μSCAN: Deep Learning Detection of Faulty Micro-architecture States and Patterns from Scan-Chain Data

Dillibabu Shanmugam, Zhenyuan Liu, Andrew Malnicof, and Patrick Schaumont

Worcester Polytechnic Institute, Worcester, MA 01609, USA
{dshanmugam,zliu12,armalnicof,pschaumont}@wpi.edu

Abstract. Hardware Fault injection can leave processors in *weird* micro-architecture states that evade detection by conventional software-level monitors, jeopardizing system reliability. We propose μSCAN, a deep learning framework that leverages scan-chain observability to detect such anomalous states. μSCAN fine-tunes a large language model (LLM) to classify single-cycle processor states as *weird* or *sane*, achieving an average classification accuracy of 92% and maintaining robust performance across different CPU architectures (MSP430, PICO (RV32IC), IBEX (RV32IMC)). We further refine this LLM classifier with reinforcement learning, which sharpens its decision boundaries and improves detection of borderline anomalies. μSCAN also employs a graph neural network (GNN) to analyze multi-cycle fault patterns, capturing complex temporal dependencies that single-cycle analysis might miss. This GNN-based analysis successfully identifies recurring fault sequences and maps them to known Common Weakness Enumeration (CWE) vulnerability classes, revealing potential hardware design flaws. μSCAN demonstrates scalability and generalization on multiple processor architectures (including micro-coded and pipelined cores) and is evaluated with both pre-silicon simulation data and a post-silicon prototype. Our results show that μSCAN enables early detection of micro-architecture vulnerabilities in the design phase and provides a robust post-silicon anomaly detection mechanism.

Keywords: Hardware Fault Analysis· Weird State Machine· Large Language Models

1 Introduction

Silent Data Corruption, a recently observed and quantified phenomenon [2], causes one out of every one thousand central processing units to silently produce incorrect results without triggering any obvious malfunction. Although the precise rate can vary according to architecture and workload, this concern applies just as critically to embedded processors, which often operate in safety-critical settings. A minor fault at the micro-architectural level can jeopardize an entire system, especially under constraints of limited power, tight cost targets, and demanding

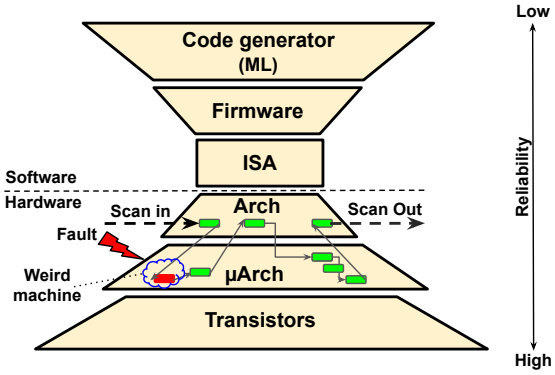


Fig. 1: Overview of the hardware–software abstraction layers—starting at code generation and firmware, passing through the ISA and architectural registers, and reaching down to the micro-architecture and transistors. A micro-architectural fault can create a "weird machine," so scan-chain instrumentation ("scan in/scan out") is used to spot errors otherwise hidden at the micro-architectural level.

environmental conditions. This phenomenon underscores the need for robust fault-detection mechanisms throughout the hardware–software abstraction layers, ensuring that latent errors are discovered early, and that reliability remains intact. In this paper, we propose an ML based mechanism to detect errors that affect the runtime hardware state of a processor. Since the hardware state can cover thousands of bits, the state space is enormous, and the challenge is to reliably distinguish anomalous (faulty) states from the normal states.

Problem Statement: Understanding a fault at the micro-architectural level requires analysis of a massive state space, which remains hidden from software. Under these circumstances, a key question emerges: How can one determine if a micro-architecture state is faulty or not? A second question is how can one determine if a given fault behavior is catastrophic for the firmware's execution or not, in the sense that a fault may lead to extended anomalous behavior. The second question cannot be answered from a single faulty state, but rather requires observation of a sequence of weird and sane states. To the best of our knowledge, no systematic methodology exists to answer either of these questions. The solution proposed in μSCAN is to rely on machine learning techniques that leverage micro-architectural state information obtained through the scan chain of hardware-software system. We aim to show that the classification of single faulty states, as well as the classification of faulty sequences, can be solved adequately using machine learning.

Hardware-Software Abstraction Layers: In modern processor design, the instruction set architecture (ISA) acts as both the interface and a limiting factor as highlighted in the Fig. 1. From the software perspective, the ISA acts as a

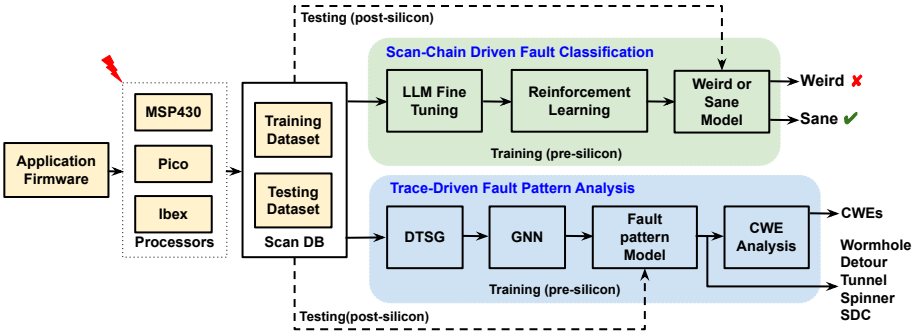


Fig. 2: Overall framework for micro-architectural fault classification and analysis. Micro-architectural data from MSP430, PICO, or IBEX processors is collected via scan chain and divided into training and testing datasets. The upper part uses LLM fine-tuning and reinforcement learning to classify processor states as weird or sane, while the lower part employs graph neural networks for fault pattern analysis, ultimately mapping anomalies to relevant CWE categories.

gateway because firmware and higher-level applications cannot directly access every flip-flop within the processor. Meanwhile, from a hardware perspective, the ISA provides essential protection by preventing users from inadvertently manipulating micro-architectural registers. This separation is beneficial under normal circumstances, but it becomes problematic in fault scenarios. In particular, each layer in the hardware–software abstraction layer tends to mask faults, thereby making it difficult to pinpoint error sources.

Observability Through Scan Chain: Under normal circumstances, firmware developers have no straightforward means of detecting micro-architecture level faults via the usual architectural registers, as the ISA does not expose micro-architectural state. However, the scan-chain, a structure present in many complex chip designs, gives us a handle on this problem. The idea of a scan-chain is to organize every register of the processor, regardless of the abstraction level, into a sequential data structure. By stopping the program and scanning out the contents of the scan-chain, full observability of the processor state becomes possible. Thus, the scan-chain offers the means to detect faulty processor states even before they are able to affect the behavior of the firmware. The faulty state obtained from the scan-chain can be combined with simulation of the processor structure, to obtain a full and detailed trace of a fault effect in the processor hardware. Faulty scan-chain data is a key to the proposed ML based framework that classifies the processor status.

Weird vs Sane Machine: The software community has struggled with precisely defining a faulted processor, in part because the high-level architecture state alone is clearly not enough to capture the fault impact in its entirety. Dullien's [3] proposal defines a *weird machine* as a software state machine that deviates from its intended execution specification. But he does not address how to detect such

anomalies in practice. However, using the scan-chain, the state of a weird machine becomes known as a scan-state where one or more registers in the processor are faulted. FaultDetective [8] demonstrates this idea as a proof-of-concept, and examines the classification of *weird machine* and fault patterns at the hardware level using scan-chain instrumentation. FaultDetective traces short sequences of faulty states and represents them in a data structure called a Dynamic State Transition Graph (DSTG). However, the approach in FaultDetective is application-specific, requiring an exact comparison between a non-faulted ground truth (sane state) and the weird state.

Given the high dimensionality and sequential nature of scan-chain data, we require models capable of capturing long-range dependencies to detect subtle anomalies. Hence, we propose an LLM-based solution that recognises when a given micro-architectural state deviates from normal execution. Furthermore, hardware faults manifest as a *chain* of state changes, rather than a single weird state. For this reason, we also explore a graph-based approach to identify evolving fault patterns over multiple clock cycles. Building on these foundations, this paper introduces two complementary methods. First, *Scan-Chain Driven Fault Classification* leverages scan-chain data and a fine-tuned large language model (LLM) [6] to classify processor states as *weird* or *sane* in real time (Figure 2, top flow). Second, *Trace-Driven Fault Pattern Analysis* applies a graph neural network to DSTGs for detailed fault pattern identification (Figure 2, bottom flow). Together, these methods improve the scalability of fault detection across different abstraction layers.

Technique 1 - Scan-Chain Driven Fault Classification: The primary objective is to determine whether a micro-architectural state is weird or sane. Because real-time observation of every state is impractical, we adopt a two-phase strategy. In the profiling phase (pre-silicon), we simulate both faulty and non-faulty program executions and collect each state using scan-chain. We then fine-tune a large language model (LLM) to capture deviations from the intended semantics. In the deployment phase (post-silicon), we extract a single state from the device via scan chain and classify it as weird or sane based on the trained model. Any state that diverges from the intended behavior is labeled weird, designating a weird machine.

Technique 2 - Trace-Driven Fault Pattern Analysis: This technique employs a graph neural network (GNN) to interpret multi-cycle data and observe a fault pattern. Its primary objective is to analyze the DSTG and to identify recurring fault patterns thereby revealing where a fault originates in the control/data flow of the application. The approach uses a profiling phase (pre-silicon), which simulates both faulty and non-faulty program executions, collects micro-architectural states via scan chain, and constructs the DSTG for GNN training. Because GNNs excel at modeling relational dependencies in graph-based structures, they are well-suited to capturing the complex interactions of fault-induced transitions. In the deployment phase, the trained model detects fault patterns in from observed DSTGs for a large number of

applications. Frequent re-occurrences of a given pattern can signify heightened vulnerability, prompting further netlist analysis to map the identified fault category to a specific Common Weakness Enumeration (CWE). The μSCAN implementation and all datasets used in this paper are available at `https://github.com/Secure-Embedded-Systems/microscan.git`.

Contributions of the paper. We demonstrate the proposed ML-based techniques to classify faults on several processor architectures, including a micro-coded MSP430, a micro-coded PICORV32, and a pipelined IBEX. We construct a dataset consisting of faulty and non-faulty executions for several sample applications on these processors, and capture the micro-architectural register state by gate-level simulation. We then develop a classification framework by fine-tuning a large language model (LLM) to recognize weird or sane patterns within the pre-silicon dataset, validating the approach with both pre-silicon and post-silicon data. We further enhance inference through a reinforcement learning strategy, achieving improved classification accuracy under real-world conditions. At the architectural level, we employ a graph neural network (GNN) for pattern classification. Finally, we attempt to map the discovered vulnerabilities to relevant CWE categories.

Outline of the paper. The rest of this paper is organized as follows: Section 2 reviews background and related work. Section 3 shares the preliminaries and the dataset. Section 4 presents the methodology for micro-architectural fault classification and pattern analysis. Section 5 details the experimental results. Finally, Section 6 concludes the paper.

2 Background and Related Work

A fault injection attack on a processor aims to extract secret data, to escalate privilege, or to induce failures. Although multiple fault injection mechanisms exist, we focus on transient faults created through three injection mechanisms: electromagnetic fault injection (EMFI), laser fault injection (LFI), and clock-glitch injection (CGI). Each fault injection technique causes one or more temporary bit-flips in the processor hardware, and these bit-flips eventually affect the software's behavior. Commonly reported effects include instruction-skip [11] and blocking the execution of a branch [9].

Anomaly detection using ML. A first area of related work is in the area of detecting faults using machine-learning techniques. At the software level, machine-learning techniques have been used to assess multicore soft error reliability [12], to locate faults [10], and to find bugs through large language models [17]. These methods observe run-time states, recognizing patterns that suggest anomalies. At the hardware level, machine-learning techniques have been used for fault characterisation [13], for multisource injection detection [16], and for clock-glitch detection [5]. Our proposed method crosses the hardware/software boundary, by using the low-level machine state observed through a scan chain to infer future anomalous behavior.

Fault modeling. A fault simulation stands or falls with the accuracy of the fault model [1]. The intrinsic characteristics of physical faults must be appropriately translated into a simulation model, reflecting complex phenomena such as multibit flips and timing violations. This is a known hard problem, frequently relying on probabilistic assumptions. In our approach, we avoid fault modeling because we observe fault effects immediately after fault injection by reading out the scan chain of the post-silicon prototype. We then use the faulty bits observed to seed the pre-silicon fault simulation model.

Scan Chain and Dynamic State Transition Graphs. Scan chains allow direct access to the register of a hardware design, enabling an external observer to record the processor state. Using such a recorded (and possibly faulty) state, subsequent processor states can be simulated over time using a pre-silicon model. This simulation provides a precise cycle-by-cycle record of processor states. An Architectural Dynamic State Transition Graph (ADSTG) captures programmer-visible architectural bits by mapping state changes at instruction boundaries [8]. In contrast, the Micro-architectural Dynamic State Transition Graph (MDSTG) logs per-clock-cycle register snapshots—including both ISA-visible and internal micro-architectural bits—to reveal and analyze pipeline-level fault behaviors. By analyzing these state-transition graphs under fault conditions, one can detect patterns that compromise normal operation. A graph neural network (GNN) [14] often supports pattern recognition in these scenarios, highlighting anomalies that arise from injected faults.

Fault Patterns. The following fault patterns, identified through FSMRED firmware experiments, characterize distinct deviations observed in the microarchitecture; graphical representations appear in Appendix Section section 7. The *Sane* pattern denotes fault-free microarchitectural execution, with pipeline and control-flow transitions occurring without disruption. The *Detour* pattern signifies transient deviations, briefly diverging from the intended execution path before recovering, implying temporary signal or timing disturbances. The *Tunnel* pattern involves skipped or delayed state transitions, introducing subtle shortcuts that disrupt expected control-flow progression. The *Wormhole* pattern describes an abrupt, high-impact jump from valid to anomalous states, bypassing sequential progression. Finally, the *Spinner* pattern reflects indefinite looping within a limited set of states due to compromised control-flow management, effectively halting normal processor operation.

3 Preliminaries and Dataset

In this section, we explain the experimental setup, and the programs we have analyzed using μSCAN.

Fault simulation setup Capturing every glitch or laser-induced fault at the microarchitectural level demands a carefully orchestrated simulation flow that

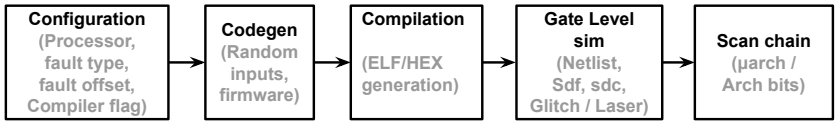


Fig. 3: Block diagram of the integrated fault simulation flow, where the chosen parameters (processor core, fault type, offset, etc.) closely approximate real-world fault conditions. The process moves from firmware generation through gate-level netlist simulation, finally capturing micro-architecture states for anomaly classification.

configures, compiles, and logs hardware states in one cohesive pipeline. The fault simulation setup begins with a configuration stage, where parameters such as the target processor core (PICO, MSP430, or IBEX), fault type (glitch or laser), fault injection offset from trigger or start of a program in clock cycles, and compiler optimizations are chosen. Next, a code generator creates a set of test cases that each combine a randomized input vector into the firmware C code under test. Each of these programs is then compiled into memory images (ELF/HEX). These memory images then become inputs to a gate-level simulation (ModelSim/Verilator) that combines the netlist, delay constraints, and a test script to inject faults by global clock glitching or layout-local laser fault injection. During the gate-level simulation, every clock cycle triggers a scan-chain capture of microarchitectural registers, allowing precise logging of internal state transitions. If a laser fault is configured, the simulator toggles the specified registers at the indicated clock offset, using layout-level information to select the precise register(s) to be faulted. For glitch-based faults, the test script modifies the global clock signal. Each simulation run thus produces a trace of clock-by-clock register states, enabling subsequent analysis—such as anomaly detection via large language models or graph neural networks. We perform all fault simulation experiments on an Intel Xeon Gold 6248 server.

Example: Simulation of Micro-architecture Faults on ASCON Sbox. We apply our fault-injection workflow to CAPRI6, an SoC with six openMSP430 cores in lockstep and scan-chain observability. The ASCON Sbox bitslice routine starts at clock cycle 67 and ends at cycle 155, with each cycle paused to log micro-architecture registers using scan chain. Under normal conditions, the 89 states (cycles 67–155) remain *sane*. However, clock glitches or laser pulses may cause one or more *weird* states. In one campaign, a laser fault occurred in cycle 122 at instruction target `f0b8: cmp r14, r9`. The fault flipped `inst_sa_reg_2_` from 0 to 1 in core3 and this *weird* state persisted until `jnz $-26` restored normal operation in cycle 125. Although no *architectural* registers changed, such faults may leak cryptographic data or cause malfunctions. Listing 1 and Figure 4 refer to this same experiment, showing a partial assembly snippet and potential weird-state transitions using the MD-STG. In another MSP430 design, a corrupted `ADD` instruction became `XOR`, converting $b + (w \times a)$ into $b \oplus (w \times a)$, thereby breaking arithmetic correctness. Integrating these results with pre-silicon simulations

Listing 1 Partial assembly (left) and bitslice C implementation (right) for the ASCON Sbox. The red-highlighted instruction (`f0b8: cmp r14, r9`) was targeted by a laser fault.

```
f086: mov #128, &0x001a
f08c: mov r1,   r14
f08e: add #16,  r14
f092: mov r1,   r15
f094: add #24,  r15
f098: mov r1,   r10
f09a: mov r1,   r11
f09c: add #8,   r11
f09e: mov r15,  r9
f0a0: mov @r15+, r13
f0a2: mov r13,  r12
f0a4: xor @r10+, r12
f0a6: mov @r11+, r8
f0a8: bic r12,  r8
f0aa: mov r8,   r12
f0ac: xor @r14+, r13
f0ae: xor r13,  r12
f0b0: and #511, r12
f0b4: mov r12, &0x0028
f0b8: cmp r14, r9
f0ba: jnz $-26
f0bc: mov #0, &0x001a
```

```
Input: x[0..4],Output: y[0..4]
for(j = 0; j < WORD_SIZE; ++j){
q0[j]=!(x3[j]^x4[j]);
q1[j]=!x4[j];
t0[j]=q0[j]&q1[j];
q2[j]=x0[j]^x2[j]^x4[j];
q3[j]=x1[j];
t1[j]=q2[j]&q3[j];
q4[j]=x0[j]^x1[j]^x4[j];
q5[j]=x1[j];
t2[j]=q4[j]&q5[j];
q6[j]=x3[j]^x4[j];
q7[j]=x0[j];
t3[j]=q6[j]&q7[j];
q8[j]=x3[j]^t1[j]^t2[j];
q9[j]=x1[j]^x2[j];
t4[j]=q8[j]&q9[j];
y0[j]=x0[j]^x1[j]^x2[j]^x3[j]^t1[j];
y1[j]=x0[j]^x2[j]^x3[j]^x4[j]^t4[j];
y2[j]=x1[j]^x2[j]^x3[j]^t0[j];
y3[j]=x0[j]^x1[j]^x2[j]^x3[j]^x4[j]^t3[j];
y4[j]=x3[j]^x4[j]^t2[j]; }
```

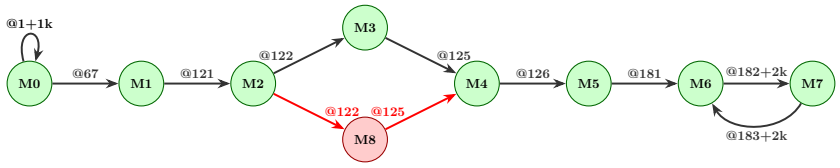


Fig. 4: MD-STG of the ASCON Sbox showing potential weird-state transitions (red color) under fault injection.

yields a comprehensive set of *sane* and *weird* states for subsequent training in our framework.

Dataset. Building upon the weird and sane states observed in these campaigns, we now systematically compile them into a dataset. We generated random-instruction tests for *PICO* (RV32IC) and *IBEX* (RV32IMC), each covering more than 40 types of ISA instruction (load / store, branching, arithmetic, logic) to exercise all CPU registers. In our post-silicon evaluation on CAPRI6, we specifically targeted the `MOV`, `ADD`, and `MUL` instructions under two fault-injection campaigns (laser-based and clock-glitch-based), collecting 517 *weird* states in total (265 from laser, 252 from clock glitches). Table 1 summarizes the *weird* and *sane* states from these experiments alongside the ASCON Sbox runs.

Processors	Arch. Reg	μarch. Reg	Firmware	Sane	Weird
MSP430	385	275	FSMRed	28	627
			ASCON Sbox [18]	89	3120
			PinVerify5 [4]	248	22576
PICO	1024	1265	Random Instr.	80	480
			ASCON Sbox	24	350
IBEX	993	1013	Random Instr.	486	1013
			ASCON Sbox	402	1000
Post-Silicon (CAPRI6)			MOV/ADD/MUL [15]	517	517

Table 1: Each processor exhibits distinct architecture and microarchitecture register counts (Columns 2 and 3). Microarchitectural registers are typically not directly accessible by programmers. Column 4 indicates the firmware scenario, while Columns 5 and 6 show the sane and weird state counts, respectively. The last row lists instructions (MOV/ADD/MUL) used on CAPRI6.

4 Methodology: Fault Classification and Pattern Analysis

Our approach consists of two complementary models: (*i*) an LLM-based classifier to flag individual weird states, and (*ii*) a GNN-based analyzer to identify patterns of faults across multiple states. We describe each in turn.

4.1 LLM-based classifier

We arrive at a robust mathematical model that classifies microarchitectural states as weird or sane.

Fine tuning LLM Models We treat the classification task as a pattern-recognition problem, leveraging a large language model capable of detecting subtle semantic relationships within high-dimensional scan-chain data. Such models excel at capturing long-range dependencies, making them especially effective for distributed fault scenarios in which anomalies span thousands of bits. We employ *DeepSeek-R1-Distill-Qwen-1.5B* as our base model due to its open-source.

Profiling and Deployment Phase. We represent the microarchitectural register dump as a sequence of tokens (bits or groups of bits) that the LLM can ingest via its tokenizer. To reduce computational overhead while maintaining accuracy, we adopt Quantized Low-Rank Adaptation (QLoRA). This method quantizes the base model's weights to 4-bit precision and inserts low-rank adapters that learn task-specific features. During fine-tuning, we freeze the main LLM parameters and update only the LoRA components [7], allowing the model to capture essential directions of variance via a low-rank decomposition. After training, the low-rank matrices merge back into the base model, resulting in a final classifier that infers *weird* or *sane* states on test data. Figure 5 shows the block diagram (Right) and the network hyperparameters (Left) used for the model.

Hyperparameter	Value
EPOCHS	4/16
Learning Rate	$1 \times 10^{-5}/2 \times 10^{-5}$
LoRA Rank	4/8/16
LoRA Alpha	64/128
LoRA Dropout	0.1/0.2
Batch Size	4/8/16
Warmup Steps	50/200
Threshold	0.48/0.5/0.52

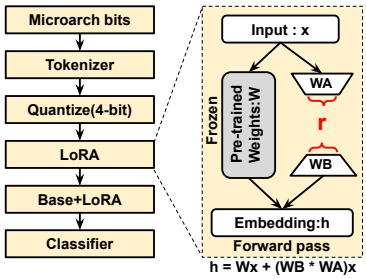


Fig. 5: (Left) Key hyperparameters used in LoRA fine-tuning. (Right) LoRA + Base Model architecture, illustrating how LoRA adapters integrate with the frozen pretrained weights.

Mathematical Model for Anomaly Detection Let $\mathbf{x} \in \mathbb{R}^d$ represent the microarchitectural state captured via scan-chain, where d denotes the dimensionality (the number of micro-architecture bits). We employ a large language model (LLM) with QLoRA to classify $\mathbf{x}$ as either *weird* (faulty) or *sane* (non-faulty). Formally, we define

$$y = \mathrm{LLM}_{\theta+\Delta}(\mathbf{x}), \tag{1}$$

where θ are the frozen weights of the base pre-trained model, and Δ represents the low-rank updates learned during fine-tuning. The output y is a logit or score that reflects the likelihood of $\mathbf{x}$ being a *weird* state.

For a binary classification task, we apply a sigmoid function $\sigma(\cdot)$ to convert y into a probability:

$$p(\text{weird} \mid \mathbf{x}) = \sigma\big(y(\mathbf{x})\big). \tag{2}$$

A decision boundary $\tau \in [0, 1]$ determines the final label:

$$\mathrm{Class}(\mathbf{x}) = \begin{cases} \text{weird}, & \text{if } p(\text{weird} \mid \mathbf{x}) \geq \tau, \\ \text{sane}, & \text{otherwise.} \end{cases} \tag{3}$$

During fine-tuning, we minimise a supervised loss (e.g., cross-entropy) over labelled states $\{(\mathbf{x}_i, y_i)\}$, where $y_i \in \{\text{weird}, \text{sane}\}$. Notably, only the LoRA parameters Δ receive gradient updates, while the base weights θ remain fixed. This strategy ensures that the essential directions of variation in the data are captured in the low-rank subspace, thus allowing efficient adaptation of the large model to our anomaly detection task.

Reinforcement Learning on Top of LLM Fine-Tuning Although supervised fine-tuning equips an LLM with broad discriminative features for *weird* vs. *sane* states, borderline or outlier cases may remain problematic. Reinforcement learning refines decision boundaries by rewarding correct classifications and penalising errors, prompting the model to adapt to ambiguous samples. Over repeated updates, it strengthens effective classification strategies and better detects subtle anomalies. This iterative process is especially valuable for microarchitectural data, where faults can span thousands of bits.

Mathematical Model. Let $\mathbf{x} \in \mathbb{R}^d$ denote the extracted microarchitectural state, and let $a \in \{\text{weird}, \text{sane}\}$ be the classification action. The LLM parameters, initially tuned via supervised learning, define a policy $\pi_\theta(a \mid \mathbf{x})$ that assigns a probability to each action. After an action a is chosen, a reward function $r(\mathbf{x}, a)$ is provided: for example,

$$r(\mathbf{x}, \text{weird}) = p(\text{weird} \mid \mathbf{x}) \quad \text{and} \quad r(\mathbf{x}, \text{sane}) = p(\text{sane} \mid \mathbf{x}),$$

where $p(\cdot \mid \mathbf{x})$ corresponds to the model's confidence for each label. The RL objective is to maximize the expected return,

$$\max_\theta \ \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\big[\mathbb{E}_{a\sim\pi_\theta}\big(r(\mathbf{x}, a)\big)\big],$$

where $\mathcal{D}$ is the distribution of microarchitectural states. Intuitively, correct classifications (matching ground truth) yield higher rewards, nudging the policy to strengthen those decisions. As the model repeatedly observes states where it previously misclassified, the updated reward feedback guides it to adjust its predictions. Thus, the policy gradually converges to an improved discriminator for weird versus sane states, leveraging both the representation learned by fine-tuning and the adaptive reward-driven updates of RL.

4.2 GNN-based analyzer

Next, we leverage graph neural networks to classify fault patterns over time.

Motivation and DSTG Representation. While single-state classification can identify anomalies at a single clock cycle, many hardware faults emerge across multiple cycles and transitions. To capture these multi-cycle behaviours, we adopted a *Dynamic State Transition Graph* (DSTG), in which each node represents a microarchitectural state (the set of registers at a given cycle), and each directed edge denotes a transition triggered by instruction execution or a fault injection. For instance, Figure 4 shows an assembly snippet from the `ASCON SBox` where a fault changes the instruction register, causing a weird state that lasts for three clock cycles before returning to normal. Because this anomaly spans multiple cycles, a graph-based model is more effective than per-cycle analysis at capturing the fault's evolution over time.

Profiling and Deployment Phase. In the profiling phase, we simulate both normal and faulty executions for each target application (e.g. `ASCON SBox`, `FSMRED`, `VP5`). These simulations yield DSTGs that record normal transitions as well as fault-induced deviations. We label each DSTG according to the observed pattern among six known fault classes (e.g. wormhole, tunnel, spinner, detour, silent data corruption, or crash). The GNN is then trained on these labelled graphs, learning to recognise topological and feature-based indicators of each fault category. In practice, we store node attributes (such as instruction type, clock cycle index, or register usage) and edge attributes (timing differences, potential glitch sites,

etc.) in a graph database, ensuring the GNN has sufficient context to learn discriminative patterns. Once trained, the GNN is deployed on device traces collected *post-silicon* via the scan chain. As new microarchitectural states are read, the corresponding DSTG is constructed in batches and fed to the GNN. The model then predicts which of the six patterns best describes the transitions within that graph.

Mathematical Model for Fault Pattern Classification. Formally, let $G = (V, E)$ be a DSTG where each node $i \in V$ has features $\mathbf{x}_i$ (e.g. instruction label, cycle index) and each edge $(i, j) \in E$ has features $\mathbf{e}_{ij}$ (e.g. time deltas, potential glitch). A GNN iteratively updates node embeddings $\mathbf{h}_i^{(l)}$:

$$\mathbf{h}_i^{(l)} = \sigma\Big(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} \, \mathbf{W}^{(l)} \, \mathbf{h}_j^{(l-1)} \Big),$$

where $\alpha_{ij}^{(l)}$ is an attention coefficient (potentially derived from $\mathbf{e}_{ij}$) and $\mathbf{W}^{(l)}$ are learnable parameters. After L layers, the graph-level embedding emerges from pooling the final node embeddings:

$$\mathbf{z} = \mathrm{Pool}(\{\mathbf{h}_i^{(L)} : i \in V\}),$$

which is then fed into a classifier $\mathrm{MLP}(\mathbf{z})$ that outputs a fault-pattern label among the six categories. This GNN-based method scales well to large DSTGs and offers efficient inference, as the local connectivity constrains the amount of message passing per node. Consequently, it provides an effective framework for multi-cycle fault analysis, bridging the gap between low-level transitions and high-level categorisation of fault behaviour.

5 Experimental Evaluation and Analysis

In this section, we present a comprehensive evaluation of our fault detection framework, which integrates an LLM-based classifier enhanced through reinforcement learning (RL) and a graph neural network (GNN) for fault pattern analysis. The experiments are designed to validate our hypothesis that RL-enhanced classification can significantly improve the detection of anomalous states, while the GNN-based approach effectively identifies multi-cycle fault patterns. Evaluation metrics include classification accuracy, convergence speed, and fault pattern recognition precision.

5.1 Classification Performance with LLM

Table 2 shows a progressive improvement in classification accuracy and offers insight into the trade-offs between model performance and training cost. In Case 1, the classifier obtains a modest accuracy (77%), which reflects a relatively small training set and limited coverage of fault scenarios. Moving to Case 2, a

Test Case	Target	FSMRed	ASCON Sbox	VP5	Random Inst	MOV/ADD/MUL	Dataset Sane	Dataset Weird	Train Sane	Train Weird	Test Sane	Test Weird	Acc. (%)	Time (min) (Train+Test)
1	MSP430	•	•	•			365	365	183	183	182	182	79	6 + 1
2	MSP430	•	•	•			365	26,273	292	21,019	73	5,254	94	460 + 8
3	MSP430	•	•	•			365	26,273	365	26,273	564	39,085	98	620 + 25
4	PICO		•		•		104	830	80	480	24	350	94	60 + 5
5	IBEX		•		•		1499	1,402	486	1,013	402	1,000	97	370 + 15
6	CAPRI6					•	365	26,273	292	21,019	517	517	90	460 + 6

Table 2: Classification performance of the LLM-based fault-state classifier evaluated across six distinct experimental cases. Case 1 (Balanced): Dataset with equal numbers of sane and weird states (365 each) from three MSP430 firmware, evenly split for training and testing. Case 2 (Imbalanced): Full MSP430 dataset (365 sane, 26,273 weird) divided into 80% training and 20% testing subsets. Case 3 (Augmented): Entire MSP430 dataset used for training (100%) and expanded testing dataset by 50% (approximately 150% total states). Case 4 (Cross-firmware, PICO): Classifier trained on random-instruction firmware, evaluated on ASCON firmware states (PICO platform). Case 5 (Cross-firmware, IBEX): Same approach as Case 4 but on IBEX platform, using a smaller GPU (NVIDIA T4, 16GB RAM) compared to other experiments (NVIDIA A100, 40GB RAM). Case 6 (Cross-platform, CAPRI6): Classifier trained on MSP430 firmware states, tested on MOV/ADD/MUL instruction states of CAPRI6 processor. Results highlight progressive improvements in accuracy with increasing data size and demonstrate computational trade-offs in scaling and cross-platform evaluation.

more extensive dataset yields higher accuracy (94%), but the training phase grows proportionally longer. In Case 3, augmenting the data further elevates the accuracy (98%) yet incurs an even greater training time, underscoring the overhead of large-scale learning. PICO and IBEX confirm that the approach generalizes across distinct processor architectures, PICO and IBEX, while preserving strong precision (94% and 97%, respectively). Finally, `CAPRI6` achieves a precision 90%, demonstrating the viability of the method under realistic constraints, but also highlighting the computational demands of increased data diversity.

Comparison. Figure 6 presents a head-to-head comparison of six detectors on the 365-state test subset from Case 1. The QLoRA fine-tuned classifier—using 4-bit weight quantization and rank-8 adapters—requires 99 % fewer trainable parameters and 80 % less GPU memory than conventional full-model tuning, yet achieves the highest accuracy (79.00 %), substantially outperforming both transformer baselines (BERT-base: 51.00 %; RoBERTa-base: 57.00 %) and three non-LLM approaches (Isolation Forest: 49.45 %; Autoencoder: 50.55 %; 1D-CNN: 55.22 %). This significant performance margin indicates that low-rank adapters, when applied to domain-specific "bit tokens," capture subtle microarchitecture-level fault signatures that generic language models and unsupervised detectors fail to discern.

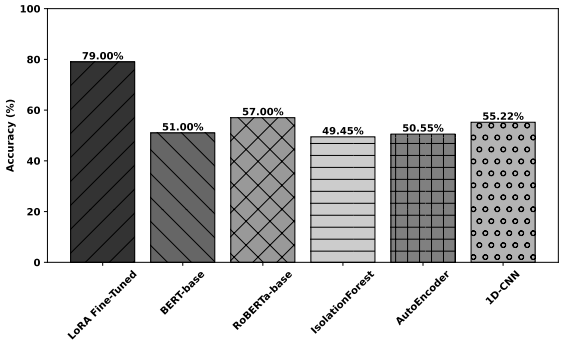


Fig. 6: Comparison of model accuracies.

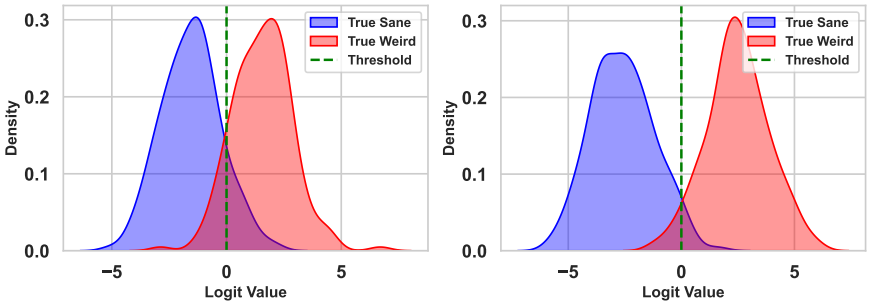


Fig. 7: This figure shows the distribution of predicted logits for two true labels (sane vs. weird) for the SFT model (left) and for the RL_SFT model (right). The vertical dashed line at logit = 0 corresponds to a probability of 0.5, illustrating how borderline anomalies arise where the two distributions overlap.

LLM Classification and RL Enhancements In Test Case 1, the SFT LLM shows limited success in distinguishing weird from sane states, resulting in lower accuracy. We incorporate the MSP430 case 1 LoRA adapter into the model and then apply RL to 30% of the data, following Section 4.2. The resulting RL model is evaluated on the same test samples to measure improvements over the SFT baseline. Figure 7 compares the logit distributions for the SFT model (left) and the RL+SFT model (right). The left distribution, with 79.12% accuracy, shows moderate overlap between sane and weird classes, revealing borderline cases that the SFT model struggles to classify. The right distribution, produced after RL on top of SFT, presents fewer borderline cases and an accuracy of 90.22%, indicating that RL refines the model's ability to distinguish weird from sane states. This improvement suggests that combining RL with SFT yields more effective separation of borderline samples and enhances overall classification accuracy.

Injection	Firmware	Campaigns		Clock	GNN
		Train	Test	Cycles	Acc.
Glitch	FSMRED	140	28	28	67.86%
	ASCON	480	96	89	96.88%
Laser	ASCON	2171	435	89	98.62%
	VP5	2171	435	293	99.70%
-	All	4962	994	293	99.80%

Table 3: GNN classification accuracy for multi-cycle fault pattern detection on MSP430. Training utilized cores {0,1,2,4,5}; testing on core 3 to verify cross-core generalization.

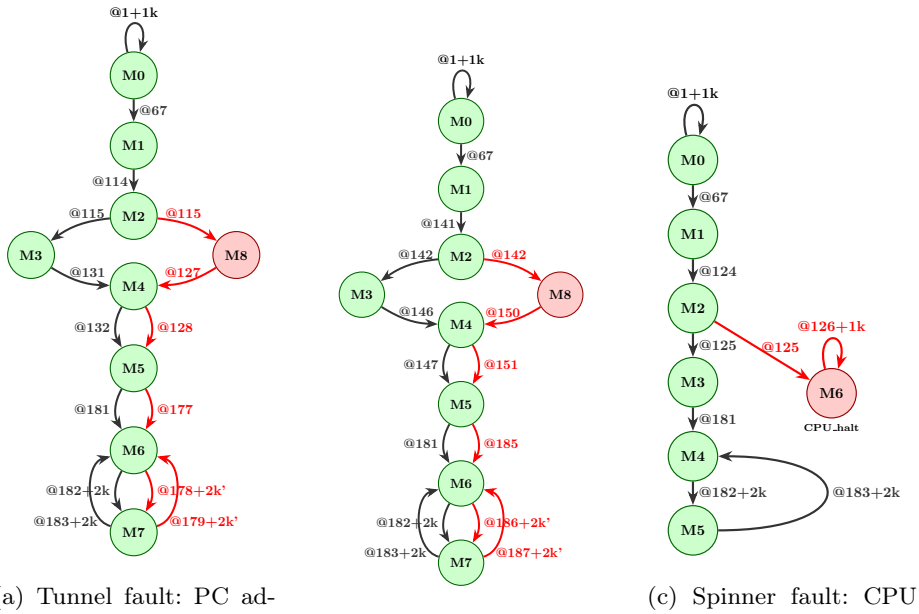
	sane	wormhole_imp	wormhole_noimp	spinner	tunnel	detour
sane	117	0	0	0	0	0
wormhole_imp	0	98	0	0	0	0
wormhole_noimp	0	0	420	0	0	0
spinner	0	0	0	17	0	0
tunnel	0	0	0	2	47	0
detour	0	0	0	0	0	293

Table 4: Confusion matrix (combined scenario). Rows represent true fault patterns, columns show predictions.

5.2 GNN-Based Multi-Cycle Pattern Detection

Dataset and Methodology. We generated multi-cycle fault datasets on MSP430 using three firmware scenarios (`FSMRED`, `ASCON`, and `VP5`) under clock-glitch and laser injections. Each *campaign* yielded an MDSTG capturing micro-architectural state transitions across full execution: `FSMRED` (28 cycles), `ASCON` (89 cycles), and `VP5` (293 cycles). Training utilized states from cores {0,1,2,4,5}, reserving core 3 exclusively for testing. The GNN classifier processed MDSTGs, predicting fault patterns (e.g., *wormhole*, *spinner*, *detour*). We employed a multi-layer attention-based architecture with ReLU activation and assessed classification accuracy.

Results and Discussion. Table 3 summarizes experimental results across injection types and firmware. Glitch-based scenarios achieved around 80% accuracy, with `ASCON` significantly outperforming `FSMRED`, attributed to more distinguishable fault patterns. Laser injections yielded accuracies above 98%, demonstrating GNN efficacy under varied fault mechanisms. The combined scenario confirmed robustness with 99% accuracy, highlighting strong generalization. The confusion matrix (Table 4) shows precise classification across fault types, confirming the GNN's effectiveness in capturing distinctive multi-cycle fault patterns. To identify recurring fault motifs, we first assign each MDSTG the class with the highest softmax score and extract its penultimate-layer embedding. We then cluster these embeddings via agglomerative clustering with cosine affinity and flag any cluster exceeding a user-defined size threshold as a recurring motif. Finally, each identified motif is manually mapped to its corresponding CWE category. This methodology is scalable to large SoCs and enables validation of complex applications via periodic scan-out.



(a) Tunnel fault: PC advance

(b) Tunnel fault: PC lag

(c) Spinner fault: CPU halt

Fig. 8: Micro-architectural state-transition graphs illustrating sane (green nodes, black edges) versus weird/fault-induced (red nodes, red edges) execution paths.

5.3 Correlating Fault Patterns with CWEs

Fault patterns observed in experiments closely align with CWE classifications, highlighting hardware control-flow vulnerabilities.

Tunnel Fault: Program Counter (PC) Advance. Figure 8a depicts a tunnel fault induced by laser injection at instruction `f0ac` (`xor @r14+, r13`). This injection prematurely advances the program counter (PC), bypassing three critical instructions (`f0ac`, `f0ae`, `f0b0`). Consequently, the ASCON SBox computation omits an entire iteration, failing to update register data correctly. While immediate erroneous states are not evident, the final output incorrectly resolves to `0x0018` instead of `0x00E5`. This behavior precisely aligns with **CWE-1332** (Skipped Valid Transitions), wherein valid, necessary transitions are unintentionally skipped due to faults, causing incorrect system states.

Tunnel Fault: Program Counter (PC) Lag. Figure 8b illustrates another tunnel fault scenario introduced by laser injection at instruction `f0b4` (`mov r12, &0x0028`). This fault introduces a delay of four clock cycles without skipping instructions. After these additional cycles, the processor's micro-architectural state realigns, preserving the correctness of final output data. Nonetheless, the internal timing disturbance remains undetected by the CPU, indicative of insufficient

hardware-level control flow management. This critical fault effect maps directly to **CWE-691** (Insufficient Control Flow Management), where transient timing disturbances in the execution path go unnoticed due to inadequate checking mechanisms.

Repeated Tunnel Fault Patterns and Vulnerability Analysis. Repeated occurrences of tunnel faults were observed consistently within experiments on core 3, highlighting underlying vulnerabilities in the program counter logic. Netlist analysis revealed that PC registers (e.g., `pc_reg_2`,`pc_reg_3`, `pc_reg_4`) sample memory address bus signals directly without protective synchronization. Mid-cycle glitches in these registers thus lead to unintended forward or backward jumps in instruction flow. The absence of dedicated synchronization or protective logic facilitates recurrent tunnel faults, further reinforcing the identified vulnerabilities associated with CWE mappings **CWE-1332** and **CWE-691**.

Spinner Fault in ASCON Computation. Figure 8c illustrates a spinner fault observed during laser-based fault injection experiments on the ASCON bitslice implementation. A targeted laser pulse at coordinates (2788, 2176) at clock cycle 125 triggered this fault. Detailed layout-level analysis confirmed that all relevant pipeline registers maintained their correct state prior to fault injection. Immediately after the laser pulse, the signal `dbg_0.cpu_ctl_reg_4_.Q` transitioned from 0 to 1. This transition activated the debug halt/reset logic within the processor's control circuitry, propagating through a dedicated debug pathway to assert a global CPU halt signal (evidenced by the subsequent assertion of `clock_module_0.dbg_rst_noscan_reg.Q`). The global halt condition forced the frontend pipeline logic, particularly the program counter registers (`frontend_0.pc_reg_[15..0].Q`), to reset to zero. Consequently, the processor immediately ceased executing further instructions, becoming trapped in an unresponsive halted state.

The spinner fault emerged repeatedly, observed in 17 out of 994 campaigns conducted under similar injection parameters. This recurrent behavior directly corresponds to **CWE-835** (Loop with Unreachable Exit Condition). Specifically, the debug halt path creates an internal state from which no recovery is possible without external intervention or reset. This fault demonstrates a critical vulnerability in the processor's debug and control-flow logic, underscoring the necessity for enhanced protection mechanisms against targeted physical fault injections, such as precise laser strikes, capable of exploiting sensitive internal debug signals.

Limitations. The proposed methodology relies on scan-chain data collected primarily during pre-silicon stages, which may not be feasible or permitted post-silicon due to restricted access in authorized environments. In cases where scan-chain or memory-based access to micro-architectural states is unavailable, partial visibility can still be obtained through alternative methods such as JTAG interfaces, performance counters, or firmware-based debug instructions. Furthermore, dimensionality-reduction techniques (e.g., PCA or t-SNE) were intentionally not explored in this study to preserve subtle yet critical fault

signatures that could otherwise be diluted. Future research may investigate suitable approaches to dimensionality reduction that maintain diagnostic precision while enhancing computational efficiency. The scan-chain implementation requires four dedicated I/O pins and imposes a 2–4 % overhead in scan flip-flop area and critical-path delay. A full scan capture operation completes in under one second.

6 Conclusion

A Deep-Learning-based framework for micro-architecture fault detection uses scan-chain data to expose hidden anomalies in embedded processors. Our experiments achieved 92% detection accuracy across multiple architectures and fault injection methods, indicating that direct microarchitecture observability can substantially mitigate reliability and security risks. By systematically correlating recurring hardware fault patterns with established CWE categories, the proposed approach facilitates a deeper understanding of potential vulnerabilities and enables standardized mitigation strategies. These findings underscore the value of early, cross-layer detection in safety-critical and cryptographic contexts. Future work includes targeted fault analysis of memory modules under realistic application workloads, the development of hybrid techniques that integrate hardware trace data with software profiling, and the evaluation of on-chip inference accelerators for real-time fault detection. Utimately, this methodology can help formalize hardware-level fault detection.

Acknowledgements. This research was supported in part by NSF Award 2219810.

References

[1] Adhikary, A., Petrucci, G.T., Tanguy, P., Lapôtre, V., Buhan, I.: SoK: The apprentice guide to automated fault injection simulation for security evaluation. Cryptology ePrint Archive, Paper 2024/1944 (2024), https://eprint.iacr.org/2024/1944

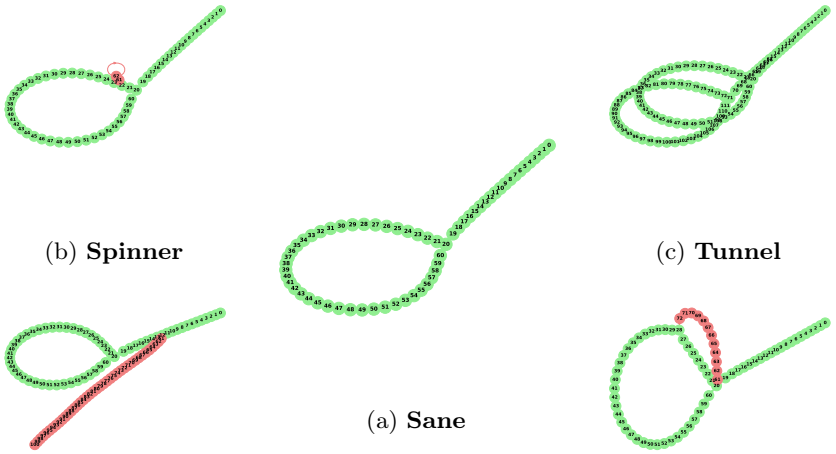
[2] Dixit, H.D., Pendharkar, S., Beadon, M., Mason, C., Chakravarthy, T., Muthiah, B., Sankar, S.: Silent data corruptions at scale (2021), https://arxiv.org/abs/2102.11245

[3] Dullien, T.: Weird machines, exploitability, and provable unexploitability. IEEE Transactions on Emerging Topics in Computing **8**(2), 391–403 (2020). https://doi.org/10.1109/TETC.2017.2785299

[4] Dureuil, L., Petiot, G., Potet, M., Le, T., Crohen, A., de Choudens, P.: FISSC: A fault injection and simulation secure collection. In: Skavhaug, A., Guiochet, J., Bitsch, F. (eds.) Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9922, pp. 3–11. Springer (2016). https://doi.org/10.1007/978-3-319-45477-1_1, https://doi.org/10.1007/978-3-319-45477-1_1

[5] Gambra, A., Chatterjee, D., Rioja, U., Armendariz, I., Batina, L.: Machine learning-based detection of glitch attacks in clock signal data. IACR Cryptol. ePrint Arch. p. 1939 (2024), https://eprint.iacr.org/2024/1939
[6] Howard, J., Ruder, S.: Fine-tuned language models for text classification. CoRR **abs/1801.06146** (2018), http://arxiv.org/abs/1801.06146
[7] Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: Lora: Low-rank adaptation of large language models (2021), https://arxiv.org/abs/2106.09685
[8] Liu, Z., Shanmugam, D., Schaumont, P.: Faultdetective: Explainable to a fault, from the design layout to the software. IACR Transactions on Cryptographic Hardware and Embedded Systems **2024**, 610–632 (2024). https://doi.org/10.46586/tches.v2024.i4.610-632, https://tches.iacr.org/index.php/TCHES/article/view/11804
[9] Moore, S., Anderson, R., Cunningham, P., Mullins, R., Taylor, G.: Improving smart card security using self-timed circuits. In: Proceedings Eighth International Symposium on Asynchronous Circuits and Systems. pp. 211–218 (2002). https://doi.org/10.1109/ASYNC.2002.1000311
[10] Rafi, M.N., Kim, D.J., Chen, T.H., Wang, S.: Enhancing fault localization through ordered code analysis with llm agents and self-reflection (2024), https://arxiv.org/abs/2409.13642
[11] Rivière, L., Najm, Z., Rauzy, P., Danger, J.L., Bringer, J., Sauvage, L.: High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures. In: IEEE International Symposium on Hardware-Oriented Security and Trust (2015). https://doi.org/10.1109/HST.2015.7140238
[12] da Rosa, F.R., Garibotti, R., Ost, L., Reis, R.: Using machine learning techniques to evaluate multicore soft error reliability. IEEE Transactions on Circuits and Systems I: Regular Papers **66**(6), 2151–2164 (2019). https://doi.org/10.1109/TCSI.2019.2906155
[13] Saha, S., Jap, D., Patranabis, S., Mukhopadhyay, D., Bhasin, S., Dasgupta, P.: Automatic characterization of exploitable faults: A machine learning approach. IEEE Transactions on Information Forensics and Security **14**(4), 954–968 (2019). https://doi.org/10.1109/TIFS.2018.2868245
[14] Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G.: The graph neural network model. IEEE Transactions on Neural Networks **20**(1), 61–80 (2009). https://doi.org/10.1109/TNN.2008.2005605
[15] Shanmugam, D., Liu, Z., Schaumont, P.: Capri6: A solution for fault root cause detection. Proceedings of the 34th Microelectronics Design and Test Symposium (IEEE MDTS 2025), Accepted for publication (2025), https://mdts.ieee.org/full-agenda-2025/
[16] Shrivastwa, R.R., Guilley, S., Danger, J.L.: Multi-source fault injection detection using machine learning and sensor fusion. In: Stănică, P., Mesnager, S., Debnath, S.K. (eds.) Security and Privacy. pp. 93–107. Springer International Publishing, Cham (2021)
[17] Stein, A., Wayne, A., Naik, A., Naik, M., Wong, E.: Where's the bug? attention probing for scalable fault localization (2025), https://arxiv.org/abs/2502.13966
[18] Stoffelen, K.: Optimizing s-box implementations for several criteria using sat solvers. In: Peyrin, T. (ed.) Fast Software Encryption. pp. 140–160. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)

7 Appendix



(b) **Spinner** (c) **Tunnel**

(a) **Sane**

(d) **Wormhole** (e) **Detour**

Fig. 9: Five principal fault patterns derived from FSMRED firmware experiments. Each subfigure illustrates a distinct micro-architectural path: (a) Sane: The nominal, fault-free execution in which all microarchitectural state transitions occur strictly according to the design specification. (b) Spinner: A fault pattern in which execution becomes trapped within a confined subset of microarchitectural states, endlessly cycling without advancing the control-flow. (c) Tunnel: A fault-induced anomaly that bypasses one or more intended intermediate states, creating a direct shortcut through the state sequence. (d) Wormhole: A high-impact fault event characterized by an instantaneous jump to a distant, non-sequential microarchitectural state, circumventing all intermediate transitions. (e) Detour: A transient fault pattern whereby execution momentarily diverges along an alternate sequence of microarchitectural states before reconverging with the intended progression.