# CRAFT: Characterizing and Root-Causing Fault Injection Threats at Pre-Silicon

Arsalan Ali Malik
North Carolina State University
Raleigh, USA
aamalik3@ncsu.edu

Harshvadan Mihir
North Carolina State University
Raleigh, USA
hmihir@ncsu.edu

Aydin Aysu
North Carolina State University
Raleigh, USA
aaysu@ncsu.edu

## Abstract

Fault injection attacks (FIA) pose significant security threats to embedded systems as they exploit weaknesses across multiple layers, including system software, instruction set architecture (ISA), microarchitecture, and physical hardware. Early detection and understanding of how physical faults propagate to system-level behavior are essential to safeguarding cyberinfrastructure.

This work introduces CRAFT, a framework that combines pre-silicon analysis with post-silicon validation to systematically uncover and analyze fault injection vulnerabilities. Our study, conducted on a RISC-V soft-core processor (cv32e40x) reveals two novel vulnerabilities. First, we demonstrate a method to induce instruction skips by glitching the clock (single-glitch attack), which prevents critical values from being loaded from memory, thus disrupting program execution. Second, we show a technique that converts a fetched legal instruction into an illegal one mid-execution, diverting control flow in a manner exploitable by attackers. Notably, we identified a specific timing window in which the processor fails to detect these illegal control-flow diversions, allowing silent, undetected corruption of the program state.

By simulating 9248 FIA scenarios at pre-silicon and conducting root-cause analysis of the RISC-V pipeline, we trace the faults to a previously unreported vulnerability in a pipeline register shared between the instruction fetch and decode stages. Our approach reduced the search space for post-silicon experiments by 97.31%, showing pre-silicon advantages for post-silicon testing. Finally, we validate our identified exploit cases on real hardware (FPGA).

## CCS Concepts

• **Security and privacy** → *Hardware attacks and countermeasures.*

## Keywords

Fault injection attack (FIA), Clock glitch, RISC-V, FPGAs

**ACM Reference Format:**
Arsalan Ali Malik, Harshvadan Mihir, and Aydin Aysu. 2025. CRAFT: Characterizing and Root-Causing Fault Injection Threats at Pre-Silicon. In *Hardware and Architectural Support for Security and Privacy 2025 (HASP 2025), October 19, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3768725.3768726


ACM ISBN 979-8-4007-2198-4/25/10
https://doi.org/10.1145/3768725.3768726

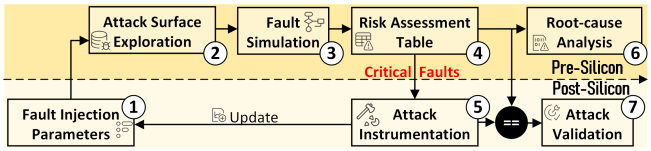


**Figure 1: CRAFT follows a seven-step process: ① select fault parameters, such as glitch width and glitch offset, based on post-silicon capabilities; ② explore the attack surface by analyzing the target application and identifying instructions of interest; ③ inject faults in pre-silicon (simulation) to identify failing instructions; ④ compile a risk assessment table (RAT) to classify and prioritize critical faults; ⑤ use RAT to execute an attack on post-silicon; ⑥ perform root-cause analysis on failing instructions and update fault parameters to improve accuracy for future iterations, and ⑦ validate pre-silicon attack results against post-silicon outcomes to confirm the vulnerability.**

## 1 Introduction

Fault injection attacks (FIAs) deliberately introduce faults into a system to alter its behavior, exploiting, *e.g.*, timing characteristics of designs to induce malfunctions. An attacker may have various goals, such as (a) bypassing security mechanisms to gain unauthorized access to sensitive resources [3, 22]; (b) triggering errors during cryptographic computations to reveal encryption key [17, 38]; (c) disrupting critical system functionality, causing failures in IoT applications [5, 10, 19]; (d) injecting faults in embedded firmware to execute unauthorized instructions [23], and (e) introducing (data) errors to undermine the reliability [2, 6]. These attacks can target different layers of the system stack—from software down to hardware—and exploit the gap between how systems are designed for normal conditions vs. how they operate under physical stress.

Designers previously relied on post-silicon analysis to apply FIA countermeasures. However, rising attack sophistication and hardware complexity now necessitate a multi-stage approach across the design process. The interconnected nature of modern systems demands comprehensive protection throughout the development lifecycle, from initial design to final implementation. Therefore, to successfully defend against FIAs, vulnerabilities must be analyzed and caught at multiple stages of the design process. Pre-silicon analysis (simulation or emulation using RTL/netlist) identifies weaknesses early, allowing fixes before fabrication [11, 16, 25, 33]. However, without physical testing, real-chip fault appearance remains uncertain. Conversely, post-silicon testing validates exploits on real hardware but often treats the processor as a 'black box,' hindering root-cause tracing within the design.

While prior works aim to inject various types of faults [26, 28] or mitigate them by addressing their symptoms [24], limited research has focused on identifying the root causes of FIA vulnerabilities across pre- and post-silicon stages [16, 37]. Furthermore, a key limitation of these works is their emphasis on analyzing the effects of faults, such as instruction skips or data corruption, while providing

insufficient attention to investigating the underlying causes of fault generation. Understanding why these faults occur across multiple layers—like the instruction set architecture (ISA), microarchitecture, and physical hardware—remains critical to designing effective countermeasures and uncovering novel (undocumented) exploits.

Figure 1 illustrates our proposed framework, CRAFT, which integrates pre-silicon fault analysis with post-silicon fault validation. CRAFT employs a seven-step process to identify, characterize, and root-cause vulnerabilities, focusing on critical pipeline control flow faults like instruction skips or unintended jumps [18]. The proposed framework's principles, including clock glitch characterization, are fundamentally applicable across different technologies (ASIC/FPGA). Using CRAFT, we conducted an extensive case study on the open-source RISC-V processor (cv32e40x), uncovering a new clock-glitch attack vector and its root cause. CRAFT systematically leverages insights from pre-silicon analysis to guide post-silicon testing, focusing on faults that affect pipeline control flow—such as instruction skips or unintended jumps. We trace how a single glitch propagates through the pipeline by conducting 9248 fault injection simulations at the post-synthesis netlist level and reproducing a subset on real hardware. This end-to-end analysis identifies which faults occur, why they happen, and where to implement hardware defenses.

The key contributions of this work are the following:

- **RISC-V instruction vulnerability characterization.** We inject faults into eight representative RISC-V instructions from an embedded neural network workload through each pipeline stage. We rank instruction vulnerability, with the highest percentages representing the most vulnerable instructions and stages, to produce a risk assessment table (RAT) that highlights at-risk areas.
- **Discovery of novel fault scenarios.** We identify four exploit cases where precise clock glitches redirect program flow or corrupt results. These include a novel way to induce instruction skips, preventing critical operations. CRAFT converts legal instructions into illegal ones on the fly, bypassing detection. We also demonstrate cases when the processor fails to raise an illegal instruction exception, allowing the attack to remain undetected.
- **Root-cause analysis and verification.** We trace and root cause the silent illegal instruction vulnerability to a specific pipeline register (IF/ID stage) and its interaction with the RISC-V compressed instruction decoder. We confirm how corrupted bits cause erroneous program counter (PC) updates and missed exception flags by analyzing simulation waveforms and RTL code. To the best of our knowledge, this is the *first* report of this vulnerability in the cv32e40x core.
- **Pre- to post-silicon validation of attacks.** Using insights from pre-silicon analysis, we conduct and verify targeted clock-glitch experiments on the FPGA implementation of the same RISC-V core. Our approach cut hardware experiment search space by 97.31%, proving design-time (pre-silicon) insights' value for post-silicon testing.

Our work presents a comprehensive approach to uncovering and understanding fault injection threats. By bridging pre-silicon analysis and post-silicon validation, CRAFT provides system designers with a template for identifying hidden hardware vulnerabilities before they can be exploited, guiding the development of effective defenses. While our case study centers on a specific RISC-V core and a subset of instructions, the framework is generalizable to other processors and larger instruction sets.

## 2 Background

This section outlines the key concepts relevant to our work and formalizes the threat model assumed in this work.

### 2.1 The Research Gap

Although a growing body of research has aimed to analyze and mitigate fault injection threats [15, 28, 32, 39], two critical limitations persist for *clock glitch attacks*. First, these works either focus on pre-silicon [7] or post-silicon [12, 27, 36, 37] but rarely on both [11, 16]. Both approaches are crucial, as pre-silicon analysis simulates internal intricacies and identifies root causes, while post-silicon validation confirms that the identified vulnerabilities persist in the final product. For example, post-silicon characterization [37] can identify the effects of the faults and the attack parameters needed to achieve these effects, but it cannot identify which paths in the circuit are causing these faults, which are needed to build low-overhead defenses or eliminate issues in the current design.

Commercial emulation platforms, such as Synopsys ZeBu and Cadence Palladium, excel at high-speed functional verification and general system-level emulation [4, 30]. However, they do not primarily support the fine-grained, precise timing fault injection characterization critical for discovering subtle hardware vulnerabilities [1]. Moreover, their abstraction level lacks the granularity necessary for systematically exploring specific timing windows or performing root-cause analysis of security-critical physical faults [7].

The closest works to our proposal conduct both pre- and post-silicon fault characterization of RISC-V and MSP430 ISA, respectively [11, 16]. Kazemi *et al.* conducted pre-silicon analysis using a C++ cycle-accurate model of processor behavior, which limits them to certain standard libraries and high-level functions. By contrast, our work identifies and tracks the traversal of faults from critical circuit elements to the application layer. FaultDetective investigates hardware-level faults by observing their manifested effects at the software level [16]. However, it relies on a redundant microcontroller design in lock-step[1] and requires scan registers to observe internal states, limiting its scope and applicability.

### 2.2 Impact of Clock Glitch

A clock glitch attack deliberately and temporarily disrupts the clock signal, causing a misalignment between clock edges that corrupts the data being processed. In such an attack, an adversary introduces a brief pulse or delay into the clock signal, processing data earlier or later than intended. The resulting timing disturbance can lead to outcomes ranging from simple data misalignment to severe data corruption or even system control failures [37]. Two parameters define the nature of a clock glitch: glitch offset and glitch width [37]. Collectively, these parameters determine how the glitch affects the system's timing and functionality [38].

[1]In a lock-step configuration, two or more processors (or cores) execute the same instructions simultaneously and in parallel, cycle by cycle.

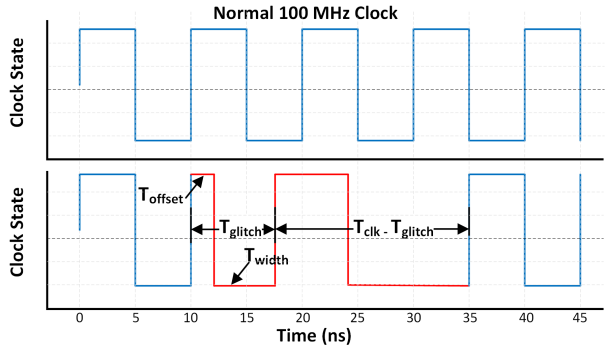


**Figure 2: Illustration of a 100 MHz clock and a clock with a glitch. The top graph shows the regular clock signal with a stable period of 10 ns, while the bottom graph illustrates a clock glitch, where the positive edge is advanced, resulting in a shortened clock period. This disruption can lead to timing violations and data corruption in digital circuits.**

**Glitch offset ($T_{offset}$)** defines the point in the clock cycle where the glitch is introduced, as shown in Figure 2. A glitch occurring near the rising or falling edge of the clock can delay or advance transitions, misaligning data latching and clock edges. Depending on the timing, this disruption may affect different parts of the circuit, leading to data corruption in registers or incorrect state transitions.

**Glitch width ($T_{width}$)** describes the duration of the clock disruption (see Fig 2). In a clock glitch attack, reducing the glitch width increases the intensity of the fault[2]. The duration of the glitch influences how long the circuit remains unstable. This timing disturbance can lead to anything from minor data misalignment to severe data corruption or even a total failure of system controls. While longer glitches increase the likelihood of unintended behavior, not all glitch settings necessarily lead to data corruption.

## 2.3 Threat Model

We conduct FIA on an AI/ML system by inserting clock glitches during the inference process of an embedded neural network as our test case [21]. While traditional glitching attacks have primarily targeted cryptographic systems and PIN verification, applying fault injection to AI/ML systems is a lucrative target because, unlike cryptographic implementations, AI/ML systems are particularly vulnerable due to their complex designs and noisy execution traces [6, 28]. Moreover, as the reliance on AI hardware accelerators grows, it introduces new security challenges and opportunities, not present in traditional cryptographic systems [35].

We follow the conventional assumptions in clock glitch attacks [2, 3, 24, 25, 37, 38]. Specifically, we assume that the attacker has physical access to the circuit's clock signal. This access allows them to execute a *single-glitch* attack with precise control over its width and offset. Using a phase-locked loop (PLL) [15], the attacker can dynamically generate and modify glitch parameters, enabling precise manipulation of the clock signal during runtime. In the context of AI/ML applications, the attacker aims to inject glitches during critical operations to disrupt the program's control flow, inducing data corruption or system misbehavior. This ultimately compromises model functionality and accuracy through carefully crafted perturbations. While we assume in-person manipulation, a sophisticated attacker may also tune these parameters remotely by updating FPGA firmware, further elevating the risk of such attacks.

[2]"Fault intensity refers to the level of physical stress exerted on the microprocessor hardware, pushing it beyond its standard operating limits." [37]

# 3 The Proposed Framework: CRAFT

Prior works predominantly focused on post-silicon experiments, targeting applications, such as the final round of AES encryption, and used fault sensitivity analysis to evaluate cryptographic software vulnerabilities [37, 38]. By In contrast, we introduce CRAFT, a framework for systematically discovering and identifying the root causes of vulnerabilities in a RISC-V softcore processor. CRAFT achieves this by combining pre-silicon analysis with post-silicon validation, using targeted clock glitches. CRAFT assesses the effectiveness of these clock glitches on the RISC-V processor through a seven-step process, as shown in Figure 1.

## 3.1 Fault Injection Parameters

CRAFT begins by selecting the $T_{offset}$ and $T_{width}$ values informed by post-silicon PLL capabilities. This step aims to select coarse-grained parameters[3] that induce timing violations, which are representative of potential fault scenarios, thereby reducing the search (post-silicon) space. The selection process is guided by post-silicon factors such as the onboard PLL capabilities, hardware constraints, operating conditions (temperature/voltage), clock skew, and clock jitter. The careful selection of these parameters helps achieve clock glitch effects such as pulse addition, cycle skipping, duty cycle change, and phase shift [9, 20, 31]. By systematically exploring these parameters, CRAFT corrupts instruction(s) while avoiding system instability/crashes.

## 3.2 Attack Surface Exploration

CRAFT explores the attack surface by analyzing the target application to identify possible vulnerable instructions and execution points. To this end, CRAFT conducts static code analysis to locate sensitive instruction sequences, such as neural network inference layers, activation function computations, or matrix multiplication operations during model inference that translate to instructions such as jumps, branches, loads, and so on. Through this analysis, CRAFT generates a list of candidate instructions that represent the most promising targets for the fault injection.

## 3.3 Fault Simulation

The pre-silicon simulation phase enables CRAFT to scrutinize parameters before reenacting them on hardware. Once CRAFT identifies instructions of interest, it begins fault injection to validate their susceptibility. CRAFT uses post-synthesis netlist simulations to test fault effects on the target instruction. CRAFT adjusts $T_{offset}$ and $T_{width}$ parameters within the simulation to identify all scenarios causing instruction corruption while avoiding system-wide failures (e.g., processor reset). This strategy helps CRAFT identify instructions exhibiting critical faults[4].

**Critical fault.** CRAFT recognizes a fault as critical if it manifests in one of the following ways:

- **Instruction skip.** The instruction is skipped entirely or its result is corrupted (e.g., for a load instruction, its old value is

[3]Fine-tuning of these parameters occurs in the attack instrumentation step.

[4]During our experiments, we observed eight distinct categories of fault behavior. However, in this study, we focused only on critical faults.

retained, or a new corrupted value is stored in the destination register when the instruction retires).
- **Program counter (PC) redirection.** The PC redirects to an incorrect address, disrupting program flow (e.g., for a jump instruction, a corruption in the next PC value computation leads to faults in subsequent PC calculations).

**Non-critical fault.** Clock glitches can freeze the CPU, suspending it in an undefined state [29]. Recovery from such states often requires a CPU reset. While harmful (denial of service), such crashes are less useful for stealthy exploitation, as they are immediately noticeable and do not directly provide the attacker with stealthy control or knowledge. CRAFT classifies these faults as non-critical due to limited application and does not explore them further.

### 3.4 Risk Assessment Table

After the simulation phase, CRAFT compiles a risk assessment table (RAT) to prioritize the most critical vulnerabilities. This table helps guide efficient and targeted attacks by capturing the following attributes(in percentage):

- **Instruction Type.** The category of the instruction (e.g., arithmetic, logic, control flow).
- **Priority Score.** A percentage score that represents the number of instructions that exhibited critical faults among the total number of instructions evaluated.

The RAT helps CRAFT prioritize instructions for targeted post-silicon fault injection. By quantifying fault reproducibility, RAT enables CRAFT to focus on instructions with high impact and reproducibility. This makes the RAT a valuable utility for two key groups: *attackers*, who can pinpoint vulnerable instructions at specific pipeline stages, and *system designers*, who can proactively address these vulnerabilities to enhance the processor defenses.

### 3.5 Attack Instrumentation

Insights from pre-silicon simulations (captured in the RAT) guide the selection of glitch parameters for targeted and efficient post-silicon experiments. CRAFT replicates these parameters to execute fault injections and monitors the system behavior using an on-chip debugger. An attack is considered successful when it produces the expected fault outcomes, such as instruction skips, corrupted memory values (e.g., failed loads), or unauthorized PC redirection.

### 3.6 Root-Cause Analysis

CRAFT is used to conduct a root-cause analysis to understand why specific instructions fail in the presence of a fault. This analysis helps trace the fault's origin to underlying microarchitectural events, such as shared pipeline registers or instruction decoding errors. Identifying patterns in the failing instructions then guides the refinement of glitch parameters. When recurring failures are observed, the specific glitch width or offset are flagged as the fault's trigger. Ultimately, this phase also provides insights into potential countermeasures for mitigating the identified vulnerabilities.

### 3.7 Attack Validation

Finally, CRAFT validates its pre-silicon attack results by repeating the experiments on actual hardware and comparing the outcomes. This process confirms that the vulnerabilities identified in the simulated environment (pre-silicon) also exist in real-world conditions (post-silicon), providing empirical evidence that the target is vulnerable in both pre- and post-silicon stages.

**Table 1: Risk assessment table (RAT) displaying the total percentage of faults observed across the four pipeline stages of the RISC-V processor, with the sum of all cells equaling 100%.**

| Instruction | Total Faults Observed (%) | | |
|---|---|---|---|
| | IF/ID | ID/EX | EX/WB |
| `c.addi` | 0.40 | 5.24 | 6.45 |
| `auipc` | 0.0 | 0.0 | 2.03 |
| `JAL` | **14.92** | **13.31** | **6.85** |
| `bne` | 2.42 | 1.61 | 3.23 |
| `bge` | 1.61 | 6.05 | 1.21 |
| `c.lwsp` | 1.61 | 0.81 | 2.82 |
| `c.mv` | 0.0 | 0.81 | 0.0 |
| `lw` | **11.29** | **9.27** | **8.06** |

## 4 Case Study: Targeting AI/ML Applications

We now demonstrate CRAFT's effectiveness with a case study on AI/ML applications.

### 4.1 Fault Injection Parameters

We used Vivado 2020.2 for pre-silicon instrumentation, and deployed the (same) cv32e40x soft-core processor on the Xilinx Kintex-7 XC7K160T FPGA (SAKURA-X) for post-silicon verification. To demonstrate CRAFT's effectiveness, we targeted an embedded neural network inference application [21]. CRAFT tested $17 \times 17$ distinct clock glitch configurations, varying glitch offset ($T_{offset}$) and glitch width ($T_{width}$) between 0.278ns and 8.89ns, with a step size of 0.5ns[5]. This configuration was broad enough to induce various timing violations—covering a number of fault scenarios—while maintaining efficiency during multiple fault injection experiments.

### 4.2 Attack Surface Exploration

We selected eight instructions crucial for neural network inference, following well-established paradigms [11, 16]. We compiled the code and analyzed eight carefully chosen instructions critical to the inference process, such as the loop-trip count check (a branch instruction in RISC-V) and memory load/store operations for the weights or biases. We focused on this subset for workload relevance, to ensure coverage of different instruction formats and pipeline behaviors. **However, CRAFT is *not* limited to these eight instructions**; it is scalable and capable of evaluating all instructions. We focused on a subset of instructions to manage experimental complexity; however, no fundamental limitation prevents analyzing more RISC-V ISA instructions except time and resources.

### 4.3 Fault Simulation

For the selected eight instructions, CRAFT tested 9248 glitch configurations (17 glitch offsets $\times$ 17 glitch widths $\times$ 4 pipeline stages $\times$ 8 instructions) to exhaustively identify timing paths that trigger critical faults. CRAFT inserts a single-clock glitch at the precise moment to disrupt the critical timing path of a targeted instruction, thereby inducing execution errors. CRAFT targets each instruction as it moves through individual pipeline stages to achieve three goals: (1) mapping the timing path associated with each instruction at each stage, (2) assessing the instruction's vulnerability within

[5]We set $T_{offset}$ and $T_{width}$ to align with the 20ns clock period for our 50MHz design. This precise timing enables us to detect violations in fault scenarios. We chose 0.5ns for these parameters, considering post-silicon PLL capabilities, signal integrity, hardware limitations, operating conditions (temperature and voltage), clock skew, and jitter. This careful selection helps us observe effects such as pulse addition, cycle skipping, duty cycle changes, and phase shifts. [9, 20, 31].

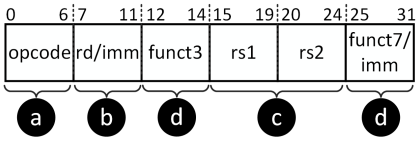


**Figure 3: RISC-V instruction format depicting the opcode, register fields, function codes, and immediate values, highlighting the utilization of various fields depending on the instruction type, *e.g.,* R-type, I-type. Corruption in different fields of instruction can lead to distinct behaviors.**

each stage, and (3) narrowing the post-silicon validation space by shortlisting *only* those glitch parameters that produce critical faults.

## 4.4 Risk Assessment Table

Table 1 lists the eight selected instructions and categorizes the critical faults observed in each pipeline stage. The table displays the total percentage of critical faults observed, with a higher percentage indicating greater vulnerability to clock glitches[6]. We evaluated only these eight instructions, following the well-established paradigm in academic literature [11, 16]. However, this is *not* a limitation because CRAFT is designed to be scalable, enabling it to evaluate any instruction. Using the Table 1 RAT ranking, we hypothesized the causes of faults and validated these hypotheses through our systematic approach, which is detailed in the following subsections.

## 4.5 Attack Instrumentaion

In the pre-silicon phases (Sections 4.3 and 4.4), CRAFT began with 9248 glitch configurations. These phases guide CRAFT to determine (a) the exact timing path of each instruction to induce a timing violation, (b) which instruction(s) is vulnerable and at which pipeline stage, and (c) the impact of glitches on these instructions. Using the insights from these pre-silicon phases, CRAFT identified and narrowed down 248 cases that resulted in critical faults, reducing our focus (for the post-silicon) from 9248 to 248 configurations—achieving a 97.31% reduction.

## 4.6 Root-Cause Analysis

To effectively defend against identified faults, it is essential to understand their root cause(s). Therefore, we conducted an in-depth analysis of these faults, starting by examining them with the assistance of the RISC-V instruction set manual [34]. Figure 3 shows the instruction format of the RISC-V ISA. The format consists of an opcode, a destination register, source registers, and, depending on the instruction type, either a sub-function (for R-type instructions) or an immediate (imm) field (for I-type instructions). A clock glitch that corrupts this format can have four possible outcomes: ⓐ altering the opcode to execute an unintended instruction, causing

[6]Since in RISC-V processor adjacent pipeline stages share the pipeline registers, we show faults as a cumulative sum for each shared stage: IF/ID. ID/EX. and EX/WB.

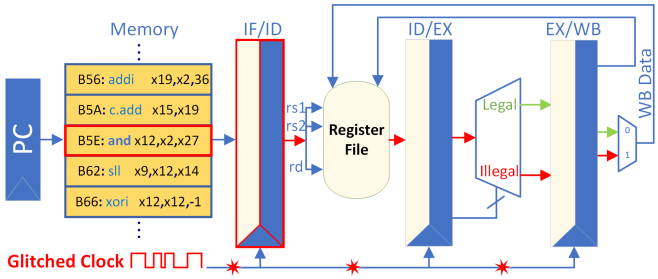


**Figure 4: Visualization of the cv32e40x 4-stage pipeline processor. In the Fetch stage, when the program counter loads the 'and' instruction from memory, a clock glitch disrupts the operation, corrupting the pipeline registers shared between the Fetch and Decode stages, resulting in the misclassification of the legal 'and' instruction as an illegal instruction.**

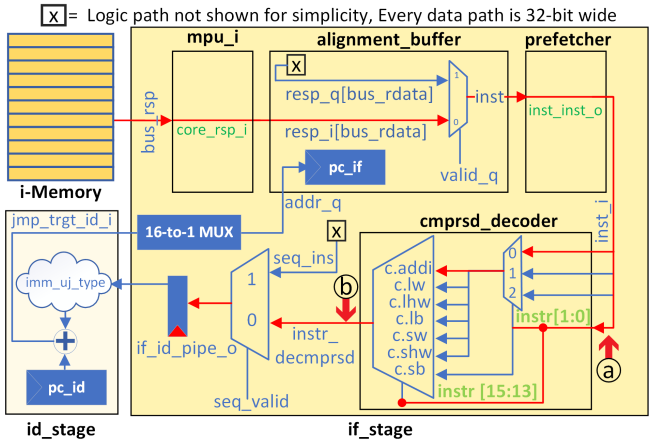


**Figure 5: The root cause of failure in the cv32e40x processor. A clock glitch disrupts the long combinational logic path, causing the input to the compressed decoder module to be latched prematurely (see ⓐ). This leads to misclassifying legal instructions as illegal in subsequent clock cycles.**

incorrect system behavior; ⓑ corrupting the destination register, leading to incorrect data storage; ⓒ corrupting source registers, causing data to be fetched from incorrect locations; or ⓓ affecting the sub-function or immediate fields, resulting in incorrect operation execution or erroneous PC calculations.

To better understand this scenario, consider an example involving the RISC-V processor. Figure 4 illustrates the RISC-V processor pipeline suffering from a clock glitch attack. At every clock cycle, the PC points to the memory address containing the next instruction to fetch. The fetched instruction is stored in the IF/ID pipeline register[7] which is shared between the fetch (IF) and the decode (ID) stage (shown as IF/ID in Figure 4). Consider the scenario where the PC reaches the value of 0xB5E, and an 'AND' instruction is fetched. A precise clock glitch corrupts the instruction stored in the IF/ID pipeline register—our technique in Section 4 has identified this as an instruction that can be corrupted between the IF and ID stage without corrupting other pipeline stages.

Once an instruction reaches the decode-execute (ID/EX) junction, if the instruction conforms to the RISC-V instruction format rules, it proceeds to further execution; otherwise, it is flushed from the pipeline. This flush skips the 'illegal' instruction (newly corrupted), and the PC advances to the next valid instruction. The RISC-V ISA classifies such invalid instructions as 'illegal.' In this example, a clock glitch corrupted the fetched 'AND' instruction, causing its misclassification as 'illegal' and subsequent skip, thus preventing any value from being written to its destination register.

The RISC-V ISA manual specifies conditions for flagging illegal instructions but offers no explicit corrective actions, placing the burden of exception handling on the user. Our analysis of the RISC-V source code showed that the inserted clock glitch corrupted the IF/ID pipeline register [7]. This register, named $if\_id\_pipe\_o$ in the source code, is driven by a long combinational path. The glitch caused a timing violation, corrupting its contents. The $if\_id\_pipe\_o$ register holds the fetched instruction and the control signals required for decoding the instruction in the subsequent ID stage. Therefore, its contents are vital to ensuring proper instruction flow and maintaining synchronization within the pipeline.

[7]In the RISC-V source code, the IF/ID pipeline register is named $if\_id\_pipe\_o$. In this work, we alternatively refer to it by the same name to maintain consistency.

**Table 2: Experimental results showing various glitch offset and width configurations that lead to output corruption. Adjusting the glitch width produces effects such as instruction skipping, loading all zeroes, or selective corruption of the MSBs.**

| Case # | $T_{offset}$ ($ns$) | $T_{width}$ ($ns$) | $T_{glitch}$ ($ns$) | Illegal Flag Raised | Effect Observed |
|---|---|---|---|---|---|
| 1 | 0.833 | ≤2.967 | ≤3.8 | ✓ | Instruction Skip |
| 2 | | 3.067 – 3.567 | 3.901 – 4.400 | ✓ | Data Zeroization |
| 3 | | 3.667 – 4.289 | 4.504 – 5.121 | ✗ | Data Zeroization |
| 4 | | 4.289 – 4.339 | 5.112 – 5.172 | ✗ | Partial Data Corruption |

Recognizing this register's critical role, let us examine how a fault affects the most vulnerable instruction identified by the RAT: jump ('JAL'), with the aid of Figure 5, which illustrates the critical path identified through pre-silicon testing and source code analysis. 'JAL' computes the new target PC by summing the current PC value of the decode stage ('pc_id') with an immediate value ('imm_uj_type') provided by the $if\_id\_pipe\_o$ register. The $if\_id\_pipe\_o$ register derives its input from the 'compressed decoder' module, which handles RISC-V's compressed instruction set extension, converting 16-bit compressed instructions into their 32-bit equivalents.

A clock glitch corrupts the input to the compressed decoder, which stores an incorrectly formatted value in $if\_id\_pipe\_o$. This corruption alters the 'imm_u_type' field and causes the processor to compute the PC incorrectly, jumping to an invalid address. The processor then classifies the resulting instruction as illegal. Without an exception handler, it skips the instruction to continue execution. Our root-cause analysis identifies the IF/ID pipeline register and its preceding combinational logic as the sources of fault injection vulnerabilities. Glitches cause the register to latch incorrect instruction data, leading to control-flow errors (e.g., wrong PC) or data corruption (e.g., incorrect register writes) without triggering system-level alerts.

## 4.7 Update After Attack Instrumentation

As shown in Figure 1, after Step 5, the 'Update' step can be utilized to fine-tune fault parameters. Building on insights from Sections 3 and 4, CRAFT first identified the timing parameters required to induce timing violations and the critical paths vulnerable to clock glitches through root-cause analysis. These insights were then applied using the 'Update' step to reduce the (post-silicon) search space and focus on high-impact fault scenarios. Furthermore, RAT helped reveal that the load ('lw') and jump ('JAL') instructions were the most susceptible, each producing multiple critical faults. This also correlates with their complexity—'lw' performs memory access and write-back while 'JAL' updates the PC. Both require multi-step operations across the pipeline and rely on timing-critical computations *e.g.,* memory fetch, adder for PC. By contrast, simpler instructions like adding upper immediate to PC (auipc) and move (c.mv) produced few or no faults in most stages, reflecting that some operations have more timing slack or simpler logic.

Using RAT as an initial guide, we selected the two most vulnerable instructions, 'lw' and 'JAL', for fine-grained tuning. We (correctly) hypothesized that focusing on these would reveal precise glitch settings, producing new fault behaviors that were missed in our initial sweep. We concentrated on their most vulnerable point as identified in the root-cause analysis—the IF/ID pipeline stage transition when instructions latch into $if\_id\_pipe\_o$ register. We fine-tuned glitch settings around known fault-inducing parameters *e.g.,* when a 3.0ns $T_{offset}$ with 4.0ns $T_{width}$ caused a fault, we tested nearby values (2.9ns, 3.1ns) to uncover subtle behaviors. This

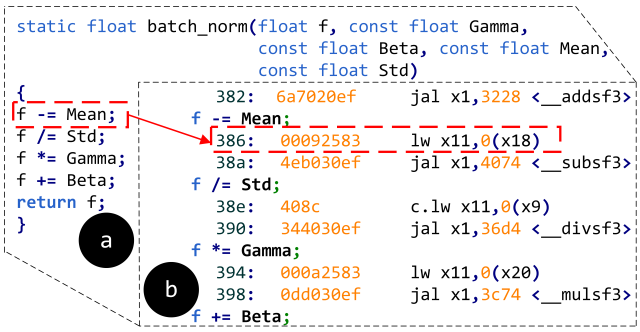


**Figure 6: (a) C-code for MNIST inference and (b) the targeted 'load' instruction in the assembly by CRAFT**

targeted approach, though more manual than our broad sweep, was guided by known weak spots, making it manageable and effective.

By tuning these parameters, we discovered two additional ranges where an attacker could utilize CRAFT's insights to either skip specific instructions or redirect the program flow **without raising an 'illegal' instruction flag** (refer Section 5 for details on the parameters). This can allow sophisticated attackers to cause significant disruptions while leaving the processor unaware of the corruption. The expected outcomes of such an attack include:

- **Case #1:** The instruction is skipped, triggering the 'illegal' flag and the exception handler.
- **Case #2:** The destination register ('rd') is zeroed while triggering the 'illegal' flag and the exception handler.
- **Case #3:** The destination reg ('rd') is zeroed **without** triggering the 'illegal' flag or the exception handler.
- **Case #4:** The destination reg ('rd') is partially corrupted **without** triggering the 'illegal' flag or exception handler.

In **Case #1** and **Case #2**, the 'illegal' flag triggers an exception, redirecting the program to the exception handler address. Without a defined handler, the program skips the instruction and continues execution. The 'illegal' flag indicates the glitch caused the core to latch an invalid instruction format when it captured the decoder output too early. These cases differ in register write-back timing: **Case #2** writes a zero value before the exception takes effect, while **Case #1** skips the instruction completely without writing. The core detects both problems by raising exceptions.

By contrast, the 'illegal' instruction flag is not even triggered in **Case #3** and **Case #4**. This is because in these cases, the *input* to the 'compressed decoder' is latched just in time, avoiding detection as an 'illegal' instruction (refer ⓐ in Figure 5). However, the *output* of the 'compressed decoder' still fails to meet timing constraints and does not update in time (refer ⓑ in Figure 5). As a result, the 'compressed decoder' processes what seems to be valid data but actually executes corrupted information. Table 2 shows glitch parameter ranges that lead to each of these cases[8].

CRAFT illustrates how adjusting the glitch width can toggle between raising an illegal exception or bypassing it entirely. The implications of these fault cases are serious. A silent skip (**Case #3** or **Case #4** affecting a branch or jump) could let an attacker bypass a security check without any log or trace—**leaving the processor completely unaware of the fault**. Silent data corruption in a load

[8]Migrating to a different device may require a one-time re-run of the fault simulation (post-synthesis) step to account for variations, such as changes in netlist and logic delay.

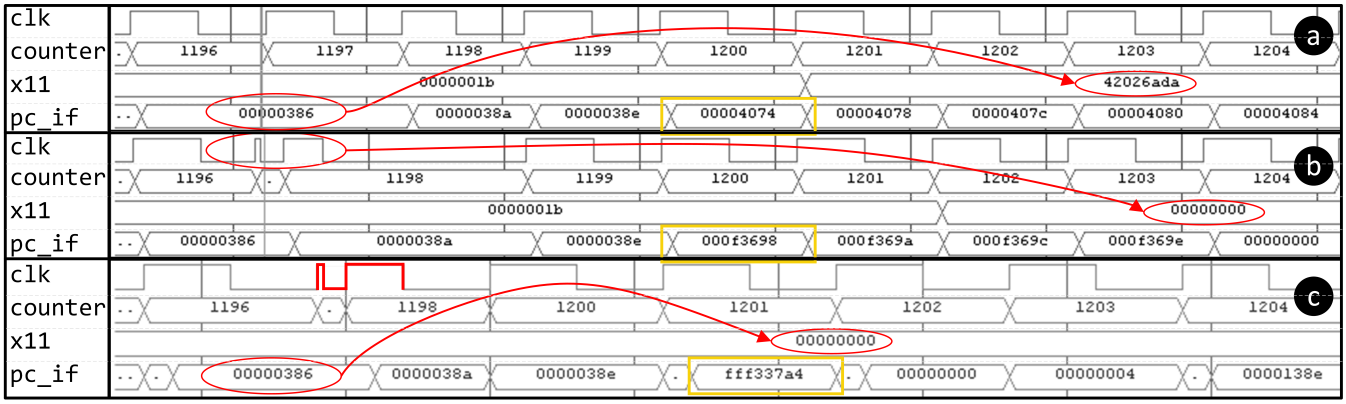


**Figure 7: CRAFT targetting the RISC-V ISA in a neural network inference algorithm [21] is depicted in three scenarios: (a) *without* and *with* a clock glitch in (b) pre- and (c) post-silicon. The first-order effect of the clock glitch attack corrupts the destination register 'x11' associated with the 'lw' instruction, loading all zeroes. The second-order effect corrupts the immediate field of the 'JAL' instruction, disrupting the normal program flow and causing incorrect control flow redirection. In (c), the glitched clock signal is drawn by hand due to the limited sampling capacity of the integrated logic analyzer (ILA).**

(lw) instruction could alter a critical parameter in an algorithm, such as modifying the weights in a neural network or corrupting the input data used for training. Such vulnerabilities pose significant risks to critical applications, including neural networks. A well-timed attack could silently corrupt crucial operations, such as loading weights or biases from memory, potentially leading to undetectable critical misclassifications.

## 5 Attack Validation: Pre- and Post-Silicon

**Test-Platform.** We used Vivado 2020.2 and the cv32e40x soft-core processor for the pre-silicon instrumentation. As part of our case study, we explored the impact of clock glitch-based fault injections on a neural network algorithm running inferences for the MNIST dataset [21]. We ran the RISC-V processor at 50MHz with a timing constraint of 20*ns* specified in the constraint (*.xdc) file. We deployed the cv32e40x soft-core processor on the Xilinx Kintex-7 XC7K160T FPGA (SAKURA-X) for post-silicon verification. Using the RISC-V compiler toolchain we generated the binary file of the inference code, comprising the necessary machine instructions and associated data. A custom controller module was developed to enable data transmission from the host PC to the FPGA.

We used the FPGA's internal PLL to introduce glitches to the clock signal. When the processor core detects the target instruction, it relays a trigger signal to the glitch circuitry. The PLL generates three distinct clock signals: 'CLK0', which serves as the reference base clock at 50 MHz, 'CLK1' and 'CLK2', which are phase-shifted versions of 'CLK0'. We can modulate the glitch parameters by dynamically reconfiguring the phase difference among these clocks. The phase difference of 'CLK1' relative to 'CLK0' and 'CLK2' defines $T_{offset}$ and $T_{width}$, respectively. To verify the effects of clock glitches, we integrated the internal logic analyzer (ILA) into our design to capture, observe, and analyze core signals, such as, the PC value, register file contents, and critical control signals. The ILA monitored and sampled these signals at 300 MHz frequency [9].

### 5.1 Pre-Silicon Verification

We used gate-level post-synthesis netlist to orchestrate CRAFT and inject faults for two reasons: (i) it is the earliest design stage that includes the circuit's timing information. Analysis at this stage allows for identifying potential vulnerabilities and timing-related issues that could be challenging and costly to address later in the design process, and (ii) it enables rapid and detailed analysis with minimal overhead compared to post-silicon evaluations.

Figure 6 (a) and (b) present a snippet of the C code and the corresponding assembly code of the inference algorithm, respectively. The attack target is the 'lw' instruction with a PC value of 0x386. CRAFT induced timing violations by injecting glitches into the nominal clock, adjusting $T_{offset}$ and $T_{width}$ to ensure that the data does not reach the $if\_id\_pipe\_o$ register in time for the next clock edge. Figure 7 (a) shows fault-free execution of the neural network inference model (without clock glitch). A 64-bit counter increments with each rising clock edge to mark time progression. At clock cycle 1197, the 'lw' instruction is fetched with a PC value of 0x386. In a fault-free execution, the instruction completes when the counter reaches 1201, loading the value 0x42026*ada* into the register 'x11', as shown in red.

Figure 7 (b) shows the first-order effect of the clock glitch (marked in red ellipses), showing zeroization of register 'x11' associated with the 'lw' instruction as expected in **Case #2** and **Case #3**. It also depicts the second-order effect (marked in yellow rectangles), where the immediate value in the 'JAL' instruction is corrupted, resulting in incorrect 'pc_if' redirection. Normally, 'JAL' correctly redirects 'pc_if' to offset 0x4074, as shown in Figure 7 (a). Through CRAFT, however, 'pc_if' is incorrectly redirected to 0xF3698 due to premature latching in the 'compressed decoder' module, as explained in Section 4.7, causing cascading faults in the program flow.

### 5.2 Post-Silicon Verification

We were able to successfully reproduce similar effects at the post-silicon level for **Case #1**, **Case #3**, and **Case #4**. Figure 7 (c) illustrates the verification results for **Case #1**, where the attacked load instruction at the PC value 0x386 is skipped (highlighted in red ellipse), raising the illegal instruction flag. The glitched clock signal is drawn by hand due to the limited sampling capacity of the ILA. The glitched clock experiences rapid transitions from $low \leftrightarrow high$ within 0.833ns, significantly shorter than the ILA's sampling period of 3.33ns, resulting in undersampling of the glitch transitions.

For **Case #2**, the fault effects differ between pre-silicon and

[9] Beyond this frequency, the signals and values start to exhibit unreliability, compromising accurate observations.

post-silicon experiments. The 'x11' register loaded non-zero values during post-silicon verification versus all zeroes in pre-silicon. We speculate that the clock glitch that the prematurely latched input to the 'compressed decoder' differs between pre-silicon and post-silicon results. This input consisting of the fetched instruction dictates the final value written to the 'x11' register as explained in Section 4.6. Also, the fetched PC gets redirected to 0xFFF337A4 instead of 0xF3698, as shown in Figure 7 (b). Although the glitch effects are similar under the same settings, we attribute the differences in corrupted values to two factors: (1) critical path delay variations between post-synthesis and place-and-route netlists, and (2) the absence of the fine-grained precision for glitch generation—available in pre-silicon—due to the FPGA's clocking constraints in post-silicon, which made it difficult to replicate the exact glitch settings. We confirmed the existence of silent faults for **Case #3** and **Case #4** in post-silicon.

# 6 Discussions

This section examines CRAFT's implications for RISC-V systems and evaluates potential security countermeasures.

## 6.1 Further Implications of CRAFT

While this study discloses a specific set of vulnerabilities, it is not confined to a single instruction or ISA. In the RISC-V architecture, different immediate fields play crucial roles, *e.g.,* 'imm_i_type' is used for load and arithmetic instructions, 'imm_s_type' encodes store offsets, 'imm_sb_type' is used for branch jump targets, and 'imm_u_type' represents larger constants or address segments. All of these immediate types depend on the $if_id_pipe_o$pipeline register, making them *potentially* susceptible to CRAFT.

## 6.2 Scalability

This study evaluates eight instructions following the well-established paradigm in the academic literature [11, 16]; however, the proposed methodology can be readily scaled to cover the entire RISC-V instruction, providing broader applicability and more comprehensive insights into mitigating vulnerabilities. No fundamental limitation prevents the analysis from being carried out on more RISC-V instructions beyond the investment of time and resources.

## 6.3 Genericness

Our results highlight the effectiveness of CRAFT's as a versatile fault analysis framework. We chose neural network inference as a test case because it represents a real-world, security-critical application where both timing accuracy and data integrity are vital. Analyzing fault effects in this context revealed complex vulnerabilities that could be exploited in practical scenarios. However, the methodology underlying CRAFT is *not* restricted to this specific workload. While CRAFT focuses on hardware-level vulnerabilities, it is adaptable to a wide range of workloads and instruction sets.

## 6.4 Potential Countermeasures

Although generic countermeasures, such as spatiotemporal redundancy, can help address the identified vulnerabilities, their broad implementation can result in substantial overhead. A better approach to defend against our proposed attack could be to embed a lightweight integrity check for the $if\_id\_pipe\_o$ pipeline register to ensure that instructions meet expected formats, reducing the risk of propagating corrupted instructions through the pipeline [8, 14]. Likewise, incorporating handshaking protocols to enhance stability and prevent premature latching on the critical paths identified in this study may ensure consistent state transitions [13]. However, evaluating such defenses lies beyond the scope of this work.

## 6.5 Profiling Pre- and Post-Silicon Discrepancies

To address the deviations between pre- and post-silicon results, a possible solution could be to conduct small-scale device profiling experiments that adjust the glitch offset and width in both environments, measuring the resulting shifts between pre- and post-silicon. This insight could also guide the refinement of pre-silicon simulations based on post-silicon capabilities. However, exploring an automated way to measure deviations between pre- and post-silicon results (by incorporating hardware-specific factors such as clock skew, signal integrity, and FPGA delays into the simulations) is presently beyond the scope of this work.

## 6.6 Device Migration Considerations

We tested CRAFT on a specific RISC-V processor; however, our findings and methodology are adaptable to other processors within the RISC-V family due to their common base architecture. Expanding the analysis to processors with different pipeline configurations offers the chance to tackle new challenges, such as managing variations in pipeline stages, handling data and control hazards, and adjusting to different instruction fetch and execution mechanisms. Additionally, transitioning to processors with deeper pipelines or out-of-order execution units presents an opportunity to refine the pre-silicon characterization approach for more effective handling of unique processing flows.

When migrating to a new hardware platform, such as switching FPGA devices or upgrading processor architectures, it is essential to re-run fault simulations to address design variations. These variations—such as differences in the netlist, logic delays, and timing characteristics—can affect fault injection results *e.g.,* the reported glitch offset and width parameters from one device may not map exactly to a new device due to differences in clock skew or signal integrity. Repeating fault simulations on the new device ensures that previously identified vulnerabilities remain valid and allows for necessary adjustments to mitigate any new risks introduced by the migration. This is an area for future work.

# 7 Conclusion

This work uncovered critical vulnerabilities in RISC-V softcore processors under clock glitch attack, revealing vulnerabilities such as instruction skips, data corruption, and illegal control flow execution. Our pre-silicon fault analysis of the cv32e40x pipeline ranks instructions based on their susceptibility at various pipeline stages. Our findings highlight pivotal attack vectors and emphasize the importance of pre- and post-silicon validation, stricter timing constraints, and improved exception handling. Future research could extend these techniques to other architectures, with an emphasis on developing automated, dynamic countermeasures.

# Acknowledgments

This work is supported by the Office of Naval Research (ONR) grant N00014-23-1-2103. The views, opinions and/or findings expressed

are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. We used GPT-4.0 to assist with editorial improvements.

## References


[1] Raghuraman Balasubramanian et al. 2014. Understanding the impact of gate-level physical reliability effects on whole program execution. In *International Symposium on High Performance Computer Architecture*. IEEE, 60–71.

[2] Thomas Chamelot, Damien Couroussé, et al. 2022. SCI-FI: control signal, code, and control flow integrity against fault injection attacks. In *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 556–559.

[3] Ludovic Claudepierre, Pierre-Yves Péneau, et al. 2021. TRAITOR: a low-cost evaluation platform for multifault injection. In *International Symposium on Advanced Security on Software and Systems*. IEEE, 51–56.

[4] Ruslan Dashkin and Rajit Manohar. 2024. Mixed-Level Emulation of Asynchronous Circuits on Synchronous FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[5] Aakash Gangolli et al. 2022. A systematic review of fault injection attacks on IoT systems. *Electronics* 11, 13 (2022), 2023.

[6] Cheng Gongye and Yunsi Fei. 2024. One Flip Away from Chaos: Unraveling Single Points of Failure in Quantized DNNs. In *IEEE International Symposium on Hardware Oriented Security and Trust*. IEEE, 332–342.

[7] Jacob Grycel and Patrick Schaumont. 2021. Simplifi: hardware simulation of embedded software fault attacks. *Cryptography* 5, 2 (2021), 15.

[8] Richard W Hamming. 1950. Error detecting and error correcting codes. *The Bell system technical journal* 29, 2 (1950), 147–160.

[9] Yan He, Yumin Su, and Kaiyuan Yang. 2024. Design-Agnostic Distributed Timing Fault Injection Monitor With End-to-End Design Automation. *IEEE Journal of Solid-State Circuits* (2024).

[10] Emre Karabulut, Arsalan Ali Malik, Amro Awad, and Aydin Aysu. 2025. THEMIS: Time, Heterogeneity, and Energy Minded Scheduling for Fair Multi-Tenant Use in FPGAs. *IEEE Transactions on Computers* 74, 7 (2025), 2515–2528. doi:10.1109/TC.2025.3566874

[11] Zahra Kazemi, Amin Norollah, et al. 2021. An in-depth vulnerability analysis of RISC-V micro-architecture against fault injection attack. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*. IEEE.

[12] Zahra Kazemi, Athanasios Papadimitriou, et al. 2019. On a low cost fault injection framework for security assessment of cyber-physical systems: Clock glitch attacks. In *IEEE International Verification and Security Workshop*. 7–12.

[13] Oliver Kömmerling and Markus G Kuhn. 1999. Design Principles for Tamper-Resistant Smartcard Processors. *Smartcard* 99 (1999), 9–20.

[14] Miloš Krstić, Stefan Weidling, Vladimir Petrović, and Egor S Sogomonyan. 2016. Enhanced architectures for soft error detection and correction in combinational and sequential circuits. *Microelectronics reliability* 56 (2016).

[15] Wenye Liu, Chang, et al. 2020. Imperceptible misclassification attack on deep learning accelerator by glitch injection. In *57th Design Automation Conference*.

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

[17] Arsalan Ali Malik, Emre Karabulut, Amro Awad, and Aydin Aysu. 2024. Enabling secure and efficient sharing of accelerators in expeditionary systems. *Journal of Hardware and Systems Security* 8, 2 (2024), 94–112.

[18] Arsalan Ali Malik, Harshvadan Mihir, and Aydin Aysu. 2025. Honest to a Fault: Root-Causing Fault Attacks with Pre-Silicon RISC Pipeline Characterization. arXiv:2503.04846 [cs.CR] https://arxiv.org/abs/2503.04846

[19] Arsalan Ali Malik, Anees Ullah, Ali Zahir, Affaq Qamar, Shadan Khan Khattak, and Pedro Reviriego. 2020. Isolation design flow effectiveness evaluation methodology for Zynq SoCs. *Electronics* 9, 5 (2020), 814.

[20] Amélie Marotta, Ronan Lashermes, Guillaume Bouffard, Olivier Sentieys, and Rachid Dafali. 2024. Characterizing and Modeling Synchronous Clock-Glitch Fault Injection. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 3–21.

[21] Bradley McDanel, Surat Teerapittayanon, and HT Kung. 2017. Embedded Binarized Neural Networks. *arXiv preprint arXiv:1709.02260* (2017).

[22] Kit Murdock, David Oswald, et al. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *IEEE Symposium on Security and Privacy*. IEEE.

[23] Onur Mutlu and Jeremie S Kim. 2019. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 8 (2019), 1555–1571.

[24] Pascal Nasahl and Stefan Mangard. 2023. SCRAMBLE-CFI: Mitigating Fault-Induced Control-Flow Attacks on OpenTitan. In *Proceedings of the Great Lakes Symposium on VLSI 2023*. 45–50.

[25] Pascal Nasahl, Miguel Osorio, Pirmin Vogel, Michael Schaffner, Timothy Trippel, Dominic Rizzo, and Stefan Mangard. 2022. SYNFI: pre-silicon fault analysis of an open-source secure element. *arXiv preprint arXiv:2205.04775* (2022).

[26] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. 2015. 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, 62–67.

[27] Satyam Shukla, Md Azam, Kailash Chandra Ray, et al. 2023. An Efficient Fault-Tolerant Instruction Decoder for RISC-V Based Dual-Core Soft-Processors. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2023).

[28] Amit Mazumder Shuvo et al. 2023. A comprehensive survey on non-invasive fault injection attacks. *Cryptology ePrint Archive* (2023).

[29] Chad Spensky, Machiry, et al. 2021. Glitching Demystified: Analyzing Control-flow-based Glitching Attacks and Defenses. In *IEEE/IFIP International Conference on Dependable Systems and Networks*.

[30] Neil Tyler. 2021. Cadence Unveils New 'Dynamic Duo'.

[31] Rajesh Velegalati, Kinjal Shah, and Jens-Peter Kaps. 2013. Glitch detection in hardware implementations on FPGAs using delay-based sampling techniques. In *2013 Euromicro Conference on Digital System Design*. IEEE.

[32] Ingrid Verbauwhede, Dusko Karaklajic, and Jorn-Marc Schmidt. 2011. The Fault Attack Jungle - A Classification Model to Guide You. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 3–8.

[33] Huanyu Wang, Henian Li, Fahim Rahman, Mark M. Tehranipoor, and Farimah Farahmandi. 2022. SoFI: Security Property-Driven Vulnerability Assessments of ICs Against Fault-Injection Attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 3 (2022), 452–465.

[34] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. 2014. The RISC-V instruction set manual, volume I: User-level ISA, v2.0. *EECS Department, University of California, Berkeley, Tech* (2014), 4.

[35] Xiaobei Yan, Chip Hong Chang, and Tianwei Zhang. 2023. Defense against ML-based power side-channel attacks on DNN accelerators with adversarial attacks. *arXiv preprint arXiv:2312.04035* (2023).

[36] Bilgiday Yuce, Nahid Ghalaty, Harika Santapuri, Chinmay Deshpande, and Patrick Schaumont. 2016. Software Fault Resistance is Futile: Effective Single-Glitch Attacks. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*.

[37] Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. 2015. Improving fault attacks on embedded software using RISC pipeline characterization. In *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 97–108.

[38] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. 2018. Fault attacks on secure embedded software: Threats, design, and evaluation. *Journal of Hardware and Systems Security* 2 (2018), 111–130.

[39] Jiliang Zhang and Gang Qu. 2019. Recent attacks and defenses on FPGA-based systems. *ACM Transactions on Reconfigurable Technology and Systems* 12, 3 (2019).