (= (select [ram] #x03) #xB2)
...
```

resulting execution paths simultaneously. Besides, when the signal `sig_sel` is not constrained, it is possible to inject a fault at any cycle. This makes possible the exploration of multiple fault injections wrt. to the maximum number of faults allowed (expressed with a constraint on `sig_cnt`). It is also possible to set a maximum number of fault injections during each cycle or the whole execution, whatever their spatial or temporal locations, using constraints on the sum of all `sig_cnt`.

C. Software Modeling

The workflow adds a representation of the input program into the part related to the RAM in the SMT models generated by Yosys. The input program is provided as a binary program (i.e., ELF) compiled for the target architecture. The binary instructions and their associated data are extracted (ELF2SMT in Figure 1) and translated into constraints on the SMT model of the RAM using the SMTC language. Typically, the generated SMT model of the RAM is a single array sized to the input program (both instructions and data).

Listing 2 illustrates an SMTC file automatically generated from an example input program. In this file, the constraints apply to the initial state of the SMT model of RAM (line 1). Lines 2 and following describe the binary encodings of the program instructions and data with `assume` SMT statements. When not constrained in the initial state, the variables in the software part of the formal model are symbolic. These variables can then take any possible value allowed by their data type, which allows the verification to cover all possible execution paths.

Finally, a specific program execution context can also be restored by constraining the initial state of the processor's register file, the PC value, and the data memory. Such execution contexts can be retrieved from simulation, for instance. This possibility is useful to focus the verification only on a specific sequence of instructions within a program.

Listing 3: Example of a vulnerability property φ expressed in SMTC. The SMT solver will check $\neg\varphi$ from step 20 to 45.

```
1 state 20:45
2 assume (not (and
3     (= [error] True) (= [data_valid] True)))
```

D. Vulnerability Property and Verification Process

The vulnerability property φ given to the workflow directly drives the BMC verification process. Our workflow currently supports what is called a safety property in model checking, i.e., expressed as an invariant on a set of states without any temporal operators over the variables of the formal model.

Listing 3 illustrates such a vulnerability property expressed using Yosys SMTC assertions. Lines 2 and 3 describe, with the keyword `assert`, the property $\neg\varphi$ that prohibits to have in a hardware state `error` and `data_valid` equal to `True` at the same time. Line 1 specified at which steps of the BMC the property $\neg\varphi$ must be checked. For instance, `state 20:45` means all steps between 20 and 45, but it is also possible to specify all steps with `always` or a set of specific steps.

The Yosys-SMTBMC script (within the BMC driver box in Figure 1) implements BMC techniques to verify the vulnerability property over the SMT model combining the hardware (i.e., CPU, RAM, and fault controllers) and the software (i.e., constraints over the RAM). The generated SMT model is thus unrolled *depth* times to update the sequential logic of the hardware design modeled in the transition system. After each synchronized transition on the model, the Yosys-SMTBMC script interacts with the SMT model checker to perform a new verification step. Furthermore, it allows inserting or removing any SMTC-based constraints at each verification step. This is how we change RTL signals subjected to FIs to their targeted values at the specified hardware clock cycles according to the fault configuration file, described in section III-B. The SMT model checker generates a counterexample whenever the vulnerability property φ turns to be satisfiable. Otherwise, the verification ends by indicating UNSAT. The depth of the BMC, i.e., *depth*, is derived from the length of the instruction sequence we verify. The SMT solver we use is Yices 2 [38] as it provides the best performance on the BitVector logic employed[1]. Note that the first five steps of the BMC are used to ensure that a hardware reset sufficiently propagates into the model of the processor.

The Yosys-SMTBMC script interprets the output of the SMT solver. It produces a VCD trace containing information about the injected fault and the successive states of the processor when a vulnerability is identified. The so-produced counterexample has two purposes. First, it accurately identifies the fault injection that satisfies the vulnerability property φ among all the fault models explored by the formal analysis. Similarly, it allows to single out the data input of the program if they are symbolic. Second, since the correspondence

[1]https://smt-comp.github.io/2020/results/qf-aufbv-incremental

between the SMT variables of a model and the RTL signals is preserved by Yosys, the VCD trace permits to accurately investigate how the fault propagates in the microarchitecture from injection to manifestation. However, understanding the fault effect and its propagation in the microarchitecture must be done manually. To facilitate this, we perform the difference between the reference VCD trace without fault and the counterexample. The following sections detail some of these vulnerability analyses.

IV. Experimental Setup

This section illustrates the use of our workflow on two case studies based on a software authentication procedure called VerifyPIN and on the CV32E40P processor. We first introduce the four different versions of VerifyPIN used. These versions are hardened with software protections inserted at the source code level, but such protections fail to catch all the possible sources of vulnerabilities leveraged by faults in the microarchitecture. Then, we present the baseline implementation of the processor, then a hardened version that ensures the integrity of control signals. Finally, we present the verification strategy. Experimental results are discussed in Section V.

A. VerifyPIN

A critical piece of software that needs to be robust to fault injections is the `memcmp`-like mechanism used in authentication or signature verification, e.g., in a secure boot process. As practical examples, we consider VerifyPIN programs from the FISSC benchmark suite [16], which provides ten implementations in C, thereafter named VerifyPIN_vN, embedding several software countermeasures of various kinds. VerifyPIN programs compare two PIN codes stored in memory: a user and a secret (card) code, allowing user authentication when the codes are identical.

We selected the versions VerifyPIN_v3 and VerifyPIN_v7 (Listing 4). VerifyPIN_v3 implements the first four countermeasures detailed below, while VerifyPIN_v7 implements all of them.

1) Booleans are hardened by using non-trivial True and False values whose Hamming distance is maximal.
2) The comparison loop has a fixed number of iteration (lines 18-20 in Listing 4).
3) The loop counter is checked at the loop exit against the expected number of iterations (in Listing 4 at line 22).
4) The call to the PIN code comparison function is inlined to protect against faults leveraging call skips (otherwise, the main comparison loop at lines 18-20 in Listing 4 is in this function).
5) Critical tests are duplicated to prevent instruction skip or conditional branch inversion (line 25 in Listing 4).

Listing 4: C implementation of VerifyPIN_v7. VerifyPIN_v3 does not include lines 25 and 29.

```
1  signed char ptc;
2  Bool authenticated;
3  Bool countermeasure;
4  unsigned char userPin[PIN_SIZE];
5  unsigned char cardPin[PIN_SIZE];
6
7  void countermeasure() { countermeasure = 1; }
8
9  void verifyPIN_v7() {
10   int i;
11   Bool diff;
12   authenticated = False;
13
14   if(ptc >= 0) {
15     ptc--;
16     diff = False;
17
18     for(i = 0; i < PIN_SIZE; i++)
19       if(userPin[i] != cardPin[i])
20         diff = True;
21
22     if(i != PIN_SIZE) countermeasure();
23
24     if (diff == False)
25       if(False == diff) {       // Not in v3
26         ptc = 3;
27         authenticated = True;
28         return; }
29       else countermeasure();    // Not in v3
30     else return;
31   }
32   return;
33 }
```

To produce the binary code provided to our workflow, we compiled the selected VerifyPIN versions for the RV32IM ISA using gcc with two different optimization levels. The compilation flag `Og` produces a machine code with a limited number of optimization passes so that the binary code is close to the C code, while the `Os` optimization level uses more aggressive optimizations and focuses on code size reduction. We use these two optimization levels for each selected VerifyPIN version to produce binary programs with different structures and instruction mixes. This allows us to analyze more patterns from a single input program and highlight the impact of the binary code on the presence of subtle vulnerabilities. Both selected optimization levels can remove or alter the implemented countermeasures. We manually added them to the final assembly file when necessary to force their presence at the binary level. In the remainder, we denote the four resulting binaries as follows: `v3_Og`, `v3_Os`, `v7_Og`, `v7_Os`.

In all the experiments, we consider that the userPIN and the cardPIN are different in each of their digits (1). In other words, changing only one of the digits does not allow authentication.

$$\forall i \in [\![0, \mathrm{PIN\_SIZE}-1]\!], \mathrm{userPIN}[i] \neq \mathrm{cardPIN}[i] \qquad (1)$$

Besides, both userPIN and cardPIN are represented as symbolic variables in the SMT model. The oracle that identifies a successful attack is encoded in the property ϕ_0 (2): authentication without triggering the countermeasure function is possible.

$$\phi_0 := (\texttt{authenticated} = \text{True}) \\ \wedge (\texttt{countermeasure} \neq 1) \tag{2}$$

We therefore check the satisfiability of $\varphi := \phi_0$ (2) under fault injections until the VerifyPIN function terminates, i.e., until the calling function context is restored. When satisfiable, we seek to understand the fault effects of the found vulnerability.

B. Baseline CV32E40P

The CV32E40P is a small and efficient, 32-bit, in-order RISC-V core from the OpenHW group designed for light-embedded use. It implements the RV32IMC ISA with a 4-stage pipeline (IF, ID, EX, WB) (Figure 3). The CV32E40P is implemented in SystemVerilog, and the sources are available in [17]. Hence, we use the tool sv2v [39] to convert its implementation to Verilog so that our workflow can process it. The baseline implementation of the CV32E40P does not include protections against faults.

In the following, we introduce three implementation features of the CV32E40P processor, which, as detailed later in Section V, are vectors of vulnerability during FI attacks.

1) Forwarding: Forwarding is a microarchitectural optimization designed to avoid processor stalls due to data hazards. Forwarding bypasses writing the result of an operation to the register file to provide it to the previous pipeline stages as soon as it is available. The implementation is subject to many factors, such as pipeline depth and location of the bypasses that retrieve the available data. In the CV32E40P, the forwarding mechanism shortcuts data dependencies in the ID stage (Figure 3, dashed lines). Any result from functional units (e.g., ALU or MULT) or the LSU can then be used in the ID stage without passing through the register file.

2) Multiplier (MULT): The MULT unit in the EX stage, visible in Figures 3 and 4, performs the multiplication between two 32-bit integers and stores the result in the register file. This module contains two memory elements: a 3-bit value corresponding to its finite state machine (FSM) and a single-bit carry. The duration of the computation depends on the requested operation. Only one cycle is sufficient to produce the 32 least significant bits of the results, whereas five cycles are needed to produce the 32 most significant bits. In this latter case, two signals indicate to the rest of the processor that a multi-cycle multiplication is in progress.

1) The signal `multicycle` is sent to the ID stage to determine if the intermediate result, computed over iterations, needs to be returned to the MULT via the signal `OpC`. As a result, the ID stage stops decoding new instructions.
2) The signal `mulh_ready` indicates if a multicycle multiplication is in progress or not. As a result, the EX stage is tagged as busy, and this information is then propagated to the IF stage to stop fetching new instructions.

The result of the multiplication is finally written back in the register file.

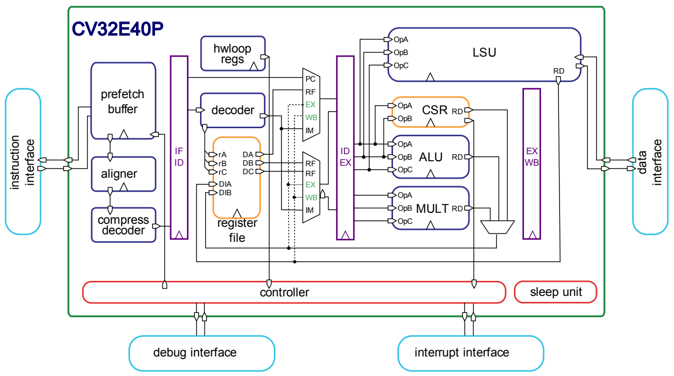

Fig. 3: CV32E40P RTL block diagram.

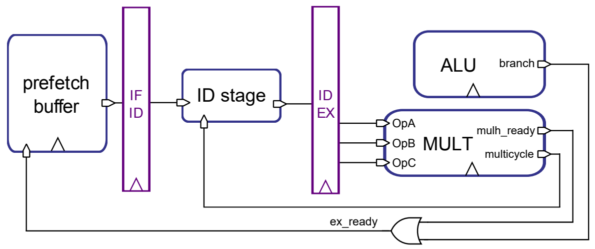

Fig. 4: Multiplier input/output signals and their use to stall the preceding pipeline stages in case of multicycle multiplication.

3) Prefetch Buffer: The CV32E40P processor has a PreFetch Buffer (PFB) in the IF stage, which performs word-aligned 32-bit prefetches and stores the fetched words in a FIFO with a queue of two instruction words. At the microarchitectural level, two independent program counters (PCs) exist. The first one, the PC_{IF}, specifies the last instruction that passed the IF stage. It is used in the following stages, in particular, to compute the target address of direct branches. The second one, the PC_{PFB}, indicates the memory address of the next instruction to fetch. Because of the speculative nature of the PFB, the PC_{PFB} is incremented ahead of time and independently from PC_{IF}. Both PCs are resynchronized when a branch is taken.

Figure 5 shows the FIFO used in the PFB to store fetched instruction words. The signal `status_cnt` corresponds to the number of instructions contained in the FIFO. The signals `full` and `empty` indicate its status, i.e., whether it is full or empty, respectively. The pointers `read` and `write` indicate the buffer location where the next value should be read or written. These pointers are incremented modulo the queue size since the buffer is circular. The FIFO ports for receiving and transmitting instructions are `data_i` and `data_o`, respectively. The behavior of the FIFO is controlled through the `push`, `pop` and `flush` signals. A `push` occurs when the next pipeline stage cannot process an instruction fetched from memory as it is not ready yet. In this case, the data on `data_i` is written to the buffer location pointed to by the `write` pointer, which is then incremented. A `pop` occurs as soon as the FIFO has at least one instruction and if the ID

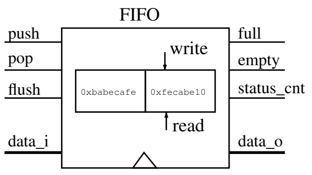

Fig. 5: FIFO of the CV32E40P prefetch buffer.

stage is ready to receive it. The appropriate data is produced on the output `data_o` and the `read` pointer is incremented. Otherwise, when the ID is ready to execute the instruction just fetched from memory, and if the FIFO is empty, the PFB is not used. A flush occurs at each taken branch as the content of the FIFO sequentially and speculatively fetched from memory is not relevant anymore. Pointers `read` and `write` are then reset to the same buffer location (location `0b0`) and the `status_cnt` is also set to 0. The flush does not erase the content of the FIFO, however.

C. Hardened CV32E40P

SCI-FI is a countermeasure designed against fault injection attacks that ensures code, control-flow, and execution integrity [18]. Execution integrity refers to the integrity of control signals emitted by the ID stage in the processor microarchitecture. The *pipeline state* built from these control signals is the cornerstone of the countermeasure. It is designed to capture any alteration of (i) the binary encoding of program instructions, or (ii) the control signals in the IF and ID stages At each cycle, an integrity signature is calculated from the current pipeline state value and the previous integrity signature. Hence, any modification of the pipeline state should result in an alteration of the integrity signature. The integrity signature is then checked against reference values placed at specific locations in the binary program (e.g., control-flow transfers). This ensures code and control-flow integrity until the end of the ID stage. A redundancy mechanism protects the control signals issued by the ID stage (i.e., downstream of the pipeline state calculation) to complete the integrity protection coverage throughout the processor microarchitecture. An alarm signal is emitted when an integrity violation is detected.

A complete security assessment of an implementation of SCI-FI wrt. FI attacks needs to consider the following security properties:

- ϕ_1 Faults applied upstream from the pipeline state lead to an alteration of the pipeline state.
- ϕ_2 Faults applied downstream from the pipeline state are detected by the redundancy mechanism, i.e., raise the alarm signal.
- ϕ_3 Pipeline state alterations or faults in the integrity signature calculation do not produce valid signature values (signature collisions). Faults applied to verifying the integrity signature to the propagation of the alarm signal do not leverage vulnerabilities.

In this paper, we study SCI-FI implementation integrated to the baseline implementation of the CV32E40P core [18].

TABLE I: Signal distribution in the SMT CV32E40P model that FIs can target according to their width (in bits).

1 bit	2 bits	3 bits	4 bits	5 bits	6 bits	7-31 bits	32 bits	32+ bits	Total
249	57	16	8	26	25	19	144	5	549
45.3 %	10.4%	2.9%	1.5%	4.7%	4.6%	3.5%	26.2%	0.9%	100%

The pipeline state integrates 13 control signals and has a size of 46 bits. Our study focuses on verifying properties ϕ_1 and ϕ_2 as introduced above. The verification of ϕ_3 is related to the security analysis of the integrity signature wrt. an attacker model, which depends on the function signature implemented. This was already carried out in the original work [18]. The security assessment of VerifyPIN programs run on SCI-FI consists in verifying that the vulnerability property φ is satisfiable under fault injections (3).

$$\varphi := \phi_0 \land \neg\phi_1 \land \neg\phi_2 \tag{3}$$

D. Verification Strategy

The vulnerability analysis we propose is motivated by the need to consider microarchitectural details when evaluating and explaining the effects of faults. However, at the microarchitectural level, analyzing the effects of a single fault targeting the data is not relevant as their consequences can be modeled at a higher level, e.g., corrupting an operand value or a general-purpose register just before its use. On the opposite, the control signals that drive the instruction fetching, decoding, and the data path are interesting targets to be faulted since such a fault can lead to effects that cannot be modeled at ISA level [12]. Note that when considering multiple fault injections, faults on both data and control signals must be considered as an exploit can be achieved by a combination of such faults. In this work, we only consider single faults. Consequently, we arbitrarily filter out large signals as they correspond to data signals. We only consider those smaller than 6 bits. Table I illustrates the width distribution of the target signals in the SMT model. Our analyses focus on the signals of width less or equal to 6 bits, i.e., 381 of the 549 available signals.

Our workflow is designed either for exploring the effects of faults (Section V-A), or for demonstrating the robustness of an HW/SW system to a given fault model (Sections V-B and V-C). However, when the vulnerability property is satisfiable, our workflow outputs a single counterexample. This only shows a single fault injection that leads to a vulnerability, even if others may exist. To overcome this limitation when exploring the effects of faults, we create several SMT problem instances checking the vulnerability property. More precisely, for each signal a fault can target, we generate as many SMT problem instances as the number of execution steps where a fault may occur as described in the corresponding fault model. In all the SMT problem instances, the fault value is symbolic. Thus, all different execution paths resulting from the possible fault values are explored in each single verification run. This solution enables us to identify which signals are vulnerable at which clock cycle.

In all the following workflow uses, we add a constraint that restricts the fault model to one fault injection. Without loss of generality of the method, this limits the results obtained to simpler and easier to explain effects. Consequently, only one fault injection is allowed to bypass secure authentication successfully. Finally, the input data (i.e., userPIN and cardPIN) are symbolic variables in both uses of our workflow (i.e., exploration the effects of faults in Section V-A and demonstrating robustness of systems in Sections V-B and V-C). Thus, one verification encompasses all the possible input data configurations under the initial constraint expressing that userPIN and cardPIN differ (1).

V. Experimental Results

This section presents the analysis results wrt. the experimental setup presented in Section IV. This section illustrates in particular the adaptability of our workflow.

A. Exploration of Fault Effects in the Microarchitecture

The analyses performed on `v3_Og` and `v3_Os` reported, in total, 59 vulnerabilities in `v3_Og` and 189 in `v3_Os`. Table II summarizes these results since several vulnerabilities identified are strictly equivalent or produce the same effect due to signal renaming in the hardware description[2], i.e., the same wire can have multiple names. As a consequence, the same signal can be targeted by fault injections at different spatial locations. In Table II, each successful fault injection is given according to its Verilog module (leftmost column, corresponding to the block diagram in Figure 3), one of the target signal names (i.e., spatial location), and the processor cycle at runtime (i.e., temporal location). Each fault injection lasts one processor cycle. The categories (column category) refer to the hardware feature that the fault injection corrupts. In the remainder of this section, we illustrate the effects corresponding to each category on selected examples of successful fault injection. The injected faulty value is not mentioned in this table but is given when relevant for the illustrative examples developed below.

1) Forwarding: By faulting the forwarding, an attacker can retrieve a value previously computed by a functional unit or read from memory into one of the operands of the EX stage. Laurent et al. have already shown similar results on the Rocket Processor [34]. The vulnerabilities reported in Table II in the category *FWD* exploit this mechanism. Both `v3_Og` and `v3_Os` are vulnerable to such faults. The fault injection inverts the conditional branch corresponding to the `if` statement at line 24 in Listing 4. As a consequence, the malicious authentication succeeds. Because of the difference between the two binary programs, the fault must be injected at cycle 57 in `v3_Og` and at cycle 47 in `v3_Os`.

2) Multiplier: The *MULT* category in Table II corresponds to faults impacting the FSM of the Multiplier module even if no multiplication is performed in the VerifyPIN program. As explained in Section IV, when the MULT enters an active state, it stalls the ID stage via the `multicycle` signal for up to 4 cycles. The security concern only arises when the ALU calculates a branch address at the instant of fault injection. In such a situation, the EX stage notifies the IF stage that it remains ready(`ex_ready` signal as shown in Figure 4) since a branch could be taken. Finally, the IF stage is not interrupted and continues to fetch instructions from memory while the ID and EX stages are busy performing multiplication. Fetched instructions are therefore ignored. Otherwise, if no branch is computed in the ALU, the injected fault does not affect the program's behavior since the multiplication result is ignored anyway. Depending on the state injected in the multiplication module, up to three instructions following a non-taken branch can be skipped. As previously explained in the forwarding vulnerability, this also corresponds to skip the `if` instruction at line 24 in Listing 4. This vulnerability does not exist on `v3_Og` since the instructions are differently ordered.

3) Aligner: The attacks targeting the Aligner module behavior are grouped in Table II under the category *ALGNR*. Injecting a fault into the Aligner module prevents the PC_{IF} from being updated, which desynchronizes it from the PC_{PFB}. Until the next *taken* direct branch, the program keeps running normally since the instructions continue to be correctly read from memory and sent to the next stage. When a direct branch

TABLE II: Results of the FI analysis on `v3_Og` and `v3_Os`: each *fault model* (designated by the signal name and the FI cycle) permits satisfying the vulnerability property $\varphi := \phi_0$.

Module	Targeted RTL Signal	Category	Cycle of Fault Injection	
			`v3_Og`	`v3_Os`
fifo	empty	*PFB*		46
	status_cnt_n	*PFB*	18, 21-26	26, 27, 45
prefetch_ctrl	flush_cnt_q	*PFB*		18, 45, 46
aligner	instr_valid	*other*	18	18, 19, 46
	branch_i	*ALGNR*	47	
	update_state	*ALGNR*	47	
	state	*other*		18, 19
id_stage	alu_bmask_b_mux_sel	*other*		39
	alu_vec_mode_ex	*other*	58	48
	bmask_b_mux	*other*	46	19, 20, 39
	branch_taken_ex	*other*	58	48
	id_valid	*other*	19	19, 20, 47
	imm_b_mux_sel	*other*		19, 20
	reg_d_alu_is_reg_a_id	*FWD*	57	20, 47
	regfile_alu_waddr_mux_sel	*other*	19	19, 20
controller	ctrl_fsm_cs	*other*		19, 20, 47, 48
	deassert_we	*other*	19	19, 20, 47
	halt_id	*other*	19	19, 20, 47
	is_decoding	*other*	19	19, 20, 47
	jump_in_dec	*other*	57	
	operand_a_fw_mux_sel	*FWD*	56, 57	19, 20, 46, 47
	operand_b_fw_mux_sel	*FWD*	57	47
	pc_set	*other*		48
decoder	alu_en	*other*		19, 20, 47
	alu_op_a_mux_sel	*other*	57	19, 20, 47
	alu_op_b_mux_sel	*FWD*	57	19, 20, 39, 47
	ctrl_transfer_insn	*other*	57	47
	regfile_alu_we	*other*	19	19, 20
	regfile_mem_we	*other*	46, 51	19, 20, 39
ex_stage	alu_cmp_result	*other*	58	48
	mult_en	*MULT*		19, 20
	mult_multicycle	*other*	19	19, 20, 47
	regfile_alu_we_fw	*FWD*	20	20, 21
	regfile_we_wb	*other*		21, 22
alu	cmp_result	*other*		48
	shift_left	*other*		20, 21
mult	multicycle	*MULT*	19	
	mulh_CS	*MULT*		46
lsu	data_be	*other*		17

[2]Implementation available at https://github.com/openhwgroup/cv32e40p

is taken, the PC_{IF} is used as a reference to calculate the target address. As the PC_{IF} does not correspond to the address of the currently executed instruction because of the attack, the program jumps to an incorrect address. The instruction flow then differs from the expected one.

4) Prefetch Buffer: Faults categorized with the *PFB* label in Table II consist in modifying signals that control the FIFO. For instance, the signal `status_cnt` indicates the number of instructions currently in the queue. The different possible effects of an attack injecting a non-null value in the signal `status_cnt` when the FIFO is empty are described below and illustrated in Figure 6.

1) **Execute the content of the FIFO:** A fault injection at cycle 22 sets the signal `status_cnt` to 1. As a consequence, the FIFO is no longer considered empty: the instruction `IA`†, which is a witness of a previous execution, is executed (`pop` signal). Moreover, the fetched instruction `I1`* is written in the FIFO (`push` signal). The `status_cnt` is then still equal to one, but the `read` ($\overset{r}{\downarrow}$) and `write` ($\overset{w}{\downarrow}$) pointers have been incremented. As a result, at cycle 23, instruction `IB`, which is also a witness of a previous execution, is sent to the ID stage (`pop` signal at cycle 23) and executed. The FIFO is now empty (cycle 24), and the `read` pointer points to I1, which has not been executed yet.
2) **Desynchronize the read and write pointers:** When reading instructions `IA` and `IB` in the FIFO due to the fault injection, the `read` pointer is incremented. Consequently, when the FIFO is empty again (cycle 24), the two pointers which should point to the same address are now misaligned. This has no effect during cycles 24-27 because the instructions are directly transmitted to the ID stage without going into the PFB since the FIFO is empty. However, when a stall occurs at cycle 27, the instruction `I5`* is stored and should be executed at the next cycle. Instead, the instruction `I1`† from the location pointed by the `read` pointer is sent to the pipeline at cycle 28. Only a taken branch permits resynchronizing the two pointers together by flushing the FIFO. The effects of the fault, i.e., executing a previously stored instruction instead of the correct one after each stall, disappear.
3) **Branching to an incorrect address:** As explained before, the PFB has its own PC (PC_{PFB}) to perform read requests in the instruction memory. However, since the contents of the FIFO have been replayed without notifying the rest of the processor, the PC_{IF} has shifted by one instruction (the value injected to the initially null `status_cnt` signal). Consequently, the next branch will be made one instruction too far (`I10` instead of `I9` at cycle 32).

As a result, a single fault injection in the PFB can lead to various effects with immediate and potentially longer-term consequences. In particular, the faulty value set to the 2-bit signal `status_cnt` can further desynchronize the read and write pointers: a branch up to 3 instructions after the correct address may happen.

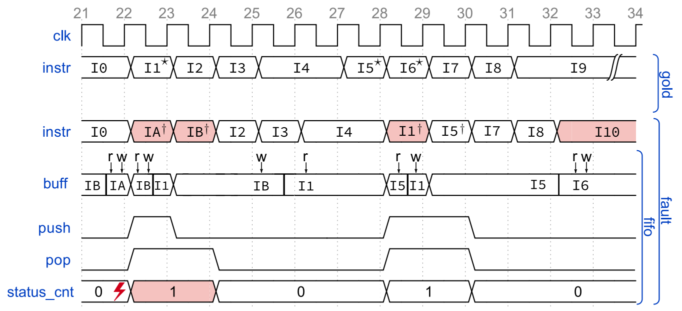

Fig. 6: Effects of fault injection on the PFB. Upper part named *gold* represents the non-faulty execution while lower part named *fault* illustrates the faulty execution where the signal `status_cnt` is subject to fault injection. Symbols $\overset{r}{\downarrow}$ and $\overset{w}{\downarrow}$ indicate the position of the read and write pointers in the prefetch buffer (*buff*). `Ix`* denotes instructions that should have been executed but were not because of the fault. `Ix`† identifies instructions badly fetched from the prefetch buffer due to the fault. Colored signals indicate a divergence wrt. to the gold execution.

In both `v3_Og` and `v3_Os`, the instructions contained in the buffer are replayed without side effects for successful faults that fall in this category. However, the desynchronization of the pointers `read` and `write` allows the PIN-comparison loop to be exited prematurely. The countermeasure verifying the loop counter detects this faulty behavior (corresponding to line 22 in Listing 4). A call to the countermeasure function is then performed but it jumps three instructions further. Consequently, the execution continues where the user is assumed to be authenticated (corresponding to line 27 in Listing 4).

5) Remaining Faults: Some faults identified in Table II are categorized as *other*. Many of them are related to multiplexer selector signals that control operations in the EX stage. These fault effects are mostly computational errors, and we develop one of them below. Performing a fault injection on the `bmask_b_mux` signal for both `Og` and `Os` version of verifyPIN allows a successful authentication. In the last update of the variable `diff` (Listing 4 line 20) the boolean False is written instead of the value True. This is due to the choice made for encoding hardened Booleans, i.e., False = 0x55 and True = 0xAA. The fault injection triggers a one-bit shift of the result to the right, i.e., 0xAA $\gg$ 1 = 0x55. This behavior raises questions about the choice of hardened Booleans. These certainly have a maximum hamming distance, but some elementary operations allow to pass from one to the other (e.g., False + False = True; False $\ll$ 1 = True), which should be avoided.

6) Conclusion: The described results involve subtle fault effects that are not directly addressable with ISA-level analysis. Specifically, we have targeted specific mechanisms in the microarchitecture, that are not visible at the ISA level, such as the pipeline or speculative behavior of the PFB. We also

TABLE III: Results of FI analysis on VerifyPIN_v7. Two *fault models* permit satisfying the vulnerability property $\varphi := \phi_0$.

Module	Targeted RTL Signal	Category	Cycle of Fault Injection	
			`v7_Og`	`v7_Os`
aligner	update_state	*ALGNR*	47, 52	

noticed that the found vulnerabilities are strongly related to the microarchitectural state at the fault injection time. Both the microarchitecture state before the fault injection (e.g., the content of the prefetch buffer which is replayed) and the state after the fault injection (e.g., the data remaining in the microarchitecture's memory elements) need to be considered. It can lead to the manifestation of the vulnerability only a long time after the injection.

B. Software Robustness Analysis

We now consider VerifyPIN_v7, which has a software countermeasure against test inversions. Some of the faults reported in the previous section lead to such an effect. Table III shows the results of the formal analysis on `v7_Og` and `v7_Os`. The analysis indicated that no fault injection exists to satisfy the attack oracle $\varphi := \phi_0$ for the considered fault model on `v7_Os`. The formal analysis only identifies two successful fault injections targeting the signal `update_state` on `v7_Og`. As explained before, by exploiting the desynchronization between PC_{IF} and PC_{PFB} in the IF stage, it is possible to jump a certain amount of instructions. Here, in both found vulnerabilities, the exit branch of the comparison loop directly jumps at the instruction that allows a successful authentication (line 27 in Listing 4).

In order to explain the difference between the results from VerifyPIN_v3 and VerifyPIN_v7, two reasons can be given. First, many fault injections previously identified by the analysis result in inverting or skipping a conditional branch. The *test duplication* countermeasure makes it impossible to reverse both tests with a single fault injection and these identified fault injections disappear. Two fault injections from the *ALGNR* category remain on `v7_Og` and permit to jump to a relative offset from a faulty PC value. The duplication test countermeasure is of no use against this fault effect. Finally, it is worth mentioning that vulnerabilities are often the result of a good timing of the fault injection with a well-chosen instruction sequence and a particular microarchitectural state left by the preceding executed instructions. Consequently, some vulnerabilities can disappear (or appear) due to minor differences in the binary code: number of instructions, code layout, instruction order, etc. Some previously found vulnerabilities rendered possible due to a particular order and instructions layout no longer exist due to the addition of test duplication countermeasure. In conclusion, vulnerabilities due to specificities of the hardware implementation can be very subtle and need to be analyzed by considering both the software and the hardware as our workflow proposes.

C. Hardware Countermeasure Analysis

SCI-FI is verified using two versions of VerifyPIN that are known to be the most vulnerable to single fault injection after our security analysis on the unprotected CV32E40P: `v3_Og`, and a specific version of VerifyPIN_v3 without the verification on the loop counter (in Figure 4, line 22). We use the vulnerability oracle φ (3) and the same microarchitectural fault models as previously (Section V-B). The verification did not identify any vulnerability.

TABLE IV: Verification time for each VerifyPIN analysis. # Fault Injection column indicates the number of fault locations (spatial and temporal) explored. userPIN, cardPIN and the fault effects are left symbolic.

VerifyPIN version	Overall Run Time (h)	# Fault Injections
`v3_Og`	12.9	15240
`v7_Og`	13.9	17526
`v3_Os`	14.1	14478
`v7_Os`	14.5	15240

D. Performance

Table IV shows the verification times required to obtain the results previously presented in Section V-B. Tests were carried out on Intel(R) Xeon(R) CPU E7-4870 @ 2.40GHz, 80 cores. As the PIN codes of the program are symbolic variables, we have at least 64 bits of degrees of freedom on each verification run. Similarly, the effects of the injected faults are not constrained either. The maximal depth of the BMC for the VerifyPIN programs reaches 64 for `v7_Og`. For the hardware countermeasure assessment (Section V-C), the overall verification run time is 25 hours, and the maximum BMC depth reaches 120. These results illustrate that the developed workflow can formally check in a reasonable amount of verification time the execution of a hundred instructions with symbolic data in the presence of faults.

VI. Conclusion

Facing the subtle effects of fault injection related to the implementation of microarchitecture and software characteristics, the design of sensitive systems requires security analyses encompassing both hardware and software. We presented a workflow that combines software and hardware models to formally explore fault injection effects. This workflow generates an SMT model of the hardware and extends it with fault injection modeling. Software is modeled as the content of the memory of the hardware SMT model. The workflow uses bounded model checking to verify the absence of vulnerability or outputs counterexamples that permit understanding the fault injection effects that result in an exploit. We illustrated the benefit of our workflow considering two versions of the CV32E40P RISC-V processor executing several versions of a PIN authentication. On the baseline processor implementation, the conducted analyses highlight new fault effects, highly dependent on the microarchitectural state at the fault injection time as well as the instruction flow afterward. We also reported well-known fault effects, but we showed that our workflow enables us to understand the fault propagation in the microarchitecture. We used the proposed workflow to verify that the countermeasure implemented in CV32E40P effectively detects microarchitectural fault injection. These use cases show that the proposed workflow is a valuable tool for exploring fault injection effects or designing secure hardware.

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*, vol. 2, no. 2, 2018.

[2] L. Zussa, J.-M. Dutertre, J. Clediere, and A. Tria, "Power supply glitch induced faults on fpga: An in-depth analysis of the injection mechanism," in *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. IEEE, 2013.

[3] L. Riviere, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer, and L. Sauvage, "High precision fault injections on the instruction cache of armv7-m architectures," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2015.

[4] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2013.

[5] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2002.

[6] J. Balasch, B. Gierlichs, and I. 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*, 2011.

[7] L. Dureuil, M.-L. Potet, P. d. Choudens, C. Dumas, and J. Clédière, "From code review to fault injection attacks: Filling the gap using fault model inference," in *International conference on smart card research and advanced applications*. Springer, 2015.

[8] T. Trouchkine, G. Bouffard, and J. Clédière, "Fault injection characterization on modern cpus," in *Information Security Theory and Practice*, M. Laurent and T. Giannetsos, Eds. Cham: Springer International Publishing, 2020, pp. 123–138.

[9] A. Menu, J.-M. Dutertre, O. Potin, J.-B. Rigaud, and J.-L. Danger, "Experimental analysis of the electromagnetic instruction skip fault model," in *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE, 2020.

[10] L. Claudepierre, P.-Y. Péneau, D. Hardy, and E. Rohou, "Traitor: a low-cost evaluation platform for multifault injection," in *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems*, 2021.

[11] J. Proy, K. Heydemann, A. Berzati, F. Majeric, and A. Cohen, "A first isa-level characterization of em pulse effects on superscalar microarchitectures: a secure software perspective," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, 2019.

[12] J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, "On the importance of analysing microarchitecture for accurate software fault models," in *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, 2018, pp. 561–564.

[13] D. Larsson and R. Hähnle, "Symbolic fault injection," in *International Verification Workshop (VERIFY)*, vol. 259. Citeseer, 2007.

[14] 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*, 2019.

[15] T. Given-Wilson, A. Heuser, N. Jafri, and A. Legay, "An automated and scalable formal process for detecting fault injection vulnerabilities in binaries," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 23, 2019.

[16] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. d. Choudens, "Fissc: A fault injection and simulation secure collection," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2016.

[17] OpenHW group, "CORE-V CV32E40P User Manual," https://cv32e40p.readthedocs.io/en/latest/intro/, 2020.

[18] T. Chamelot, D. Couroussé, and K. Heydemann, "SCI-FI: control signal, code, and control flow integrity against fault injection attacks," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022.

[19] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "The essence of bluespec: a core language for rule-based hardware design," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 243–257.

[20] D. Lustig, M. Pellauer, and M. Martonosi, "Pipe check: Specifying and verifying microarchitectural enforcement of memory consistency models," in *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014*. IEEE Computer Society, 2014.

[21] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: a platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 24:1–24:30, 2017.

[22] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification, 22nd International Conference, CAV 2010*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. B. Jackson, Eds., vol. 6174. Springer, 2010.

[23] C. Mattarei, M. Mann, C. W. Barrett, R. G. Daly, D. Huff, and P. Hanrahan, "Cosa: Integrated verification for agile hardware design," in *2018 Formal Methods in Computer Aided Design, FMCAD 2018*, N. S. Bjørner and A. Gurfinkel, Eds. IEEE, 2018, pp. 1–5.

[24] A. Goel and K. A. Sakallah, "AVR: abstractly verifying reachability," in *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020*, ser. Lecture Notes in Computer Science, A. Biere and D. Parker, Eds., vol. 12078. Springer, 2020, pp. 413–422.

[25] K. McMillan, "Cadence SMV."

[26] D. Kroening and M. Purandare, "EBMC," http://www.cprover.org/ebmc/.

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

[28] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani, "Verilog2smv: A tool for word-level verification," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016.

[29] M. Srivastava, P. Slpsk, I. Roy, C. Rebeiro, A. Hazra, and S. Bhunia, "Solomon: An automated framework for detecting fault attack vulnerabilities in hardware," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020.

[30] 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)*. IEEE, 2020.

[31] 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)*. IEEE, 2008.

[32] 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 *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014.

[33] 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)*. IEEE, 2016, pp. 47–58.

[34] 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)*. IEEE, 2019.

[35] J. Laurent, C. Deleuze, F. Pebay-Peyroula, and V. Beroulle, "Bridging the gap between RTL and software fault injection," *ACM Journal of Emerging Technologies in Computing System*, vol. 17, no. 3, 2021.

[36] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-end automated exploit generation for processor security validation," *IEEE Design & Test*, vol. 38, no. 3, 2021.

[37] C. Barrett, A. Stump, C. Tinelli *et al.*, "The SMT-LIB standard: Version 2.0," in *Proceedings of the 8th international workshop on satisfiability modulo theories*, vol. 13, 2010.

[38] B. Dutertre, "Yices 2.2," in *International Conference on Computer Aided Verification*. Springer, 2014.

[39] Z. Snow, "sv2v," https://github.com/zachjs/sv2v.