USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

GlitchGlück: Enabling Software Vulnerabilities through Guided Hardware Fault Injection

Zhenyuan Liu, Dillibabu Shanmugam, and Patrick Schaumont,
Worcester Polytechnic Institute

https://www.usenix.org/conference/woot25/presentation/liu

This paper is included in the Proceedings of the
19th USENIX WOOT Conference on Offensive Technologies.

August 11–12, 2025 • Seattle, WA, USA

ISBN 978-1-939133-50-2

Open access to the Proceedings of the
19th USENIX WOOT Conference on Offensive Technologies
is sponsored by USENIX.

GlitchGlück: Enabling Software Vulnerabilities through Guided Hardware Fault Injection

Zhenyuan Liu, Dillibabu Shanmugam and Patrick Schaumont
{zliu12, dshanmugam, pschaumont}@wpi.edu, Worcester Polytechnic Institute, Worcester, MA, USA

Abstract

While many software vulnerabilities are blamed on software bugs, they can also be caused by hardware fault injection. Traditional fault injection methods rely on blind attacks based on simplified fault models, such as instruction skipping. These attacks require exhaustive experimentation across a wide range of fault parameters, with the methodology inferred solely from faulty outcomes, resulting in limited insight into fault impact and an overall inefficient approach. We present GLITCHGLÜCK, a novel approach that combines a tool for simulating hardware-software interactions with a methodology for guiding fault injection. The tool observes the system via scan-chain-accessible states and constructs the Dynamic State Transition Graph (DSTG), a temporal representation of how software instructions trigger interactions with hardware components. By analyzing the DSTG, GLITCHGLÜCK pinpoints fault injection parameters – such as when, where, and what to fault without relying on predefined fault models – thus avoiding the need for an exhaustive fault parameter search. This targeted, data-driven method bridges the gap between simulation and physical fault observation by using scan-chain. GLITCHGLÜCK is demonstrated on a physical OpenMSP430 ASIC chip with scan-chain support, and validated in simulation on PicoRV32 (RV32I) and IBEX (RV32IM) to confirm its applicability across different instruction set architectures and microarchitectures. We assess the effectiveness of several software countermeasures, such as instruction duplication and pin verification, using layout-aware fault simulations to guide fault attacks via clock glitching and laser-induced faults.

1 Introduction

Software vulnerabilities [41] have been extensively studied for decades, with substantial efforts focused on mitigating software-based exploits [13]. However, the execution of software instructions is inherently tied to the processor microarchitecture, which introduces a second layer of vulnerabilities at the hardware level. Numerous studies have demonstrated that even when software vulnerabilities are addressed, both the software and its exploit countermeasures remain vulnerable to hardware fault injection [16]. Various fault injection techniques induce distinct physical effects in the hardware, including timing violations [4], power supply fluctuations [25], laser-induced faults [46], temperature changes [48], electromagnetic (EM) interference [42], rowhammer effects [39] and x-ray radiation effects [3]. Traditional fault injection is commonly performed in a black-box manner, where faults are exhaustively introduced across a broad range of parameters [19]. Because the effect of a fault is difficult to predict, blind attacks require extensive trial-and-error to discover effective fault parameters. This inefficiency reflects a deeper challenge in fault injection: not merely introducing faults, but doing so in a controlled and predictable manner.

Black-box methods lack a systematic strategy to pinpoint the precise timing and location for fault introduction. Even under identical fault conditions, physical noise can lead to variations in fault outcome, making it difficult to reproduce observed effects. Studies have shown that physical effects such as voltage fluctuations or power surges can alter the state of flip-flops, disrupting both control and data paths [27], while clock frequency variations can modulate electromagnetic and voltage behavior, impacting the success of fault injection [26]. In addition, the impact of the fault is assessed by observing only the immediate result, which could fall into one of three cases: a faulty output, a correct output, or a system timeout (no output). However, this assessment does not provide enough insight into the trajectory from hardware fault onset to software effect, leaving critical aspects of the fault's impact unexamined. Thus, it is difficult to understand how fault countermeasures fail, and hence how to improve them.

White-box Approach. To address these limitations, fault injection must be informed by an internal understanding of how faults propagate from hardware to software, enabling more effective fault campaigns. A white-box approach plays a central role in this shift. By leveraging detailed hardware knowledge, such as layout-level information, simulation can analyze fault effects and guide physical fault injection, en-

abling more precise determination of when, where, and what to fault, thereby reducing the complexity of a blind search. For example, the work in [49] adopts microarchitectural fault models to study how faults affect an open-source processor, capturing effects that would be missed by instruction-level abstractions. Previous works have also shown that models such as instruction skipping are inadequate, as they fail to represent the complex and unintuitive fault responses observed on hardware [29, 35]. While the need to access detailed models of the hardware implementation may appear to be a strong requirement, the rise of open-source hardware has made white-box analysis increasingly feasible. Platforms like RISC-V [18] have driven the development of open-source root-of-trust systems [23, 33], expanding the possibilities for studying fault effects and building effective countermeasures. In parallel, the tool flow for custom hardware has been transformed by open-source initiatives, with frameworks like Chipyard [2] and OpenROAD [1] enabling detailed design, simulation, and analysis workflows.

Contribution. We introduce GLITCHGLÜCK, which breaks the quandary of blind fault injection by combining a simulation tool with a methodology for guiding fault injection. The tool observes the system through scan-chain outputs and constructs the Dynamic State Transition Graph (DSTG) – a temporal, graphical representation that captures the interactions between hardware components and software instructions. The methodology then analyzes the DSTG to pinpoint critical fault injection parameters to enable potential software vulnerabilities. The simulation component of GLITCHGLÜCK is used to analyze fault manifestation and propagation, and it is *not* responsible for fault injection itself. The simulated attack parameters are used to guide the physical fault injection. Faults can be observed through the scan-chain which serves as a bridge from physical prototype to simulation, and which avoids the assumptions of fault modeling. GLITCHGLÜCK supports the secure software developer by focusing the analysis on those hardware fault effects that directly enable the software vulnerability. While hardware fault injection is well-established as a method for exploiting software vulnerabilities, existing software development tools do not support the analysis of hardware-induced vulnerabilities. The key contributions can be summarized as follows.

1. We present a tool that performs scan-chain-based state simulation to capture hardware state transitions triggered by software instructions, and visualizes these interactions using a novel data structure, the DSTG.
2. We build a methodology based on the DSTG to identify three critical parameters for target fault injection.
   (a) The *Vulnerable Decision State (VDS)*: A critical state in the software execution flow where modifying it can introduce software vulnerabilities.
   (b) The *Fault Injection State (FIS)*: The state in which the fault should be injected to manipulate the behavior of the VDS.
   (c) The *Fault Injection Timing (FIT)*: The exact cycle at which the fault should be injected into the FIS.
3. We demonstrate GLITCHGLÜCK on a customized OpenMSP430 ASIC (Application-Specific Integrated Circuit) chip with scan-chain support, and confirm its portability in simulation on PicoRV32 (RV32I) and IBEX (RV32IM). We evaluate several software countermeasures using layout-aware fault simulations to guide clock glitching and laser injection attacks.

To support reproducibility, the source code for DSTG generation and analysis results are available online[1].

Outline. This paper is organized as follows. Section 2 provides the necessary background for the paper. Section 3 reviews related work. Section 4 presents the tool aspect of GLITCHGLÜCK. Section 5 outlines the guided attack methodology of GLITCHGLÜCK with an example. Section 6 demonstrates the application of GLITCHGLÜCK on three different processors. Finally, Section 7 concludes the paper.

2 Background

This section covers the attacker and fault models, sane vs. weird machines, and how hardware faults cause software bugs.

2.1 Attacker Model

We consider an attacker capable of performing non-invasive or semi-invasive fault injection techniques, such as clock glitching or laser-induced faults. The injected faults are assumed to be transient, disappearing after a processor reset. We define a faulty state as a system state that deviates through a hardware fault. This deviation can manifest itself as incorrect register values, unexpected control flow, or other abnormal behavior that eventually affects the software layer. GLITCHGLÜCK can observe such a faulty state in two ways (Figure 1). In both cases, the goal is to achieve the highest fidelity in fault capture. One approach is to inject the fault into a white-box simulation, documenting the fault propagation. Low-level design data – such as layout-level timing information – is used to model faults with high accuracy. Another approach is to observe the fault directly on a physical prototype by capturing the low-level state of the system through a scan-chain. A scan-chain is a common test structure used in Integrated Circuit (IC) testing that configures each register as part of a shift register, enabling serial access to flip-flops. Scan-chain synthesis is well supported in ASIC design flows [55, Chapter 15], allowing automatic insertion of scan chains in standard-cell-based designs. We acknowledge that a scan chain itself might be a target of attack [37]; however, scan-chain-based attacks are considered out of scope in this paper.

[1] https://github.com/Secure-Embedded-Systems/woot2025-GlitchGluck/archive/refs/tags/woot25-artifact.tar.gz

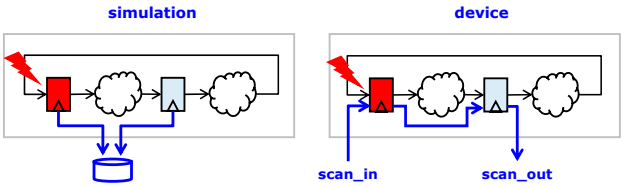

Figure 1: The faulty state can be obtained by simulated fault injection or by measuring the fault through a scan-chain of the processor.

2.2 Classic Fault Model

Fault modeling in software has traditionally relied on high-level instruction models. The framework introduced by Höller *et al.* in [22] present a system-level analysis of software countermeasures by simulating high-level hardware faults, targeting components such as memory cells or instruction execution. These models focus primarily on two types of faults: *instruction skipping* and *instruction replication*.

Instruction Skip. This model results in the processor skipping an instruction that was meant to be executed. The effect of a skipped instruction depends on the type of the instruction. Skipping branch instructions is desirable when the attacker aims to change the control flow of the software, such as changing the outcome of a security-critical decision. Skipping data movement instructions is desirable when the attacker aims to change the data flow of the software, such as for differential fault analysis of cryptographic applications. This fault model has been used in various attacks [11, 34], leading to the development of countermeasures to mitigate its impact [36, 57].

Instruction Replication. In this model, a fault causes the processor to execute the same instruction multiple times. The impact of replicated instructions on a software's execution is more subtle, and it only affects non-idempotent instructions.

While instruction-based fault models simplify fault simulation, they fail to capture the full spectrum of complexities present in real-world hardware. For instance, these models do not account for faults within the microarchitecture itself, such as faults in the data path or control logic. Faults can cause issues like incorrect data fetching, processing, or writing, which are not captured by the instruction skip or replication models. Additionally, traditional models overlook inexplicable fault responses that may occur in actual systems. For example, a hardware fault could trigger a core reset, unexpectedly clearing the processor's state and disrupting execution, or the processor might become muted, halting instruction processing without any apparent cause.

2.3 Sane and Weird Machine

A formal definition of the *sane machine* and *weird machine* is introduced in [9]. The sane machine refers to the execution of a program on a processor according to the programmer's intent, where the system follows a deterministic and predictable path. It represents a fault-free execution flow, where the program behaves exactly as expected, with no interference from external faults. In contrast, the weird machine emerges when faults cause the system to deviate from its intended execution path, transitioning it away from the sane machine. These faults can take many forms, such as bit flips, memory corruption, or control flow errors. For example, the authors in [31] visualize four potential patterns of the weird machine, highlighting its diverse manifestations. As a weird machine, the system's behavior becomes unpredictable, and the software community calls the behavior of the weird machine *unknown* due to the lack of insight into the hardware details. Modeling the weird machine is highly challenging because its behavior depends on system transitions between faulty system states, and those states may never be a part of the original hardware design. The key challenge lies in tracking these unknown states and following their impact on the system's behavior. Accurately modeling the weird machine's behavior would require full knowledge of the hardware implementation. However, with a white-box modeling technique as adopted in this work, the weird machine remains tractable. Starting from a faulty system state, we can compute (simulate) the faulty system next-states and thus reconstruct the emergent behavior of the weird machine.

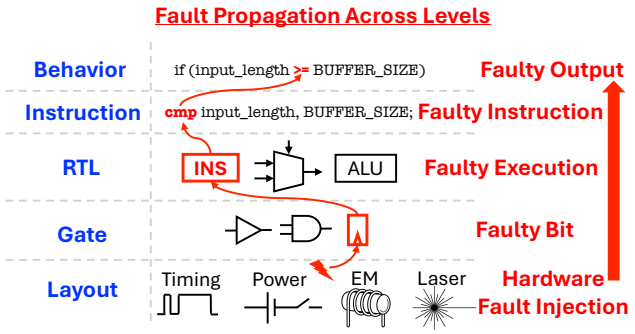

Figure 2: A hardware fault is injected by exploiting the physical properties of the chip, propagating from the layout level through the gate, RTL, and instruction levels, and ultimately leading to a faulty output at the behavior level, which can result in software vulnerabilities, such as a buffer overflow.

2.4 From Hardware Faults to Software Vulnerabilities

In modern computing systems, software interacts with hardware through several abstraction levels, each essential for the proper execution of tasks. Software programs are initially written in high-level programming languages and then compiled into machine-readable instructions. These instructions are fetched, decoded, and executed by the processor's mi-

croarchitecture at the Register Transfer Level (RTL), where control signals manage the movement of data between registers and functional units. The RTL design is further synthesized into gate-level logic, which is then mapped to a physical layout, where the placement and routing of logic gates and interconnections are optimized to implement the chip's functionality. When a hardware fault occurs by exploiting the chip's physical properties to induce disruptions, it propagates in a *bottom-up* approach through the gate, RTL, and instruction levels, ultimately impacting software behavior (Figure 2). Such propagation could cause the processor to execute unexpected instructions, eventually enabling software vulnerabilities. For instance, a buffer overflow vulnerability [8] may arise when a hardware fault disrupts the memory management process, causing data to be written to unintended memory locations. Even when software-level countermeasures against buffer overflow are implemented, hardware faults can bypass these defenses [14, 40]. Consider a buffer overflow countermeasure that performs bound-checking to verify the size of incoming data before copying it into a buffer. If a fault, such as a clock glitch, disrupts the hardware execution and causes the bound-checking to be skipped, the overflow protection can be bypassed, allowing the buffer overflow to occur.

3 Related Work

Numerous studies have explored the challenges of identifying the correct fault parameters for successful fault injection attacks. Elmohr [12], Bozzato [6] and Roscian [45] have conducted fault injections with thousands of trials, each requiring significant effort to fine-tune parameters. Tunstall [51], Selmke [47] and Van Herrewegen [52] have also demonstrated that achieving meaningful fault effects requires extensive fault injection attempts. From an attacker's perspective, injecting a large number of faults is acceptable since the process can be automated, and only a few effective injections are needed to achieve the objective. In contrast, for the designer, each injected fault requires manual effort to trace, verify, and understand its impact on system behavior. For a countermeasure designer, in particular, it is crucial to ensure that the countermeasure functions correctly across the entire fault space. While high-precision fault injection setups – such as low-cost laser attack [24] or focused EM fault injection [44] – achieve more targeted results, they rely on specialized equipment and controlled environments, limiting their portability. In contrast, GLITCHGLÜCK leverages simulation-derived scan-chain data to identify physical fault injection parameters. This white-box approach reduces the complexity of physical setup search, saving time and costs while enabling more efficient fault space exploration and guided fault injection.

The benefits of white-box modeling in fault effect analysis have been demonstrated by several authors. Van Woudenberg *et al.* show that incorporating fault effects from RTL-level simulations into instruction-level simulators improves accuracy [53], and Laurent *et al.* critique conventional fault models for neglecting microarchitectural details, proposing an RTL-to-software fault mapping [28]. Trouchkine also emphasizes that high-level abstraction models obscure underlying fault effects in complex CPUs [50]. These and other studies [30, 43] emphasize that traditional instruction-level fault models overlook hardware-level complexities. Instruction-level simulation frameworks such as `ARCHIE` [17], `ARMORY` [21] and `FaultFinder` [38] enables fault analysis without requiring RTL or physical access to the design. While these tools are effective at highlighting fault vulnerabilities, they operate at a coarser abstraction and lack visibility into the low-level state transitions that give rise to faults. In comparison, GLITCHGLÜCK visualizes software–hardware interactions using white-box, cycle-accurate scan-chain data from layout-aware simulations, then analyzes the resulting graphical data (the DSTG) to extract fault parameters applicable to a physical chip, without dependence on classic fault models – capabilities that higher-level simulation tools cannot offer.

4 GLITCHGLÜCK: Tool Overview

This section presents the tool aspect of GLITCHGLÜCK. We begin by partitioning the scan-chain state to guide DSTG generation in simulation, then outline the key components of the DSTG and present the methodology for its construction.

4.1 Scan-chain State Partition

We define the scan-chain state of a processor as the complete set of its register bits, which reflect the instantaneous system state. For a fault injection to be considered successful at the hardware level, it is crucial to understand how faults injected into the hardware affect the state bits and, ultimately, propagate to observable faults at the software level. However, capturing the entire scan state at once is impractical, as the state space grows exponentially with the number of registers. Even a small core with 1000 flip-flops results in a state space of 2^{1000}. Thus, it is impossible to visualize the entire machine, weird or not, and it is challenging to keep track of even a short sequence of states. The approach to managing this complexity is to focus on the most relevant portions of the scan state – those directly influencing security-critical operations. Since security-sensitive decisions eventually come down to individual instructions, there is always a specific point in the execution flow, a single instruction or transition, that embodies a hardware vulnerability, which in turn enables software vulnerabilities. For example, Yuce *et al.* show that injecting a single fault in a seven-stage pipelined processor can bypass an instruction duplication countermeasure [58].

To address the complexity of tracking the entire scan state, we avoid treating the entire system state as a flat set of registers. Instead, we partition the full state into smaller groups of registers (Figure 3). Each group is treated as a unique state

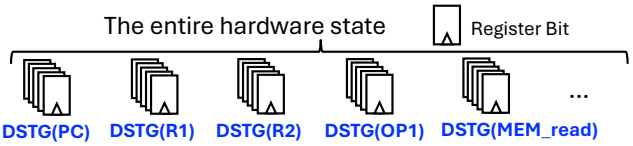

Figure 3: Decomposition of the scan-chain state into individual word-level registers, each with its own distinct DSTG.

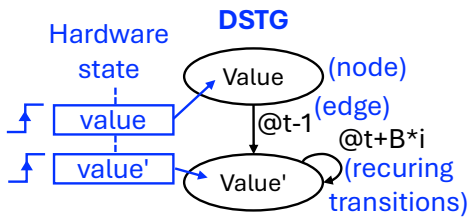

Figure 4: Key components of the DSTG.

space, with each group representing a word-level register and defined by its own set of bits. This approach reduces the overall state space from 2^n to 2^x, where x is the number of bits in each group. By breaking down the system into smaller, more focused segments, the analysis becomes tractable over those registers observed in each group. Each group is then treated as a separate DSTG. We focus on registers within the processor, including both ISA-visible and non-visible ones, such as general-purpose registers and memory buffers. These registers are directly tied to software execution, making them key points where injected hardware faults can manifest and potentially lead to software vulnerabilities. We introduce a specific notation for each DSTG based on the name of the word-level register. For example, DSTG(PC) refers to the DSTG generated from the Program Counter (`PC`), while DSTG(R1) refers to the DSTG generated from the general-purpose register `R1`. This enables tracking of potential vulnerabilities in specific registers and captures a temporal view of system behavior that static descriptions – such as assembly code or Hardware Description Language (HDL) – cannot provide.

4.2 Definition and Key Components

The DSTG is a graph-based data structure that models the evolution of the scan state over time during software execution (Figure 4). Each node in the graph represents a word-level register state at a specific timestamp, defined by the value of the selected register, such as a 1-bit control register, a 32-bit general-purpose register, or a 16-bit memory buffer. The edges of the DSTG are labeled with clock cycle counts, which capture the timestamp when the transition happens. Formally, the DSTG is a graph $G = (V, E, \ell)$ where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and ℓ is an edge labeling function:

$$\ell : E \to \{L \mid L \text{ is an ordered list of timestamps in } \mathbb{N}\}.$$

Edges can be transitioned multiple times, such as during loops or recurring system events. When the interval between consecutive timestamps in ℓ is constant, we denote that constant by the loop period B; that is,

$$t_{i+1} - t_i = B \quad \text{for all } i \geq 0.$$

4.3 Generation Flow

Figure 5 outlines the steps involved in the automated DSTG generation process. The hardware system consists of an integrated combination of the processor, memory, and firmware, operating under a white-box approach. ❶ The software source code is written with predefined input test vectors, then compiled into assembly and a binary image. ❷ The binary image is loaded into the hardware simulation at startup. ❸ Before running the simulation, the scan state of the processor is partitioned inside of the simulation testbench, with each selected word-level register treated as a unique state representation by concatenating its individual bits. At every negative clock edge, the output value of each state is recorded into a state data file along with the corresponding simulation time in both clock cycles and nanoseconds. ❹ The simulation is performed on the gate-level netlist using a standard cell library, with the layout-level Standard Delay Format (SDF) back-annotated to incorporate gate delays and timing information. A Value Change Dump (VCD) is generated to capture the transitions throughout the simulation. ❺ Once the simulation is complete, a time window is selected for plotting the DSTG, concentrating on periods where faults could potentially trigger software vulnerabilities. For instance, if the objective is to fault the buffer overflow mitigation, the window can be defined to cover the entire execution of the string copy function for the DSTG analysis. The start and end PC values are first determined by analyzing the assembly, and the corresponding start and end times are then identified by matching these PC values in the state data file. ❻ Based on the identified time window and recorded state data, the DSTG for each register is constructed and plotted. These graphs are the starting point for the attack methodology explained in Section 5.

Capabilities and Limitations. For large DSTGs, the tool supports (user-driven) dynamic time window selection and state filtering to concentrate on specific execution phases. Users can also isolate individual scan states to reduce memory and rendering overhead. The DSTG is constructed entirely from scan-chain outputs observed during simulation, providing a direct, non-speculative view of system behavior. If dynamic user input is introduced during runtime and influences system state, those effects will be reflected in the DSTG. If the input is never exercised, no corresponding transitions will appear. Similarly, if runtime variations in temperature or voltage are modeled in the simulation and affect gate behavior, the resulting state transitions will also be captured. However,

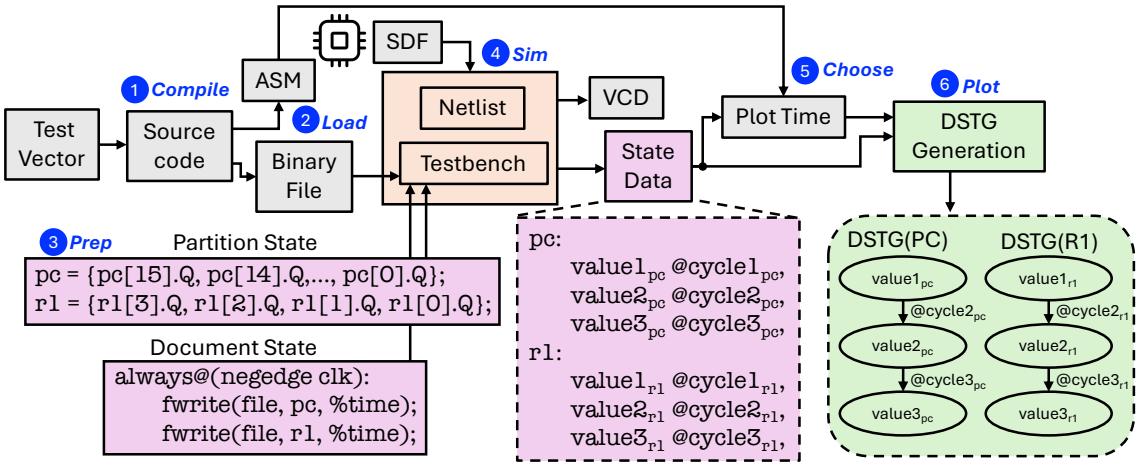

Figure 5: Automated DSTG generation flow.

the DSTG can only capture effects that are modeled and reflected in the simulation data. Physical phenomena – such as analog noise or environmental fluctuations – can be represented in the DSTG only if they are explicitly modeled and influence the scan-chain state during simulation. Otherwise, such effects remain outside the scope of the DSTG.

5 GlitchGlück: Attack Methodology

This section illustrate the methodology of GlitchGlück on a vulnerable string copy operation. By analyzing the DSTG generated from this example, we aim to identify fault injection parameters to exploit the vulnerability.

5.1 String Copy Vulnerability

Listing 1 presents a string copy operation using the function `strcpy()`. The destination buffer has a size of 16 bytes, while the attack string consists of 15 `"A"`s, followed by the memory address of the malicious code (`0xf020`). The size of the input string is designed to overflow the buffer, with the first null byte (`0x0`) preventing the buffer overflow, and the second null byte marks the end of the string. The goal of this attack is to execute the malicious function by bypassing the first null byte through fault injection, enabling the malicious memory address to overwrite the return address of the main function and redirect the program's flow. The malicious function copies the string `"it is broken"` into memory starting at address `0x200`, demonstrating the effects of a successful fault injection. After the attack, the memory contents are dumped in the testbench to confirm the result of the attack.

5.2 Attack Overview

The DSTG-guided fault injection takes a structured, four-step approach to identify hardware-related software vulnerabilities (Figure 6).

Listing 1: String copy operation and a malicious function.

```
void malicious(void) {
    memcpy((uint8_t *)0x200, "it␣is␣broken", 13);
}

char string[19] = "AAAAAAAAAAAAAAA\x0\x20\xf0\x0";

int main() {
    char buffer[16];
    strcpy(buffer, string);
    return 0;
}
```

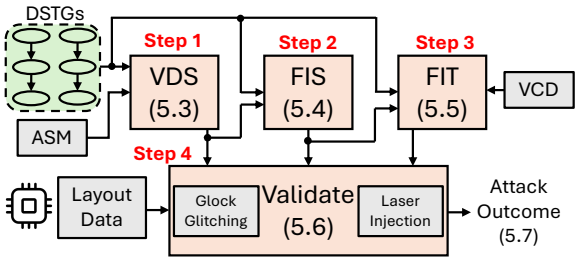

Figure 6: Overview of DSTG-guided fault attack.

Step 1: Identify the VDS. Locate the critical hardware state that, if faulted, can trigger a software-level vulnerability.

Step 2: Select the FIS. Determine the specific state where the fault must be injected to influence the VDS.

Step 3: Specify the FIT. Pinpoint the exact time at which the fault should be injected into the FIS.

Step 4: Validate the attack parameters. Use simulation to confirm that the selected VDS, FIS, and FIT lead to the intended faulty behavior. Leverage layout data to model realistic fault scenarios, and choose an appropriate fault injection method, such as a global technique (e.g., clock glitching) or a localized approach (e.g., laser injection).

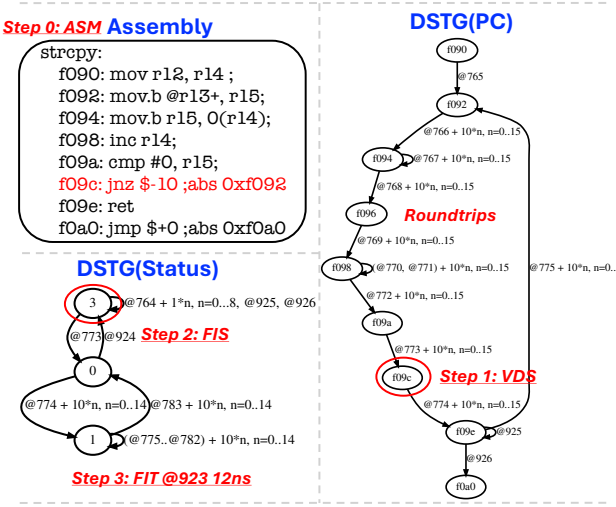

Figure 7: The assembly code for the string copy function, along with the generated DSTG(PC) and DSTG(Status), covers the entire copying operation.

Once the fault is injected and the faulty VDS is reached, the focus shifts from hardware-level concerns to traditional software analysis, treating the injected fault as a software vulnerability and assessing its impact on the software's behavior.

5.3 Step 1: Identify the VDS

Definition. A VDS is a critical state that influences the system's behavior, representing a decision point in the execution flow. This decision point arises when instructions, such as branch operations, determine the next steps in program execution. By manipulating a VDS through fault injection, the execution path can be altered, potentially triggering the software vulnerability. The VDS is typically defined in the DSTG(PC), as it reflects the software program flow through the `PC` register.

Demonstration. The assembly code for the string copy example is shown in Figure 7, and the behavior of this copy loop is visualized in the DSTG(PC). The DSTG(PC) shows a roundtrip from state `0xf092` to `0xf09e`, illustrating the process of copying the string `"A"` 15 times, with the operation exiting at cycle 925. At this point, the first null byte is encountered, causing the function to terminate safely without triggering a buffer overflow. The `ret` instruction at state `0xf09e` is pre-fetched in the DSTG, and it is not executed until the condition is satisfied at cycle 925. The critical decision point where the execution path is determined occurs at state `0xf09c`. If the program continues to `0xf092` (jump taken), a buffer overflow can be triggered. However, if it terminates at `0xf09e` (jump not taken), the operation completes safely. We define the VDS at state `0xf09c`, as this is where the critical decision occurs that can trigger a buffer overflow.

5.4 Step 2: Determine the FIS

Definition. The FIS refers to the state where the fault is injected, and it must have a dependency into VDS. Any modification to the FIS directly influences the behavior of the VDS, which ultimately alters the execution flow. A single VDS may be associated with multiple FISs, but typically, the FIS closest in time to the VDS is selected, as it is most likely to have the immediate impact on the VDS.

Demonstration. The VDS, defined by instruction `jnz` (jump if not Zero) at `0xf09c`, must be taken to trigger a buffer overflow, and this jump condition depends on the value of the `Zero Flag`, which is located in bit 1 of the `Status` register. The DSTG(Status) shows that the `Zero Flag` is set to 1 at cycle 924, which causes the jump to be skipped at cycle 925. If the `Zero Flag` is not set at cycle 924, the string copy operation will continue and eventually overwrite the return address, leading to a buffer overflow. Therefore, a fault is required to disrupt the setting of the `Zero Flag` at cycle 924. The FIS is defined as `Status` register (`0x3`) at cycle 924 to fault the `Zero Flag` transition from 0 to 1. If this transition is disrupted, the `Zero Flag` is likely to retain its previous value of 0, which will alter the decision at the VDS, causing the copy operation to continue and enabling a buffer overflow.

5.5 Step 3: Establish the FIT

Definition. The FIT is the precise moment when a fault is injected, and this timing is crucial to achieve the desired fault impact. To properly influence the VDS, the input value being loaded into the FIS must be faulted, as this value directly propagates to the FIS output, which subsequently drives the VDS decision-making process. In digital hardware systems, the behavior of a register is governed by clock edges. The input value to a register is computed by its driving logic and the updated outputs from its driving registers in preceding cycles. This input data propagates through the datapath and is loaded into the register at the next clock edge, updating the register's output. Thus, the register's output reflects the input value of the previous clock cycle; a trivial but important observation to properly select the FIS.

Demonstration. Since the DSTG captures register output transitions, and the FIS is defined as the state of the `Status` register at cycle 924 (when the `Zero Flag` is updated to 1), the FIT is automatically determined by parsing the VCD file and analyzing the cycles leading up to cycle 924. This analysis identifies cycle 923 as the point where the logic calculates the input value for the `Zero Flag`. The value stabilzation to 1 after a 12ns logic propagation time stabilizes, within a 20ns clock period, and then causes the `Zero Flag` to update in cycle 924. Thus, the FIT is defined at cycle 923, with a 12ns logic propagation time[2] of the `Zero Flag`.

[2] In hardare design, it is common to express the timing margin by *slack*, the difference between the clock period and the logic propagation time. To select

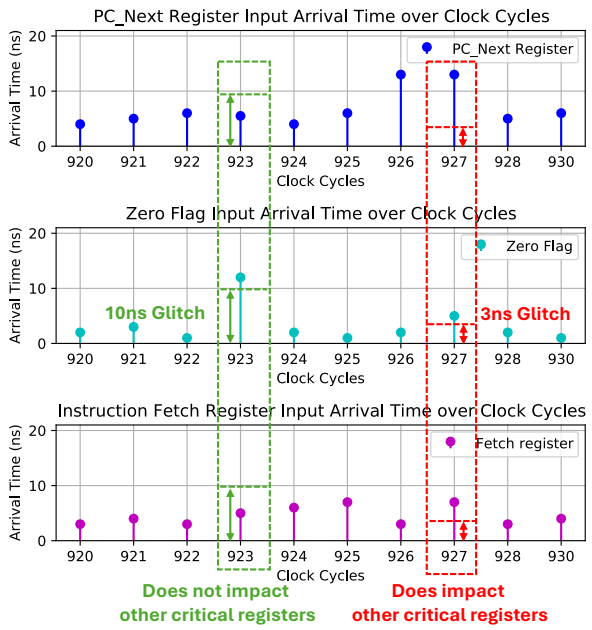

Figure 8: Input arrival times of the `PC_Next`, `Zero Flag`, and `Instruction Fetch` registers across clock cycles 920 to 930 based on the layout data, highlighting how varying glitch widths impact these registers at different times.

5.6 Step 4: Validate the Attack Parameter

Once the VDS, FIS, and FIT are established based on the DSTG, these variables are used to guide the fault attack process. The specific method of attack depends on whether the fault injection is global or localized.

Global Fault Injection. In the case of global fault injection, such as clock glitching, targeting specific bits within the FIS is not feasible because a clock glitch impacts all registers within the design. To influence the register update in the FIS, the glitch must be injected at the cycle corresponding to the FIT. The glitch width must be smaller than the logic propagation time of the FIT to ensure that the fault is applied before the input value delivered to the FIS has fully stabilized. This guarantees that the fault influences the register update and impacts the VDS decision-making process. However, the fault effect from glitching does not uniformly affect every register in the same way. The logic propagation time at a registers' input depends on the number of logic components involved in the computation of that input. This path is different for every register, and it can also change every clock cycle because of datapath reconfiguration in the hardware. Figure 8 illustrates the input arrival time of the `PC_Next`, `Zero Flag`, and `Instruction Fetch` registers across clock cycles 920 to 930 based on the layout-level timing delay. This timing variation emphasizes the importance of carefully selecting the glitch width. For example, a glitch injected into cycle 923 with a width of 10ns will only affect the `Zero Flag` register. On the other hand, if the glitch is injected at cycle 927 with a width of 3ns, it will not only fault the `Zero Flag` register but also affect critical registers such as the `PC_Next` and the `Instruction Fetch` registers. The `PC_Next` register, in particular, plays a key role in determining the next instruction, and any disruption to it could prevent the processor from correctly advancing to the following instruction. Thus, selecting the proper glitch width is crucial for targeted fault injection.

fault injection parameters, the logic propagation time is a more convenient metric.

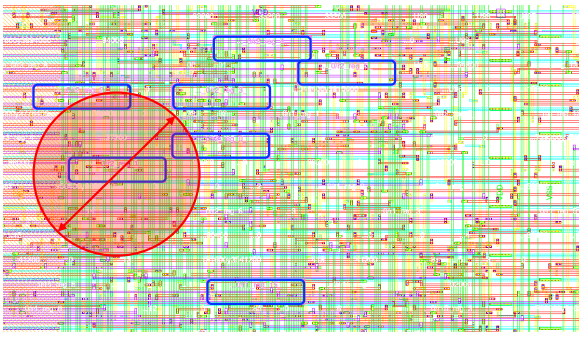

Figure 9: Illustration of a 10μm laser spot size affecting registers (blue rectangles) within a defined range, centered on the target bit, during a localized fault attack based on layout.

Localized Fault. For more localized fault attacks, such as laser injection, precise targeting of specific bits within the FIS is achievable. We discuss the case of a laser attack scenario. Laser illumination creates on-chip photo-currents and IR-drop, and a common outcome is the flipping of the state bit of a register [54]. The laser radius is selected to ensure that all registers within the defined range, measured from the center of each target bit, are affected by the bit-flip during the FIT. The layout data, which provides the XY coordinates of all registers on the die, is used to identify the registers within the defined radius that will be affected (Figure 9). The bit-flip is applied only after the target bit has stabilized, preventing any interference during its transition phase.

5.7 Attack Parameter Simulation Result

In this example, we choose clock glitching as our attack method and we inject a 11ns clock glitch at cycle 923, disrupting the transition of the `Zero Flag` from 0 to 1. After the attack, we examine the memory starting at address `0x200` and find the ASCII string `"69 74 20 69 73 20 62 72 6F 6B 65 6E"` (`"it is broken"`), indicating that the return address was overwritten and the malicious code was executed.

6 Experimental Results

GLITCHGLÜCK is demonstrated on a physical OpenMSP430 ASIC chip using scan-chain-based fault observation, and eval-

Core Type	Processor Cell		Total Cell	Num. of DSTG (Processor Only)
	Register	Gate		
MSP430 [15]	660	4,033	8,081	100
PicoRV32 [56]	2,289	8,861	57,671	360
IBEX [32]	2006	6,698	13,455	192

Table 1: Breakdown of processor register cells, gate cells, total cell counts, and the number of DSTGs generated for processor-only registers in the OpenMSP430, PicoRV32, and IBEX processors.

uated through fault parameter simulations on the PicoRV32 and IBEX to confirm its portability across different processors. This section begins by introducing the simulation setup, followed by a detailed evaluation of each case study.

6.1 Simulation Setup

GLITCHGLÜCK is demonstrated on three different processors: OpenMSP430, a multi-cycle RISC-based processor implemented as a customized ASIC with scan-chain support; PicoRV32, a multi-cycle RISC-V-based processor implementing the RV32I instruction set; and IBEX, a three-stage pipelined RISC-V-based processor implementing the RV32IM instruction set. While OpenMSP430 and PicoRV32 execute a single instruction over multiple clock cycles, IBEX processes instructions in parallel through its Instruction Fetch (IF), Instruction Decode/Execute (ID/EX), and Write Back (WB) stages. Each processor design incorporates various memory-mapped peripherals, such as memory blocks, GPIOs, or coprocessors. All designs are implemented using 180nm standard-cell technology and the layout data is used to extract the XY coordinates of all registers on the die, including their timing properties (SDF).

We exclusively focus on the processor register bits to generate the DSTGs, as these bits are critical for representing system functionality and, if faulted, can potentially trigger software-level vulnerabilities. Table 1 provides a breakdown of the component counts, including processor register cells, processor gate cells, total cell counts, and the number of DSTGs generated from processor-only registers in each core, which account for 14% of the total processor cells in OpenMSP430, 21% in PicoRV32, and 23% in IBEX. Each node in the DSTG represents the state value in hexadecimal.

For each case study, we first use the tool aspect of GLITCHGLÜCK to generate fault-free DSTGs, which serve as the ground truth for identifying the VDS, FIS, and FIT. Then, the methodology aspect of GLITCHGLÜCK is applied to guide fault injection, pinpointing fault attack parameters. A fault simulation is subsequently performed by injecting faults into the testbench to validate these parameters. If multiple faults are required, we generate the DSTGs from the previous fault simulation to guide the generation of the next VDS, FIS, and FIT for each subsequent fault injection. All simulations are executed on a gate-level netlist with back-annotation of the layout-level SDF using ModelSim. Simulations are run on a Xeon Gold 6248 CPU workstation with 384 GB of memory.

Listing 2: Buffer overflow mitigation via input size validation.

```
#define ASSERT(condition)
    if (!(condition)) {
        handle_error(__FILE__, __LINE__);
    }

void handle_error(const char *file, int line) {
    while(1);
}

char string[19] = "AAAAAAAAAAAAAAAA\x22\xf0\x0";

int main() {
    char buffer[16];
    ASSERT(sizeof(buffer) > strlen(string));
    strcpy(buffer, string);
    return 0;
}
```

6.2 Breaking Buffer Overflow Mitigation

Setup. This experiment demonstrates the application of GLITCHGLÜCK to trigger a buffer overflow in a string copy operation that is protected by an input size validation. The countermeasure, a programmer-level defense, employs a custom `ASSERT()` macro (Listing 2) to check if the buffer size exceeds the length of the string. If the check fails, the `handle_error()` function is called, which enters a while loop to mitigate the buffer overflow exploit. The attack string is intended to overflow the buffer and trigger the error-handling loop. The memory address of the malicious function is `0xf022`, and the corresponding C code is in Listing 1. The software is compiled with the `-O0` flag, and the attack simulation is demonstrated on OpenMSP430 using a single-clock glitch. The simulation runs at a frequency of 50MHz, and the DSTG plotting window covers the entire assertion checking.

Step 1: Identify the VDS. The compiled assembly code demonstrates how the assertion checking prevents the overflow, and the DSTG(PC) shows that the countermeasure is effective, with the program entering an error-handling loop at instruction `0xf0a2` (Figure 10). The VDS is identified at instruction `0xf08e`, where the program decides whether to branch to the infinite loop or proceed to the string copy function at `0xf096`, potentially overflowing the destination buffer.

Step 2: Determine the FIS. To trigger an overflow, the assertion check at instruction `0xf08c`, which compares the values of registers `r12` (string input size) and `r13` (buffer size), must pass. The DSTG(r12) and DSTG(r13) reveal that, just before reaching the VDS, the value update of `r12` and `r13` at cycles 919 and 924 result in `r12` holding the value `0x12` (18 in decimal) and `r13` holding `0xf` (15 in decimal).

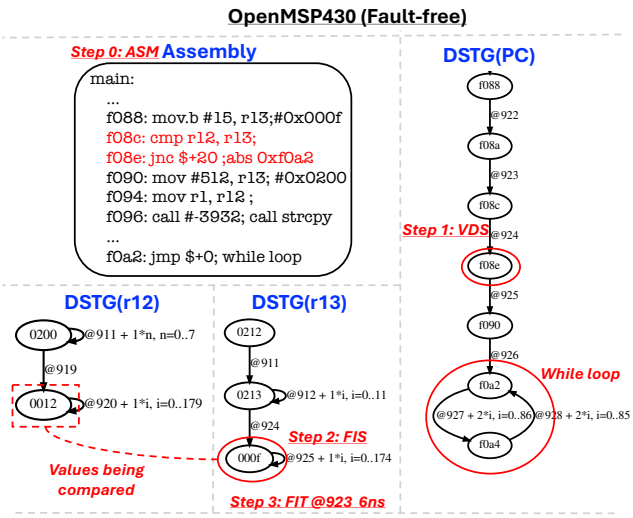

Figure 10: The assembly code for the bound checking function, along with the generated DSTG(PC), DSTG(r12) and DSTG(r13), reflects the corresponding operations.

This configuration causes the assertion check to fail, as the buffer size is smaller than the string input size. However, the DSTG(r13) shows that its previous value was `0x213`, which is larger than the compared value of `0x12` in `r12`. Therefore, if a fault interrupts the update to `r13` at cycle 924, `r13` may not be set to `0xf` as intended. Instead, it could retain its previous value of `0x213`, causing the comparison check to pass and enabling the overflow. Thus, the FIS is identified as the state of `r13` (`0xf`) at cycle 924.

Step 3: Establish the FIT. The FIT is automatically established at cycle 923, where the input value to register `r13` is computing from `0x213` to `0xf`. Analysis of the VCD file shows that four bits of `r13` stabilize within 6ns, while all critical register bits settle no later than 4.7ns, given a 20ns clock period. To disrupt this transition and potentially preserve the previous value `0x213`, the FIT is established at cycle 923 with a logic propagation time of 6ns.

Attack Parameter Simulation. A 5ns clock glitch is injected at cycle 923, causing `r13` to update to a value greater than the expected `0xf`. This disruption prevents the jump from being taken at the VDS, and the malicious code is executed, altering the intended program flow.

Physical Attack Validation. To validate the attack parameters identified during simulation, a post-silicon experiment is conducted on the OpenMSP430 ASIC, fabricated using a 180nm System-on-Chip (SoC) with scan chain support. The scan chain is utilized to monitor the register state before and after fault injection, focusing on whether the fault causes the value of `r13` to exceed `0xf`, triggering an assertion check. The validation process begins by scanning in the scan state at cycle 923, generated from the fault-free simulation, into the physical ASIC. A single glitch pulse is then injected using an FPGA, with glitch widths ranging from 1ns to 10ns, to evaluate the physical chip's response across different attack parameters. After the glitch is applied, the faulty scan state is scanned out to observe the effect of the injected fault on register `r13`. The results in Figure 11 show that for glitch widths above 6ns, `r13` retains its expected value of `0xf`, indicating no observable fault – consistent with the upper bound of the attack parameters derived from simulation. For glitch widths below 6ns, faulty bit flips occur in `r13`, producing values greater than `0xf` and eventually bypassing the bound check. Glitch widths of 1.6ns and 2.7ns – falling below the lower bound of simulation-derived attack parameters – fault critical registers such as the `instruction execute` and `PC` registers, as expected. These high-impact faults result in processor states that deviate from the targeted vulnerability, reinforcing the precision of the simulated fault window. Moreover, across repeated trials, glitching within the attack window exhibits a high degree of determinism, producing the expected faults in `r13`. This confirms that the simulation-guided parameters effectively translate to reliable physical fault injection.

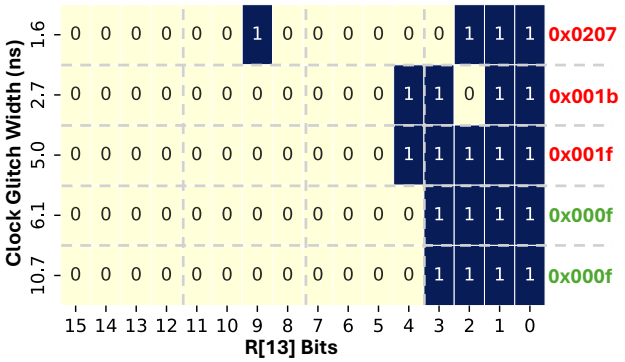

Figure 11: Measured scan-out values in register `r13` after clock glitch injection with varying glitch widths.

6.3 Bypassing Instruction Duplication

Setup. In this experiment, GLITCHGLÜCK is applied to bypass a programer-level countermeasure that implements the instruction duplication techniques proposed by Barenghi et al. in [5]. Listing 3 shows the duplication of a load instruction (`lw`), where a value is loaded from memory into the register `a2`, and the same instruction is repeated in a different register, `a3`. A comparison between these two registers is executed to detect any discrepancies, ensuring that a single fault in either load instruction is not bypassed. If a mismatch is detected, the program enters an error-handling while loop. The software is compiled with the `-O0` flag, and the simulation is demonstrated on IBEX using two glitches. The clock period is 50ns, and the DSTG plotting window visualizes the execution of the three listed instructions.

Listing 3: Load instruction duplication.

```
asm (
    "lw a2, 0(%0)\n"
    "lw a3, 0(%0)\n"
    "bne a2, a3, handle_error\n"
);
```

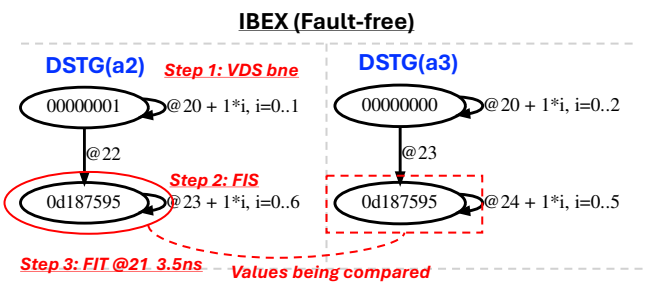

Figure 12: Generated DSTG(a2) and DSTG(a3) based on the simulation of Listing 3, showing the updates of both load instructions in IBEX.

Proposed Approach. Two strategies are considered for bypassing the countermeasure, as proposed in [58]. The first approach involves injecting identical faults into both load instructions, faulting their payloads in the same way. This ensures that both registers receive the same faulty value, preventing the system from entering the error-handling loop. The second approach involves injecting a fault into one of the load instructions, which causes the comparison to fail, followed by another fault to bypass the `bne` check.

However, the authors in [58] found that they could not bypass the countermeasure by injecting two identical faults into a seven-stage pipelined processor using clock glitching. The failure of this strategy can be predicted by DSTG analysis. The DSTG(a2) and DSTG(a3) generated from a fault-free simulation in IBEX demonstrate that the `lw` instruction loads both registers with the random value `0x0d187595` from memory (Figure 12). The registers are updated in the WB stage during clock cycles 22 and 23. Ideally, injecting identical faults should produce the same faulty value in both registers. However, variations in initial register values and fault propagation during execution make the outcomes unpredictable, even with identical fault injections. The DSTG(a2) and DSTG(a3) indicate that register `a2` was previously populated with other values, while `a3` remained unchanged. This implies that even if two identical glitches are injected, there is no guarantee that both registers `a2` and `a3` will receive the same faulty value. Although more localized attacks, such as laser fault injection, can target specific bits, they still require precise knowledge of which bits to flip and may involve modifying multiple bits simultaneously at different locations to achieve the desired effect. Thus, the second approach is explored to overcome this countermeasure.

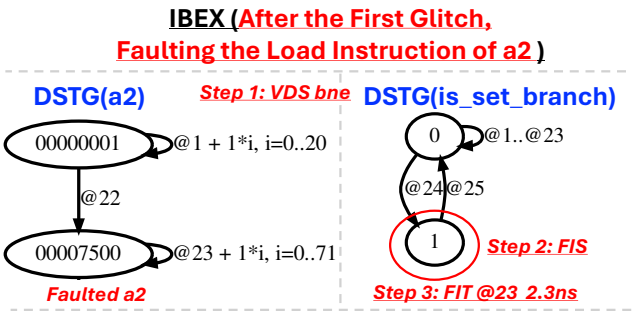

Figure 13: Generated DSTG(a2) and DSTG(is_set_branch) based on the simulation of Listing 3 after faulting the load instruction of `a2` in IBEX.

Step 1: Identify the VDS. The VDS is identified at the branch instruction `bne`, which acts as a checkpoint in the countermeasure, controlling the program flow.

First Glitch: Faulting the Load Instruction of `a2`. A fault is required to disrupt the update of register `a2` to `0x0d187595` at cycle 22 in DSTG(a2), forcing the `bne` check to fail (Figure 12). The FIS is identified as the state of `a2` (`0x0d187595`) at this cycle, and the FIT is automatically determined at cycle 21, with all toggled bits in `a2` stabilizing within 3.5ns and no critical register bits arriving later. A 3ns clock glitch is chosen to disrupt the bit transitions in `a2`, and the resulting DSTG(a2) in Figure 13 confirms the success of the fault, showing that `a2` updates to `0x7500` instead of the expected `0x0d187595`. This fault alters the comparison in `bne`, resulting in the program branching into an error-handling loop.

Second Glitch: Bypassing the Comparison. Immediately after the first glitch, the DSTG(is_set_branch) register is set from 0 to 1 at cycle 24 (Figure 13). This register controls the branch enable in IBEX, triggering the branching into the infinite loop. To bypass this, a second glitch is introduced. The FIS for the second glitch is identified as the state of the `is_set_branch` register (`0x1`) at cycle 24, and the FIT is established at cycle 23, with a logic propagation time of 2.3ns. A 2ns clock glitch is then injected at cycle 23, successfully bypassing the load instruction duplication countermeasure in IBEX, despite `a2` and `a3` holding different values (`0x7500` and `0x0d187595`, respectively).

6.4 Exploiting Pin Verification Protection

Setup. GLITCHGLÜCK is applied to bypass the PIN verification countermeasure implemented in the FISSC toolbox [10]. PIN verification is a security mechanism that compares a user-provided PIN with a stored reference PIN to determine access authorization. If the entered PIN matches the stored value, authentication succeeds; otherwise, access is denied. The toolbox offers multiple PIN verification routines, each with varying levels of protection. We demonstrate

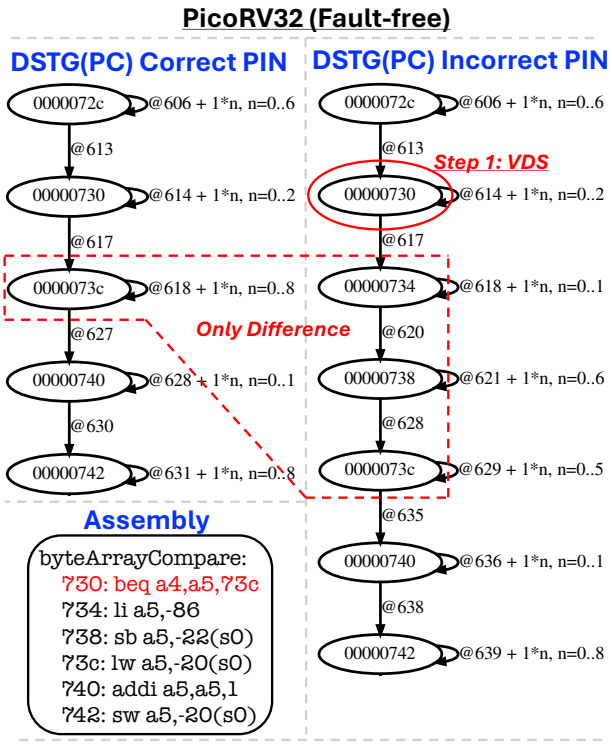

Figure 14: Generated DSTG(PC) for correct and incorrect user PIN verifications on PicoRV32, highlighting the differences in execution flow.

our case study on VerifyPIN5, a hardened implementation designed to resist fault injection attacks, incorporating countermeasures such as hardened Booleans, a fixed-time loop, a step counter, and double-testing. The goal is to exploit faults to bypass the verification process using an incorrect user PIN. Both the user and reference PINs are four digits long. The software is compiled with the `-O0` flag, and the simulation is conducted on PicoRV32 using laser fault injection. The simulation runs at 20MHz, with the DSTG plotting window capturing the entire execution of the PIN verification process.

Step 1: Identify the VDS. To identify the VDS more easily in this example, two separate fault-free simulations are performed: one with the correct user PIN and another with an incorrect user PIN, where one byte differs. The only difference in the generated DSTG(PC) between the correct and incorrect user PINs is shown in Figure 14, which occurs during the execution of the first `byteArrayCompare` function. The decision point occurs at instruction `0x730`, a `beq` (branch if equal) instruction, where the comparison result determines whether the PIN verification succeeds or fails based on the user input. Thus, we define the VDS as the `beq` instruction, where the execution path diverges based on the user PIN.

Step 2: Determine the FIS. To identify the FIS that can influence the behavior of the VDS, it is crucial to understand what is being compared during the `beq` instruction, which distinguishes the decision between the correct and incorrect user PIN. In the PicoRV32 microarchitecture, the values for the branch comparison are first loaded into registers `operand1` and `operand2`. These values in the operand registers are then compared, and the result of this comparison determines whether the branch will be taken or not. Figure 15 shows the generated DSTG(operand1) and DSTG(operand2) for the operand registers during the `beq` comparison of both correct and incorrect user PINs. The comparison occurs after clock cycle 613 (Figure 14), and at cycle 614, the first cycle of the `beq` instruction, both operand registers are loaded with the values to be compared. For the correct user PIN, the values being compared align with the correct PIN size (`4`). However, for the incorrect user PIN, the DSTG(operand1) reveals a mismatch in the comparison, where the values `3` and `4` are compared. This discrepancy results from a single incorrect byte in the user PIN, causing the comparison to fail and preventing the branch from being taken. Therefore, the FIS is identified as the state of `operand1` register (`0x3`) at cycle 614.

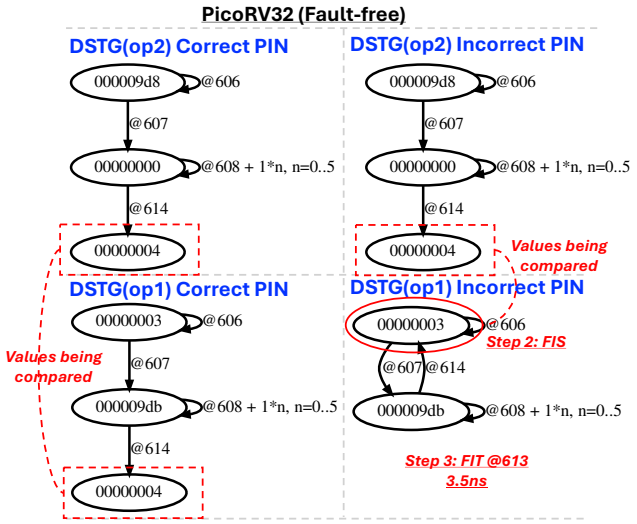

Figure 15: Generated DSTG(operand1) and DSTG(operand2) for correct and incorrect user PIN verifications on PicoRV32.

Step 3: Establish the FIT. The FIT is automatically determined at cycle 613, with a logic propagation time of 3.5ns for all toggled bits in `operand1` register.

Identify Laser Attack Parameter. A laser spot radius of 10 microns (μm) is selected, with the objective of flipping the value in register `operand1` from `0x3` (`0x0011` in binary) to `0x4` (`0x0100` in binary), which requires flipping bits 0 to 2. Using layout data, it is determined that six registers fall within the target radius and are therefore susceptible to the bit-flip. The feasibility of such an injection is supported by prior work demonstrating the practicality of multi-laser fault injection techniques, including multi-spot [7], double-laser [20], and simultaneous laser injections [47], all of which can impact multiple registers within a targeted area.

Case Study	Core Type	Sim Time (s)	DSTG Plot Time (s)	DSTG Depth (cycles)
BuffOver (6.2)	MSP430	8.5	5	173
InstrDup (6.3)	IBEX	7	3	10
VerifyPIN5 (6.4)	PicoRV32	11	9	811

Table 2: The simulation runtime, DSTG plot generation time, and the number of clock cycles covered in each DSTG plot for each case study across different microarchitectures.

Attack Parameter Simulation. The impacted registers are flipped after a delay of 4ns in cycle 613, and it successfully bypasses the first `byteArrayCompare` function in the PIN verification process with incorrect user PIN. However, VerifyPIN5 includes a double-testing mechanism, which consists of two consecutive `byteArrayCompare` functions. Thus, the same attack parameters are applied for the second bit-flip, which occurs after a delay of 4ns at cycle 1256. The second flip targets the same register bits, and the pin verification returns `BOOL_TRUE` despite the input user PIN being incorrect.

6.5 Discussion and Simulation Performance

Discussion. GLITCHGLÜCK was successfully demonstrated on a physical OpenMSP430 ASIC chip with scan-chain support, confirming that simulation-derived fault parameters can be mapped to physical fault observations. Its application to PicoRV32 and IBEX in simulation demonstrates GLITCHGLÜCK's portability across microarchitectures with varying complexity and instruction sets. By visualizing the fault-free hardware state before executing the attack, we gain a better understanding of software-hardware interactions. This approach offers advantages over blind attacks by minimizing the need for exhaustive search and by improving the efficiency of fault injection. Fault injection techniques, such as clock glitching, require precise control over glitch parameters to avoid affecting critical registers. Similarly, laser injection can impact critical registers, depending on the radius of the target region. By representing the DSTG with layout data, we can investigate why certain attack parameters fail – such as when a critical register is faulted by the glitch, leading to an undefined processor state – and why others lead to successful outcomes, allowing for the refinement of attack parameters.

Compared to more complex cores like PicoRV32 and IBEX, fault injection in OpenMSP430 is relatively simpler, as its smaller size results in fewer hardware interactions, which involves fewer critical registers that can be affected. This reduction in complexity allows faults to manifest in a more predictable manner, making attack strategies more intuitive when guided by DSTG analysis. In contrast, attacking PicoRV32 and IBEX presents greater challenges, as determining attack parameters that achieve the desired fault effect without impacting critical registers requires more careful analysis. When the attack parameters consistently involve critical registers, the next step is to identify other FISs that contribute to the VDS decision-making process, as a single VDS can be associated with multiple FISs.

Simulation Performance. Table 2 summarizes the fault-free simulation performance across the three case studies. While the table does not include fault-injected scenarios, their performance time is nearly identical to the fault-free case. The time for documenting the state data of each word-level register is included in the simulation runtime. The IBEX simulation runs slightly faster than the OpenMSP430 simulation because IBEX processes only three instructions. In the buffer overflow example, the time spent on the memory dump operation, which is used to validate the effectiveness of the attack after the fault injection, is excluded from the reported total simulation runtime. The table indicates that the PicoRV32 case study involves the most clock cycles for DSTG analysis, as it includes the entire PIN verification function. However, despite the higher number of clock cycles, the DSTG plotting time remains similar across all case studies, indicating that increasing the number of clock cycles has a small effect on overall DSTG generation time. Additionally, the time spent parsing the VCD file for identifying the FIT is not documented, as only selected clock cycles are analyzed, resulting in a negligible contribution to the overall runtime.

7 Conclusion

We propose GLITCHGLÜCK for identifying critical hardware states that enable software vulnerabilities. GLITCHGLÜCK is enabled by the DSTG, a novel data structure that visually represents software-hardware interactions through a temporal mapping, and that identifies vulnerable states in the interaction. The core innovation of our approach lies in a structured analysis of the hardware state, followed by targeted fault injection based on this analysis. By prioritizing the identification of critical hardware states before fault injection, GLITCHGLÜCK ensures that faults are injected in a more focused manner without adopting fault model assumptions. This approach reduces the need for exhaustive searches required in traditional fault injection methods, which are based on black-box fault models, improving the efficiency of security vulnerability detection and providing more targeted control over the fault injection process. We demonstrate GLITCHGLÜCK on three different processors, with each processor successfully bypassing a different software-level countermeasure. As part of future work, an ASIC taped-out implementation of the IBEX with scan-chain support is being designed to demonstrate the application of GLITCHGLÜCK in a physical environment. In parallel, we are working to automate the methodology to allow automatic identification of attack parameters.

Acknowledgments. This research was supported in part by NSF Award CNS-2219810.

References

[1] T. Ajayi, D. Blaauw, T.-B. Chan, C.-K. Cheng, V. A. Chhabria, D. K. Choo, M. Coltella, S. Dobre, R. Dreslinski, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Li, Z. Liang, U. Mallappa, P. Penzes, G. Pradipta, S. Reda, A. Rovinski, K. Samadi, S. S. Sapatnekar, L. Saul, C. Sechen, V. Srinivas, W. Swartz, D. Sylvester, D. Urquhart, L. Wang, M. Woo, and B. Xu. OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain. In *Proc. Government Microcircuit Applications and Critical Technology Conference*, pages 1105–1110, 2019.

[2] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, July 2020.

[3] Stéphanie Anceau, Pierre Bleuet, Jessy Clédière, Laurent Maingault, Jean-luc Rainard, and Rémi Tucoulou. Nanofocused x-ray beam to reprogram secure circuits. In *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 175–188. Springer, 2017.

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

[5] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented aes: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security*, WESS '10, New York, NY, USA, 2010. Association for Computing Machinery.

[6] Claudio Bozzato, Riccardo Focardi, Francesco Palmarini, et al. Shaping the glitch: optimizing voltage fault injection attacks. *IACR transactions on cryptographic hardware and embedded systems*, 2019(2):199–224, 2019.

[7] Brice Colombier, Paul Grandamme, Julien Vernay, Émilie Chanavat, Lilian Bossuet, Lucie de Laulanié, and Bruno Chassagne. Multi-spot laser fault injection setup: New possibilities for fault injection attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 151–166. Springer, 2021.

[8] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129 vol.2, 2000.

[9] Thomas Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, 2020.

[10] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. Fissc: A fault injection and simulation secure collection. In *International Conference on Computer Safety, Reliability, and Security*, 2016.

[11] Jean-Max Dutertre, Timothé Riom, Olivier Potin, and Jean-Baptiste Rigaud. Experimental analysis of the laser-induced instruction skip fault model. In *Secure IT Systems: 24th Nordic Conference, NordSec 2019, Aalborg, Denmark, November 18–20, 2019, Proceedings 24*, pages 221–237. Springer, 2019.

[12] Mahmoud A. Elmohr, Haohao Liao, and Catherine H. Gebotys. Em fault injection on arm and risc-v. In *2020 21st International Symposium on Quality Electronic Design (ISQED)*, pages 206–212, 2020.

[13] Úlfar Erlingsson. Low-level software security: Attacks and defenses. In *Foundations of Security Analysis and Design IV*, pages 92–134. Springer, Berlin, Heidelberg, 2007.

[14] Pierre-Alain Fouque, Delphine Leresteux, and Frédéric Valette. Using faults for buffer overflow effects. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, page 1638–1639, New York, NY, USA, 2012. Association for Computing Machinery.

[15] Olivier Girard. openmsp430. https://opencores.org/projects/openmsp430, 2017. Accessed Online on 4/15/2024.

[16] T. Given-Wilson, N. Jafri, and A. Legay. Combined software and hardware fault injection vulnerability detection. *Innovations in Systems and Software Engineering*, 16:101–120, June 2020. Received: 28 April 2019; Accepted: 15 May 2020; Published: 27 June 2020.

[17] Florian Hauschild, Kathrin Garb, Lukas Auer, Bodo Selmke, and Johannes Obermaier. Archie: A qemu-based framework for architecture-independent evaluation of faults. In *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 20–30. IEEE, 2021.

[18] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[19] Olivier Hériveaux. Black-box laser fault injection on a secure memory. In *Symposium sur la sécurité des technologies de l'information et des communications-SSTIC 2020*, 2020.

[20] Olivier Hériveaux. Defeating a secure element with multiple laser fault injections. In *Symposium sur la sécurité des technologies de l'information et des communications-SSTIC 2021*, 2021.

[21] Max Hoffmann, Falk Schellenberg, and Christof Paar. Armory: fully automated and exhaustive fault simulation on arm-m binaries. *IEEE Transactions on Information Forensics and Security*, 16:1058–1073, 2020.

[22] Andrea Höller, Armin Krieg, Tobias Rauter, Johannes Iber, and Christian Kreiner. Qemu-based fault injection for a system-level analysis of software countermeasures against fault attacks. In *2015 Euromicro Conference on Digital System Design*, pages 530–533, 2015.

[23] Bryan Kelly, Andrés Lagar-Cavilla, Jeff Andersen, Prabhu Jayana, Piotr Kwidzinski, Rob Strong, John Traver, Louis Ferraro, Ishwar Agarwal, Anjana Parthasarathy, Bharat Pillilli, Vishal Soni, Marius Schilder, Sudhir Mathane, Nathan Nadarajah, and Kor Nielsen. Caliptra: A datacenter system on a chip (soc) root of trust (rot). Technical report, AMD, Google, Microsoft, July 2022.

[24] Martin S Kelly and Keith Mayes. High precision laser fault injection using low-cost components. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 219–228. IEEE, 2020.

[25] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*, pages 388–397. Springer, 1999.

[26] Stefanos Koffas and Praveen Kumar Vadnala. On the effect of clock frequency on voltage and electromagnetic fault injection. In *International Conference on Applied Cryptography and Network Security*, pages 127–145. Springer, 2022.

[27] Raghavan Kumar, Philipp Jovanovic, and Ilia Polian. Precise fault-injections using voltage and temperature manipulation for differential cryptanalysis. In *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, pages 43–48. IEEE, 2014.

[28] J. Laurent, C. Deleuze, F. Pebay-Peyroula, and V. Beroulle. Bridging the gap between rtl and software fault injection. *J. Emerg. Technol. Comput. Syst.*, 17(3), May 2021.

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

[30] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor. *Microprocessors and Microsystems: Embedded Hardware Design* , 71:102862, November 2019. Conference: ASHA Convention. Boston, MA. 2018.

[31] Zhenyuan Liu, Dillibabu Shanmugam, and Patrick Schaumont. Faultdetective: Explainable to a fault, from the design layout to the software. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):610–632, 2024.

[32] LowRISC. Ibex: A small and efficient risc-v core. https://github.com/lowRISC/ibex, 2025. Accessed: 2025-01-20.

[33] LowRISC Community and Contributors. Opentitan: The first open source silicon root of trust. Online: https://opentitan.org/, 2023. Available at https://opentitan.org/.

[34] Alexandre Menu, Jean-Max Dutertre, Olivier Potin, Jean-Baptiste Rigaud, and Jean-Luc Danger. Experimental analysis of the electromagnetic instruction skip fault model. In *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–7, 2020.

[35] Sébastien Michelland, Christophe Deleuze, and Laure Gonnord. From low-level fault modeling (of a pipeline attack) to a proven hardening scheme. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, CC 2024, page 174–185, New York, NY, USA, 2024. Association for Computing Machinery.

[36] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, February 2014.

[37] Debdeep Mukhopadhyay, Shibaji Banerjee, Dipanwita Roy Chowdhury, and Bhargab B. Bhattacharya. Cryptoscan: A secured scan chain architecture. In *14th Asian Test Symposium (ATS 2005), 18-21 December 2005, Calcutta, India*, pages 348–353. IEEE Computer Society, 2005.

[38] Kit Murdock, Martin Thompson, and David Oswald. Faultfinder: lightning-fast, multi-architectural fault injection simulation. In *Proceedings of the 2024 Workshop on Attacks and Solutions in Hardware Security*, pages 78–88, 2024.

[39] Onur Mutlu and Jeremie S. Kim. Rowhammer: A retrospective. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 39(8):1555–1571, August 2020.

[40] Shoei Nashimoto, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji, and Takafumi Aoki. Buffer overflow attack with multiple fault injection and a proven countermeasure. *Journal of Cryptographic Engineering*, 7:35–46, 2017.

[41] Frank Piessens and I. Verbauwhede. Software security: Vulnerabilities and countermeasures for two attacker models. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 990–999. IEEE, March 2016.

[42] Jean-Jacques Quisquater and David Samyde. Eddy current for magnetic analysis with active sensor. In *Proceedings of eSMART*, volume 2002, 2002.

[43] Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. Revisiting fault adversary models – hardware faults in theory and practice. *IEEE Transactions on Computers*, 72(2):572–585, 2023.

[44] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High precision fault injections on the instruction cache of armv7-m architectures. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 62–67. IEEE, 2015.

[45] Cyril Roscian, Jean-Max Dutertre, and Assia Tria. Frontside laser fault injection on cryptosystems - application to the aes' last round -. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 119–124, 2013.

[46] Cyril Roscian, Jean-Max Dutertre, and Assia Tria. Frontside laser fault injection on cryptosystems-application to the aes'last round. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 119–124. IEEE, 2013.

[47] Bodo Selmke, Johann Heyszl, and Georg Sigl. Attack on a dfa protected aes by simultaneous laser fault injections. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 36–46. IEEE, 2016.

[48] Sergei Skorobogatov. Local heating attacks on flash memory devices. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 1–6. IEEE, 2009.

[49] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. Exploration of fault effects on formal risc-v microarchitecture models. In *2022 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 73–83, 2022.

[50] Thomas Trouchkine, Sébanjila Kevin K Bukasa, Mathieu Escouteloup, Ronan Lashermes, and Guillaume Bouffard. Electromagnetic fault injection against a complex CPU, toward new micro-architectural fault models. *Journal of Cryptographic Engineering*, 11(4):353–367, November 2021.

[51] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In *IFIP international workshop on information security theory and practices*, pages 224–233. Springer, 2011.

[52] Jan Van den Herrewegen, David Oswald, Flavio D Garcia, and Qais Temeiza. Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 56–81, 2021.

[53] Jasper van Woudenberg, Rajesh Velegalati, Cees-Bart Breunesse, and Dennis Vermoen Riscure. Improving cpu fault injection simulations: Insights from rtl to instruction-level models. In *2024 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 1–9, 2024.

[54] Raphael A. Camponogara Viera, Philippe Maurine, Jean-Max Dutertre, and Rodrigo Possamai Bastos. Simulation and experimental demonstration of the importance of ir-drops during laser fault injection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(6):1231–1244, 2020.

[55] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[56] Claire Wolf. Picorv32: A size-optimized risc-v cpu. https://github.com/YosysHQ/picorv32, 2025. Accessed: 2025-01-20.

[57] Yuan Yao and Patrick Schaumont. A low-cost function call protection mechanism against instruction skip fault attacks. In *Proceedings of the 2018 workshop on attacks and solutions in hardware security*, pages 55–64, 2018.

[58] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. Software fault resistance is futile: Effective single-glitch attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 47–58, 2016.