

Future Result Capture: Timing Anomalies Reveal Data from Instructions in the Successor

Roua Boulifa*, Marwa Chehab*, Paolo Maistri*, Giorgio Di Natale*

*Univ. Grenoble Alpes, CNRS, Grenoble INP¹, TIMA, 38000 Grenoble, France

Abstract—Fault injection remains a critical threat to modern embedded processors, enabling adversaries to violate the expected execution of programs. In this paper, we introduce a new fault model, named Future Result Capture (FRC), observed on a commercial processor implementing the RISC-V instruction set architecture. Unlike classical models, FRC occurs when an instruction captures the result of a subsequent instruction, effectively creating a temporal inversion in the pipeline. Through extensive clock and voltage glitch experiments on a SiFive RISC-V core, we show that this phenomenon can be consistently triggered, producing instruction-level causality violations not explained by existing models. This new fault behavior exposes unexplored vulnerabilities in commercial processors, demonstrating that subtle microarchitectural effects can be exploited by physical attacks. Our findings highlight the necessity to revisit current fault models by taking into account a thorough understanding of the microarchitectural features of microprocessors, in order to design efficient countermeasures specifically addressing such vulnerabilities.

Index Terms—RISC-V, Embedded System, Hardware Security, Fault Injection Attacks

I. INTRODUCTION

Fault injection attacks [1] are a class of active attacks in which an adversary deliberately perturbs the operation of a circuit to induce faulty behavior. By forcing the system into abnormal states, attackers may bypass security checks, reveal sensitive information, or disrupt correct execution. In processors, these perturbations are typically observed as computational faults, as unexpected changes in the system state can compromise computation integrity. Fault effects were initially studied in the aerospace industry, where radiation-induced upsets could disturb electronics. In the security domain, Boneh et al. [2] first highlighted the potential of injecting intentional faults to break cryptographic algorithms, which gave rise to a rich line of research on hardware fault attacks.

Several techniques have since been proposed to perform fault injection. Common approaches include manipulating the power supply [3], applying electromagnetic pulses [4], [5], or using laser stimulation [6], [7]. Of particular interest are clock glitch and voltage glitch attacks, which directly exploit the timing sensitivity of digital circuits and are accessible to a large public because of their simplicity and reduced cost. These low-cost and widely applicable techniques have therefore become prominent in fault injection studies targeting embedded systems. By briefly altering the clock signal or the supply voltage, attackers can disturb the pipeline flow of processors, leading to faulty instruction execution [8].

To interpret and anticipate the impact of such attacks, fault models have been introduced at different abstraction levels [6],

ranging from the electrical level to the logic gate, microarchitectural, ISA, and software levels. Lower-level models capture the physical effects of injections with higher accuracy, but they are more complex and resource-intensive to simulate. Higher-level models, in contrast, offer simpler abstractions but reduced realism. This trade-off highlights the importance of understanding the actual fault occurrence mechanisms at each abstraction level, in order to address potential vulnerabilities, and design effective countermeasures.

For these reasons, Instruction Set Architecture (ISA) level is often used to describe fault attack effects due to the reasonable compromise between realism and abstraction. At this level, faults are often modeled as corrupted or skipped instructions, which are easy to represent at the ISA level. Until now, most studies have highlighted how instructions can be skipped, repeated, or forged by merging different memory words [9], which usually depend on the mechanisms used to recover the instructions from the respective memory. Recent studies, however, have shown that the hidden microarchitectural features of microprocessors play a non-negligible role in shaping the faulty behavior. In general, it has been shown that the whole pipeline may be affected by timing faults [7], for example leading to an instruction sampling a previous result [10].

In this paper, we present a novel fault model where the desynchronization of the pipeline leads to an instruction to sample and store the result of the next one. We refer to this behavior as Future Result Capture (FRC), as the result seems coming from the future. We validate and prove this behavior through extensive clock and voltage glitch injection campaigns conducted on a 32-bit ASIC RISC-V processor. To our knowledge, this is the first time that such behavior is identified among commercial CPUs.

The remainder of this paper is organized as follows: Section II provides the necessary technical background. Section III presents the Future Result Capture fault model. Section IV describes the experimental setup, while Section V analyzes the different results of our fault injection campaigns. An exploitation example is given in Section VI. Finally, Section VII concludes the paper.

II. BACKGROUND

A. Microprocessors in Embedded Systems

Embedded systems may rely on simple processors in order to execute custom tasks. Generally, these CPUs are designed with goals of efficiency, reduced cost, and minimal power consumption. As a consequence, this leads to design choices such as simple and rather short pipelines, usually 2 to 5 stages:

¹Institute of Engineering Univ. Grenoble Alpes

fetching, decoding, execution, memory access, and write-back, possibly merged together depending on the specific architecture. Moreover, there are minimal or no optimizations at all: this means single-issue in-order execution, and deterministic branch speculation at the best. Caches are optional, as they increase cost without the associated requirement for performance. Under these circumstances, software execution is quite deterministic under nominal conditions.

B. Clock and Voltage Glitch Fault Injection

Fault injection through clock and voltage glitches is a widely adopted method for disturbing the normal execution of digital systems. Both techniques exploit the timing sensitivity of synchronous circuits by introducing short perturbations affecting the timing constraints, leading to abnormal processor states.

In the case of **clock glitches**, the attack consists of modifying the clock waveform that drives sequential elements such as flip-flops. Similarly, **voltage glitches** operate by introducing variations into the power network of an embedded system, usually by briefly lowering the power supply. Since these transitions govern signal propagation and data sampling, an injected disturbance may create timing violations in the pipeline. Depending on when and how the glitch occurs, it may alter instruction execution or corrupt register updates, potentially yielding exploitable faulty behavior.

In both cases, the fault injection process is controlled through three key parameters, well documented in the literature:

- *Delay*: the number of cycles between the synchronization trigger and the targeted clock cycle.
- *Shift*: the time offset between the reference edge of the targeted cycle and the start of the glitch.
- *Width*: the duration of the perturbation applied to the clock or the power supply.

The outcome of a glitch injection depends strongly on these parameters. Some configurations may have no visible effect, while others lead to crashes or to exploitable faults. To obtain useful results, the attacker must finely tune the glitch settings, exploring different cycles and pulse shapes until abnormal yet reproducible behaviors are achieved.

C. Fault Attack Models in CPUs

As described previously, several studies have focused on modeling fault attacks at the ISA level. This is due to the fact that ISA level allows modeling fault at the software level, with a reasonable knowledge of the underlying hardware (without its inherent complexity). Moreover, this level allows models independent of the actual fault injection technique, and reasonably comparable among different targets or architectures.

Several works have highlighted behaviors such as instructions skips (and possibly repeats), where one [11], [12], or several [13], [14] instructions are replaced by NOPs in the control flow. So far, instructions are generally affected in their entirety and possibly according to the instruction cache structure [14]; when a faulty behavior is out of the known schemes, it is usually considered as a generic corruption. These behaviors have largely been verified on both ARM embedded

CPUs such as Cortex-M, and RISC-V implementations [15], [16], and may be affected by memory alignment as well [9].

These errors are usually described as due to timing disruptions in the fetching phase, since perturbation alter the correct sampling on the instruction path. This is confirmed by works studying the impact of prefetching structures in the pipeline [17], or the precharging mechanisms of instruction buses [18]. Only recently it has been shown that other CPU stages can be affected by timing faults. In [10], authors have shown that sampling fault may occur between the Execution and Write-back stage of a RISC-V implementation, leading to a register storing the result of the previous instruction. In such case, however, the fault model was proved on an FPGA implementation, where path delays are largely dependent on the Placement and Routing performed by the design tool.

In the next section, we will show how abnormal pipeline timing can be observed on ASIC implementations as well.

III. PROPOSED FAULT MODEL: FUTURE RESULT CAPTURE

As part of our ongoing fault modeling experiments on a SiFive FE310-G002 microcontroller, we injected clock glitches while executing a simple test program combining basic arithmetic operations with register initializations. The FE310-G002 is a 32-bit RISC-V processor built around the E31 Core, which implements the RV32IMAC instruction set and a five stage in order pipeline. At first, the resulting faulty behaviors seemed irregular, almost resembling random errors. Yet, one specific anomaly consistently reappeared, which prompted us to investigate further. This observation ultimately led us to identify and define a new fault model. In this section, we describe how we first encountered this phenomenon, propose a name for the fault model, and outline its main characteristics. The following code sequence illustrates the type of program where this behavior was initially detected, where the instruction in bold indicates the line targeted by the clock glitch:

```

1 addi x25, x25, 0xf # x25 = 0x3f
2 addi x26, x26, 0xe2 # x26 = 0x142
3 addi x27, x27, 0x8d # x27 = 0x11d
4 addi x28, x28, 0x5 # x28 = 0x17

```

Listing 1: Simple target assembly code

To ensure deterministic behavior, the registers were initialized beforehand with arbitrary fixed values: x25=0x30, x26=0x60, x27=0x90, x28=0x12.

This experimental setup offered a controlled baseline for fault injection. At first, the behavior we observed under clock glitching seemed like a random corruption: register x26 failed to retain the value produced by its own update instruction and, instead, latched the value destined for register x27. The anomaly gave the impression that the processor was capturing a result from the future, almost as if it were skipping ahead in time. We refer to this phenomenon as the Future Result Capture (FRC) fault model.

Since the exact architecture and microarchitecture of the targeted RISC processor are not publicly documented, we cannot determine with certainty what happens internally during

the fault injection. However, based on our observations, we believe that a glitch may disrupt the pipeline timing in such a way that instruction I does not capture its own computed result, but instead latches the value being produced by instruction $I + 1$. In other words, the synchronization between execution and write-back stages may be disturbed, allowing a future computation to overwrite the expected result of the current instruction.

Interestingly, when we extended our experiments from clock glitching to *voltage glitching*, the same *Future Result Capture* phenomenon was still observed. However, voltage glitches also introduced a wider range of faulty behaviors. Because they perturb the processor more globally than clock glitches, these faults occurred with higher frequency and were often accompanied by additional error patterns.

In the following section, we present the research hypotheses that guide our investigation. Our goal is to better understand under which conditions, and with which types of instructions, the Future Result Capture behavior can be observed:

- (a) Does the FRC model affect all ALU operations in the same way, or does the opcode pairing (op_i, op_{i+1}) influence the likelihood and manifestation of FRC?
- (b) Does the propagation of faults generated by the model remain bounded to the initially faulty instruction, or can they extend across multiple subsequent instructions?
- (c) Does inserting NOP instructions after a targeted instruction eliminate the FRC effect?
- (d) Does the FRC fault model manifest differently for multi-cycle instructions (e.g., division) compared to single-cycle ALU operations?
- (e) How does the duration of an instruction’s execution (single-cycle vs. multi-cycle) impact the propagation behavior of the FRC fault model?
- (f) What is the impact of control flow instructions (e.g., *jump*) on FRC Propagation?

For this reason, our analysis will mainly focus on delving deeper in the comprehension of the FRC model; each question will be separately addressed in Section V. It is important to note, however, that during our experiments several fault models from the literature [10], [15] were also frequently observed.

IV. EXPERIMENTAL SETUP

A. Experimental Device

The boards that are used for the experiments are the ChipWhisperer [19] boards: ChipWhisperer Lite Capture and the CW308 UFO baseboard, along with the target device. These ChipWhisperer boards provide a dedicated environment for side channel analysis and fault injection, including voltage and clock glitching capabilities, which are both leveraged in our campaigns. The boards are configured in order to use the

Capture board to generate and control the clock fed to the target, as well as its power supply.

The target device is a SiFive 32-bit microcontroller (FE310-G002) that embeds an E31 RISC-V core [20]. This core has a 5-stage pipeline: fetch, decode, execute, data memory access, and register writeback. It has 32 general purpose 32 bit registers, named X0 to X31. The SiFive E31 core supports the full RV32IMAC instruction set, which includes the standard Integer (I), Multiply (M), Atomic (A), and Compressed (C) extensions. In our experiments, we will mainly explore the impact of Integer and Multiply set.

B. Experimental Protocol

In order to investigate the effects of fault injection, we have carried out several physical fault injection experiments. This aims at providing better characterization and description, for the wide range of faulty behaviors obtained when performing fault injection campaigns. The process is repeated for each experiments :

- (1) the FE310-G002 microcontroller is programmed to run the target code;
- (2) the computer configures the glitch parameters on the glitch board;
- (3) the target CPU executes the target program, which raises a synchronization trigger which then (4) triggers the glitch injection according to the given parameters. After the glitch, (5) the target CPU reads the registers and sends the value back to the control PC to analyze the effects of the glitch. This protocol can be repeated iteratively with different glitch settings.

The clock glitch parameters depend on the target code: in particular *Delay* is used to target specific instructions. Delay values are customized to each target program, as they depend on the number of initialization instructions before the target code, and were set to cover all the instructions within the target code. For each combination of Shift, Width, and Delay, the experiments are repeated 20 times, which led to a total number of 96000 fault injection experiments. It should be mentioned that, by properly tuning the parameters, it is possible to maximize the observability of a specific faulty behavior.

All the instructions are 32-bit-wide and are aligned in the memory [9] and were chosen with a specific initialization (shown in table I) and goal. The instructions have been explicitly chosen to be simple enough to facilitate the characterization of fault injection effects, making it easier to identify and analyze potential faulty behaviors. After the normal execution completes, each register contains a unique value, which improves the ability to detect any faults that might have occurred.

C. Cache Warm-up

When injecting a fault, whether by clock or voltage glitch, it is essential to precisely align the glitch with the execution of the target instruction. Our injection setup relies on the ChipWhisperer framework, which provides an API instruction to synchronize the fault with the processor’s execution. This synchronization is clock-accurate, since it is based directly on the processor’s clock cycles.

However, in practice, the exact execution time of an instruction does not always correspond to a fixed cycle. This

TABLE I: Initial values for registers in target programs

x5	0x300	x19	0x2	x23	0x20	x27	0x200
x6	0x500	x20	0x4	x24	0x40	x28	0x400
x7	0x2	x21	0x8	x25	0x80	x29	0x800
x18	0x1	x22	0x10	x26	0x100	x30	0x1000

variability is due to the presence of an instruction cache in the target processor. Depending on whether a cache hit or miss occurs (and on the state of preceding instructions), the same instruction may be executed earlier or later relative to the synchronization reference. As a result, the intended target instruction is not always reached at the expected cycle, complicating reliable fault injection.

To mitigate this problem, we encapsulated the target code into a simple loop that executes it twice. During the first iteration, the code runs normally without injection, ensuring that the instruction cache is warmed up. In the second iteration, the fault injection is activated, and the results are collected. This strategy minimizes cache-related timing uncertainty and increases the reliability of targeting a specific instruction.

V. EXPERIMENTAL RESULTS AND DISCUSSION

A. Single-Cycle ALU Operations

In Section III, we were able to highlight that *addi* instructions could be altered and the following result could be sampled instead. In this section, we want to assess whether this behavior is specific to some specific instructions (i.e., opcodes). For this reason, we target a simple code, shown in Listing 2, where several ALU operations are used. The operation used in the code have the property to have all no-cycle penalty, meaning that they are all completed within one single clock cycle in the execution stage. The same initialization values from the previous section were chosen to simplify the propagation effects.

In our tests, we targeted *instr #2* with glitch injections and were able to observe the FRC model, meaning that the result of *instr #3* was stored into register *x20*. We tried several variants, where *instr #3* was chosen among several ALU instructions (AND, OR, SUB, XOR, ADD, ...): in all cases, we were able to reproduce the occurrence of the FRC model. Conversely, the same behavior was observed when changing the target instruction with one from the same set of single-cycle instructions.

This confirms that the faulty behavior is consistent regardless of the opcode at either the target position, or the following one, highlighting that the FRC model stems mainly from pipeline timing.

B. FRC Propagation to Subsequent Instructions

In previous examples, we have shown how an instruction could be corrupted and forced to get the result of the next one. The effects of such pipeline disruption, however, are not limited to this. In order to assess the effect on code that is executed later, we extended the target program beyond the

```

1 addi x19, x19, 0x10 # x19 = 0x12
2 addi x20, x20, 0x40 # x20 = 0x44
3 and x23, x26, x27 # x23 = 0x0
4 xor x22, x25, x24 # x22 = 0xC0
5 sub x21, x30, x18 # x21 = 0xFFF
6 or x31, x28, x29 # x31 = 0xC00

```

Listing 2: Base program with ALU operations

```

1 addi x19, x19, 0x10 # x19 = 0x12
2 addi x23, x23, 0x100 # x23 = 0x120
3 sub x22, x5, x18 # x22 = 0x2ff
4 addi x20, x20, 0x40 # x20 = 0x44
5 add x21, x6, x5 # x21 = 0x800
6 addi x24, x24, 0x200 # x24 = 0x240
7 addi x25, x25, 0x30 # x25 = 0xb0
8 addi x27, x27, 0x60 # x27 = 0x260
9 xor x26, x7, x18 # x26 = 0x3
10 xor x28, x6, x5 # x28 = 0x600
11 addi x29, x29, 0x20 # x29 = 0x820
12 sub x30, x6, x7 # x30 = 0x4fe

```

Listing 3: Extended ALU target code (with golden values)

initial four instructions and used a longer sequence to study how faults propagate during execution. The test program is shown in Listing 3. In this code, each destination register was initialized with a one-hot encoding, ensuring that every register began with a distinct pattern.

In such extended program, we were able to observe a clear propagation pattern. When the first instruction (*addi x19, x19, 0x10*) was faulted, the FRC fault model occurred; as a collateral effect, the second instruction was skipped, whereas the third was executed correctly. This could mean that the glitch altered some of the internal control signals of the pipeline, leading to write-back of the 2nd instruction into the previous destination register.

The propagation of the glitch is, however, affecting further instruction, because the next block of four operation exhibited a specific and reproducible pattern: instruction skip, corruption, corruption again, and finally skip. The same propagation pattern occurred relatively to the targeted instruction, independently of the actual delay. For these reasons, we may assume that the pipeline state was affected more than initially thought: the missing knowledge of the internal microarchitecture, however, make further hypotheses difficult to confirm.

C. Inserting NOP Instructions

Since the FRC model causes the destination register of instruction *i* to be overwritten with the result of instruction *i + 1*, one simple countermeasure might insert one or more *nop* (no-operation) instructions immediately after the target.

In this case, it is important to take into consideration how the *nop* operation is implemented in the considered ISA: for the considered CPU, it is actually implemented as *addi x0, x0, 0*, which is a regular arithmetic instruction which has no effect and whose result is always zero. Under these conditions, a reset of the destination register of the targeted instruction was observed, which is perfectly compatible with the FRC model. We observed that the instruction preceding the *nop* captured this zero value, which is consistent with the behavior of the FRC fault model. Beside the *nop* instruction, the propagation pattern follows the same sequence described in the previous section: FRC → skip → golden → skip → corrupt → corrupt → skip. From these observations, it looks that *nop* is no different than a regular arithmetic instruction.

The observed behavior was however different when several *nop* instructions were inserted later in the code, in order to force bubbles into the pipeline and thus try to break the fault propagation. We hence tested blocks of 3 to 5 *nop* instructions, inserted two instructions later than the target. If the targeted instruction is #1, for instance, *nops* were inserted from instruction #3 up to #5, #6, or #7. In the former case with just 3 *nops*, we observed that only the second instruction after the *nop* block is skipped; when using 4 or 5 *nops*, then execution continues regularly.

In summary, inserting `nop` instruction immediately after the targeted instruction does not eliminate the manifestation of the **FRC** fault model: on the contrary, a register reset is observed. With more *nops* instructions, the fault propagation can be delayed or progressively suppressed until complete cancellation. With that in mind a possible countermeasure could be the duplication of the critical instruction, where the second execution leverages an additional (and later unused) register, followed by 4 *nop* instructions: a fault targeting the first instruction would capture the result of the duplicated one; a fault targeting the redundant instruction or the *nops* would be ineffective. We must admit, however, that such solution would incur in an exceptional overhead, likely not acceptable in common scenarios.

D. Multi-Cycle ALU Instructions

In many embedded RISC-V cores, operations such as Multiplication and Division are implemented as multi-cycle operations through a dedicated unit. These instructions have therefore different datapath timing, writeback latency, and bypass behavior compared to single-cycle ALU instructions, which may change their susceptibility to FRC faults.

The presence of such operations affects the occurrence and propagation of the fault. If the targeted instruction is a `MUL` or `DIV` instruction, then FRC does not occur and the observed fault model is rather an *Instruction Skip*, or *Skip & Repeat*. The only exception appears to be the division by 2, which may highlight a different architecture path for this special case which corresponds to left shift. In this particular case, FCR may occur, but such event is not deterministic even when tuning the parameters: other fault models from the state of the art (e.g., *Skip*) may be observed.

E. Effect of Instruction Duration on FRC Propagation

In the general case discussed in Section V-A, the FRC propagation for single-cycle instructions typically affects up to 6 instructions after the fault model appears. If the multi-cycle operations are executed after the targeted instruction, FRC still occurs as in the general case: the subsequent fault propagation, however, is heavily affected in a complex way. This is particularly true for the Division, where *Instruction Skips* and *Instruction Corruptions* are interleaved and occur differently depending on `DIV` distance from the target.

On the other hand, a few experiments showed that the multi-cycle `DIV` instruction could extend the propagation even further, impacting up to 9 subsequent instructions. This indicates that multi-cycle instructions not only increase the window of

susceptibility but also introduce a distinct propagation signature compared to single-cycle ALU instructions.

F. Control-Flow Instructions

Jump instructions alter the normal sequential execution flow by redirecting the program counter to a different address. This control-flow discontinuity modifies the instruction fetching and pipeline content. In order to study how a jump instruction affect the occurrence and the propagation of the FRC model, we modified the code at specific positions. The observed outcomes reveal interesting insights about the processor’s behavior.

When the instruction targeted by the fault injection is the jump itself, the observed behavior is an *Instruction Skip*, or *Skip & Repeat*. This is somewhat expected, as the ALU datapath is not leveraged during the execution of a direct jump.

If the jump is the instruction immediately after the target, it is not executed: as shown in Section V-A, the instruction in this position is skipped when FRC occurs. Two side effects can be observed: (1) the previous instruction should exhibit the FRC model, but since the ALU is not used, no result can be sampled from the ALU and hence a *Skip* occurs; and (2), since the jump is not executed, the following instruction are fetched, decoded, and executed when they should not have. From an external point of view, this model could resemble test inversion at the software level, which might constitute an exploitable vulnerability in several application scenarios.

If the jump instructions is executed well after the glitch, then the regular pattern (Section V-A) is preserved until the execution of the jump itself, which empties the pipeline and thus “resets” the execution. A particular case occurs however in case such as Listing 4, where jump instruction at position #3 is correctly executed (as expected), but some corrupted signals are still latent in the microarchitecture. For instance, in the proposed example, we were able to observe the skipping of the 3rd instruction after the jump destination.

G. Discussion

All these results were validated on a second identical device, demonstrating that the observed faulty behaviors are inherent to the microarchitectural implementation rather than caused by device-specific effects or process variability.

The FRC fault model can be explained by analyzing the timing and interactions between pipeline stages during fault injection (Fig. 1). We hypothesize that the glitch is injected when the targeted instruction is in the write-back stage of

```

1 add x21, x7, x5
2 addi x23, x23, 0x100
3 addi x25, x25, 0x30
4 j jump_here
5 sub x22, x5, x18
6 xor x26, x7, x18
7 addi x20, x20, 0x40
8 jump_here:
9 xor x28, x6, x5
10 addi x24, x24, 0x200
11 addi x27, x27, 0x60

```

Listing 4: Control-flow target code

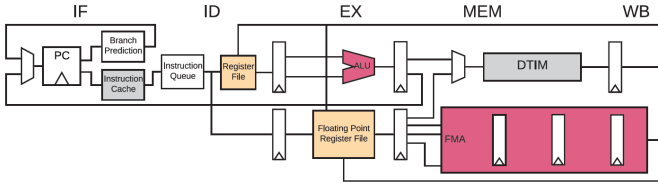


Fig. 1: Pipeline architecture of the SiFive E31 core [20]

the pipeline. The glitch induces a temporary desynchronization between the control path and the data path, preventing the write-back logic from correctly latching the intended result. As a consequence, instead of committing the correct value, the processor captures the output of the execute stage of the subsequent instruction. This mechanism explains the systematic forward result capture observed in our experiments.

The impact of the glitch is not limited to the targeted instruction. Because multiple instructions are concurrently in the pipeline, the perturbation propagates across several in-flight instructions. Experimental evidence shows that instructions located in stages close to critical timing paths are particularly vulnerable, resulting in corrupted execution, instruction skips, or incorrect decoding. Other stages, which likely exhibit larger timing margins, may complete their operations unaffected. Overall, the glitch globally perturbs pipeline operations, and the specific fault manifestation depends on the relative timing between control and data signals at the moment of injection.

Finally, the spatial extent of the observed fault pattern can be explained by the depth of the pipeline itself. The processor features a five-stage pipeline complemented by an instruction queue, meaning that up to seven instructions may be simultaneously present in different stages of execution. This architectural characteristic directly matches the experimental observation of fault effects spanning up to seven consecutive instructions.

VI. CASE STUDY: VERIFYPIN

To demonstrate the effectiveness of the proposed Future Result Capture fault model, we applied it to a widely studied security benchmark: the verifyPIN program. This real-world example highlights how the fault model can be exploited to bypass a common password verification mechanism.

The program performs a byte-wise comparison of a fixed-length candidate password against a reference, setting a success flag only if all characters are correct. This type of routine models a common security check mechanism and is therefore a natural target for fault injection analysis.

The assembly code of a basic verifyPIN is shown in Listing 5. At initialization, the address of the user-provided password is loaded into register x28, and the address of the expected password into x29. A counter (x7) is set to zero, and the loop limit (x6) is set to five, corresponding to the fixed password length. The program then enters the loop (label 1), which iterates until the counter reaches the limit. In each iteration, the current byte of the input password is accessed at $x30 = x28 + x7$ and stored in x31, while the corresponding byte of the expected password is loaded into x5. The two bytes are compared: if they match, execution continues; otherwise, the

```

1      mv x28, pw
2      mv x29, pw_ref
3      li x7, 0
4      add x6, x0, x0
5      addi x6, x6, 5
6 1b:  bge x7, x6, 3f
7      add x30, x28, x7
8      lbu x31, 0(x30)
9      add x21, pw_ref, x7
10     lbu x5, 0(x21)
11     beq x31, x5, 2f
12     sb x0, 0(&passok)
13 2f:  addi x7, x7, 1
14     j 1b
15 3f:

```

Listing 5: Verify PIN program in RISC-V assembly

variable passok (pointed to by x12) is cleared to 0, marking the comparison as failed. After each iteration, the counter (x7) is incremented, and the loop repeats. Once the counter reaches the limit, the loop exits (label 2). At this point, passok remains set to 1 only if all bytes of the input password matched the expected password.

Under the proposed FRC fault model, we demonstrate that the verifyPIN check can be bypassed. When the fault causes register x28 (the candidate pointer) to capture the value of x29 (the reference pointer), the loop compares the reference password against itself, ensuring that every comparison succeeds. As a result, the verification routine always leaves passok set to 1, even when the candidate password is incorrect. This outcome shows that our model can be directly exploited to compromise a widely used security mechanism.

VII. CONCLUSION

This work introduced a novel fault model, the Future Result Capture (FRC), and validated it on a commercial RISC-V core. FRC reveals previously unexplored microarchitectural vulnerabilities arising from pipeline timing and instruction sequencing disrupted by fault injection.

Through extensive experimental campaigns, we evaluated FRC across different instruction types and usage scenarios. The results show that the phenomenon is independent of specific opcodes and that faults can propagate beyond the initially targeted instruction, sometimes corrupting several subsequent ones. The propagation patterns depend on execution latency and instruction type. While inserting NOPs can reduce the effect, it may completely suppress FRC only at a prohibitive performance cost.

These findings highlight that existing fault models fall short of capturing the full complexity of microarchitectural behavior under fault injection. A deeper understanding of instruction timing, pipeline interactions, and execution semantics across the complete instruction set is essential to design effective countermeasures. Addressing these challenges will be key to hardening future processors against FRC and related threats.

ACKNOWLEDGMENT

This work is supported by the ARSENE project, funded by the “France 2030” government investment plan managed by the French National Research Agency (ANR-22-PECY-0004).

REFERENCES

- [1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," in *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370-382, Feb. 2006, doi: 10.1109/JPROC.2005.862424.
- [2] Boneh, D., DeMillo, R. A., and Lipton, R. J. "On the Importance of Checking Cryptographic Protocols for Faults," in *Advances in Cryptology — EUROCRYPT '97*, W. Fumy, Ed., Springer, 1997, pp. 37-51.
- [3] T. Korak and M. Hoefler, "On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms," in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, September 2014, pp. 8-17.
- [4] S. Ordas, L. Guillaume-Sage, and P. Maurine, "Electromagnetic Fault Injection: The Curse of Flip-Flops," *Journal of Cryptographic Engineering*, vol. 7, no. 3, September 2017, pp. 183-197.
- [5] M. Dumont, M. Lisart, and P. Maurine, "Modeling and Simulating Electromagnetic Fault Injection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 4, pp. 680-693, 2021.
- [6] T. Troughine, G. Bouffard, and J. Clédière, "Fault Injection Characterization on Modern CPUs: From the ISA to the Micro-Architecture," in *Proceedings of the Conference*, Springer, 2019, pp. 123-138.
- [7] V. Khuat, J.-L. Danger, and J.-M. 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)*, Milan, Italy, 2021, pp. 74-85.
- [8] M. Agoyan, J.-M. Dutertre, D. Naccache, B. Robisson, and A. Tria, "When Clocks Fail: On Critical Paths and Clock Faults," in *Smart Card Research and Advanced Application*, D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, Eds., Springer, 2010, vol. 6035, pp. 182-193.
- [9] I. Alshaer, G. Burghoorn, B. Colombier, C. Deleuze, V. Beroulle, et al., "Cross-layer analysis of clock glitch fault injection while fetching variable-length instructions," *Journal of Cryptographic Engineering*, 2024, 14 (2), pp.325-342. doi: 10.1007/s13389-024-00352-6.
- [10] R. Boulifa, P. Maistri and G. Di Natale, "Early Result Capture: Racing Conditions in Pipeline due to Clock Glitches," *2025 IEEE European Test Symposium (ETS)*, Tallinn, Estonia, 2025, pp. 1-6, doi: 10.1109/ETS63895.2025.11049616.
- [11] J. Proy, K. Heydemann, A. Berzati, F. Majeric, A. Cohen. A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures. *ARES 2019 - 14th International Conference on Availability, Reliability and Security*, Aug 2019, Canterbury, United Kingdom. pp.7:1-7:10, doi: 10.1145/3339252.3339253.
- [12] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick and P. Schaumont, "Software Fault Resistance is Futile: Effective Single-Glitch Attacks," *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Santa Barbara, CA, USA, 2016, pp. 47-58, doi: 10.1109/FDTC.2016.21.
- [13] A. Menu, J. -M. Dutertre, O. Potin, J. -B. Rigaud and J. -L. Danger, "Experimental Analysis of the Electromagnetic Instruction Skip Fault Model," *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, Marrakech, Morocco, 2020, pp. 1-7, doi: 10.1109/DTIS48698.2020.9081261.
- [14] L. Rivière, Z. Najm, P. Rauzy, J. -L. Danger, J. Bringer and L. Sauvage, "High precision fault injections on the instruction cache of ARMv7-M architectures," *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, Washington, DC, USA, 2015, pp. 62-67, doi: 10.1109/HST.2015.7140238.
- [15] I. Alshaer, A. Al-kaf, V. Egloff and V. Beroulle, "Inferred Fault Models for RISC-V and Arm: A Comparative Study," *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Didcot, United Kingdom, 2024, pp. 1-6, doi: 10.1109/DFT63277.2024.10753562.
- [16] M. A. Elmohr, H. Liao and C. H. Gebotys, "EM Fault Injection on ARM and RISC-V," *2020 21st International Symposium on Quality Electronic Design (ISQED)*, Santa Clara, CA, USA, 2020, pp. 206-212, doi: 10.1109/ISQED48828.2020.9137051.
- [17] Z. Liao, F. Bruguier and P. Maurine, "Body Bias Injection on the FLASH Memory Accelerator of a 32-Bit Microcontroller," *2025 IEEE 31st International Symposium on On-Line Testing and Robust System Design (IOLTS)*, Ischia, Italy, 2025, pp. 1-7, doi: 10.1109/IOLTS65288.2025.11117129.
- [18] I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle, P. Maistri. Microarchitectural Insights into Unexplained Behaviors under Clock Glitch Fault Injection. *22nd Smart Card Research and Advanced Application Conference (CARDIS 2023)*, Nov 2023, Amsterdam, Netherlands. pp.1-20.
- [19] C. O'Flynn and Z. (David) Chen, "ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research," in *Proceedings of the International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*, E. Prouff, Ed., Lecture Notes in Computer Science, vol. 8622, pp. 243-260, Springer, 2014, Paris, France.
- [20] SiFive, SiFive E31 Core Complex Manual v21G0, 2021. [Online]. Available: https://starfivetech.com/uploads/e31_core_complex_manual_21G1.pdf. [Accessed: Jan. 16, 2026].