

Staggered Instruction-Fetch Duplication: A Lightweight Processor-Level Countermeasure Against Fault Injection*

Shoei Nashimoto

Nashimoto.Shoei@bx.MitsubishiElectric.co.jp
Mitsubishi Electric Corporation
Kamakura, Kanagawa, Japan

Naoya Okada[†]

Okada.Naoya@cb.MitsubishiElectric.co.jp
Mitsubishi Electric Corporation
Kamakura, Kanagawa, Japan

Yuichiro Murachi[†]

Murachi.Yuichiro@dn.MitsubishiElectric.co.jp
Mitsubishi Electric Corporation
Kamakura, Kanagawa, Japan

Abstract

Fault attacks exploit transient hardware faults to bypass software-level security checks in embedded processors. Processor-level redundancy can detect such faults, but duplicating the whole processor is costly for resource-constrained devices. This paper proposes Staggered Instruction-Fetch Duplication (SIFD), a lightweight processor-level countermeasure based on instruction-fetch redundancy. SIFD fetches each instruction twice in consecutive cycles, while fetching two different instructions in every fetch cycle. This staggered schedule detects inconsistent fetch results without full processor duplication, compiler support, or cryptographic hardware. We implement SIFD on a custom RV32I processor and evaluate it on an Artix-7 FPGA using clock fault injection (CFI) and electromagnetic fault injection (EMFI). SIFD increases slice utilization by 15.0% and execution time by 16.3%. It reduces the attack success rate from 0.86% to below 0.01% under CFI and from 0.32% to 0.13% under EMFI. It also detects 73.18% of CFI attempts and 11.44% of EMFI attempts. These results indicate that SIFD can convert many fault effects into explicit detections and reduce successful fault-injection outcomes, especially under CFI.

CCS Concepts

• Security and privacy → Hardware attacks and countermeasures.

Keywords

Fault injection attack, Countermeasure, RISC-V, Processor

ACM Reference Format:

Shoei Nashimoto. 2027. Staggered Instruction-Fetch Duplication: A Lightweight Processor-Level Countermeasure Against Fault Injection. In *32nd Asia and South Pacific Design Automation Conference (ASPDAC '27)*, January 25–28, 2027, Tokyo, Japan. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3840404.3848186>

*This is the submitted version of a paper accepted at ASP-DAC 2027. The published version is available at <https://doi.org/10.1145/3840404.3848186>.

[†]These authors are not listed in the published version of this paper, owing to an error made during submission. They realized the SIFD concept as a hardware design and carried out its functional testing.



This work is licensed under a Creative Commons Attribution 4.0 International License. *ASPDAC '27, Tokyo, Japan*

© 2027 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2923-2/2027/01

<https://doi.org/10.1145/3840404.3848186>

1 Introduction

Fault attacks are a practical threat to embedded devices with physically accessible interfaces. By disturbing the clock, supply voltage, or local electromagnetic environment, an adversary can induce transient processor faults and bypass security checks. Such attacks have been demonstrated on real products, including game consoles and electronic control units in autonomous vehicles [14, 15].

Defense in depth is essential for fault-attack countermeasures [2, 13]. Prior work has proposed countermeasures at different layers, including PLL-based sensors [11], software redundancy [26], random delays [22], and control-flow checks [24]. Each targets a different combination of fault-injection conditions and fault effects. A practical system therefore combines countermeasures from several layers, chosen according to its security requirements and implementation constraints [3].

Redundancy is a standard way to detect transient faults. Lockstep execution and triple modular redundancy (TMR) compare replicated executions and can detect a wide range of processor errors [12, 19]. Their cost, however, stems from duplicating or triplicating large parts of the processor, including execution units, register files, control logic, and pipeline state. This cost is difficult to justify in small embedded processors when the goal is to add one layer of fault-attack resistance rather than to build a fully fault-tolerant processor.

This paper proposes Staggered Instruction-Fetch Duplication (SIFD), a lightweight processor-level countermeasure based on instruction-fetch redundancy. The design is motivated by instruction skips, a common explanation of successful fault attacks. Such skips are often attributed to instruction-fetch faults [16, 17]. Instead of duplicating the processor core, SIFD fetches each instruction twice in consecutive cycles and compares the aligned copies before decode. In each fetch cycle, the two fetch paths access different program-counter values. A fault that corrupts one fetch cycle therefore affects two different instruction fetches, and each corrupted value is checked against a copy fetched in another cycle.

The contributions of this paper are as follows.

- We propose SIFD, a staggered instruction-fetch redundancy mechanism for detecting instruction-fetch faults induced by fault injection.
- We implement SIFD on a seven-stage RV32I processor. The implementation reuses the existing exception path and adds a duplicated fetch path, pipeline registers, and an instruction comparator.
- We evaluate the protected core on an Artix-7 FPGA. SIFD increases slice utilization by 15.0% and execution time by 16.3%.

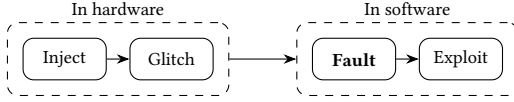


Figure 1: Countermeasures classified by the fault-propagation stage they target. SIFD acts at the Fault layer.

- We evaluate SIFD against physical fault injection. Using clock fault injection (CFI) and electromagnetic fault injection (EMFI) against the VerifyPIN benchmark [9], SIFD reduces the attack success rate from 0.86% to below 0.01% under CFI and from 0.32% to 0.13% under EMFI, while detecting 73.18% and 11.44% of the attempts, respectively.

2 Background

2.1 Fault Attacks

A common mechanism behind fault injection is a timing violation in the target processor. In CFI, the attacker supplies a malformed clock waveform and violates setup or hold constraints [16, 17]. Voltage fault injection can create a similar timing failure through a transient supply drop. In EMFI, the injected pulse couples into power-distribution or clock-distribution networks and can also induce timing violations [18].

Processor paths with tight timing margins are natural targets. Instruction fetch is one such path, especially when instructions are fetched from Flash or another memory structure outside the core. Prior studies have shown that such disturbances can result in instruction-fetch faults and instruction skips [16, 17]. This observation motivates a countermeasure that monitors the integrity of fetched instructions before they affect architectural state.

2.2 Fault-Attack Countermeasures

A fault attack proceeds through a chain of stages: physical injection, a resulting circuit-level glitch, a fault that propagates through the microarchitecture into the executed program, and finally exploitation of the faulty behavior [27]. A countermeasure can act at any point along this chain. Figure 1 groups countermeasures by the propagation stage they target, namely the Inject, Glitch, Fault, and Exploit layers [20, 27]. The Inject and Glitch layers act while the disturbance remains in hardware, and the Fault and Exploit layers act on its effects in the executed program. This grouping refers to where the fault has propagated, not to how a countermeasure is implemented. Inject-layer defenses target the physical injection, such as shielding and internal clocks. Glitch-layer defenses detect the circuit-level disturbance, such as voltage-drop detectors and electromagnetic sensors. Fault-layer defenses detect the erroneous effects of a fault after it propagates into execution. Exploit-layer defenses make those effects harder to exploit.

SIFD is a hardware mechanism that acts at the Fault layer. It does not prevent the physical injection itself. Instead, it detects a fault as it propagates, by comparing redundant instruction-fetch results before the decoded instruction is committed to the execution pipeline. A fetch-stage monitor can be combined with Inject-layer

and Glitch-layer defenses without requiring changes to application binaries.

2.3 Related Work

Processor-level countermeasures against fault attacks can be divided into two groups. The first group modifies the processor to add fault-attack resistance. A recent review summarizes countermeasures in this class [6]. These mechanisms often protect control flow or instruction integrity by adding compiler support, cryptographic checks, or dedicated control-flow monitoring. They provide stronger semantic guarantees than a fetch-focused mechanism, but they also complicate deployment through compiler changes, key management, or larger execution-time overhead. They are also typically evaluated by simulation, formal verification, or overhead measurement rather than by physical fault injection.

The second group is fault-tolerant processor design. Lockstep execution and TMR compare duplicated or triplicated executions and detect a wide range of transient errors. They have also been evaluated against fault injection [25]. Their drawback is the hardware cost of core-level duplication. In addition, a common-mode fault that affects corresponding stages in redundant cores can evade simple comparison.

SIFD is positioned between these two groups. It is not a full fault-tolerant processor, and it does not try to authenticate the complete instruction stream. It targets instruction-fetch faults with a small hardware change and no compiler support. Section 6 compares this trade-off with existing designs.

3 Staggered Instruction-Fetch Duplication

3.1 Design Rationale

SIFD is based on two design choices. First, the processor adds redundancy around instruction fetch rather than the execution pipeline. This avoids duplicating the execution units, register file, memory-access logic, and write-back logic. Second, the two fetches of the same instruction are staggered across consecutive cycles. This avoids comparing two copies that were fetched under the same physical disturbance.

The design targets faults that corrupt fetched instructions or cause instruction-fetch skips. It does not claim to detect all faults after the comparison point, such as faults confined to the execute or memory-access stages. This limitation is the cost of keeping the mechanism small. In a defense-in-depth design, SIFD should be combined with other countermeasures that cover post-decode faults.

3.2 Microarchitectural Operation

Figure 2 shows SIFD on a three-stage pipeline with instruction fetch (IF), instruction decode (ID), and execute (EX). The original IF stage is split into two fetch-alignment stages, IF1 and IF2. In IF1, the processor fetches two instructions: one from the current program counter, PC_t , and one from the next program counter, PC_{t+1} . The two fetched values are stored in IF1/IF2(1) and IF1/IF2(2), respectively. IF2 delays the PC_{t+1} path by one cycle. The comparator in the ID stage then compares two independently fetched copies of the same instruction.

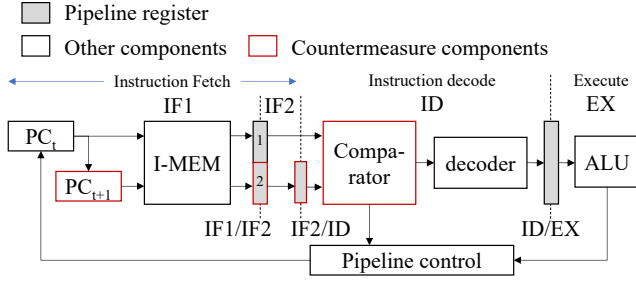


Figure 2: Overview of SIFD.

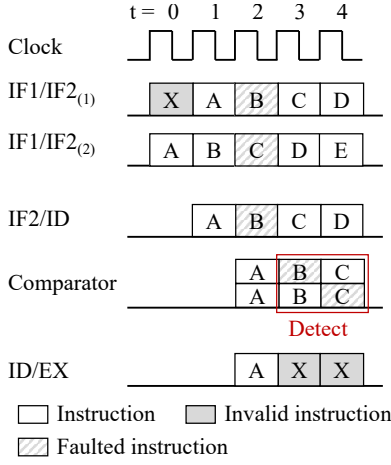


Figure 3: Timeline of SIFD. A fault injected into IF corrupts two staggered fetches and is detected when the corresponding instruction is compared in ID.

Figure 3 illustrates the timeline. The current-PC path fetches instructions A, B, C, and D in order. The next-PC path fetches B, C, D, and E in the same cycles. After the IF2 delay, the two copies of each instruction are aligned at the comparator. Under fault-free execution, both comparator inputs contain the same instruction and the pipeline proceeds normally.

When a fault corrupts a fetch cycle, it affects two different instruction fetches in the same cycle: one on the current-PC path and one on the next-PC path. Each corrupted value is compared with a copy fetched in a different cycle. The mismatch is detected when the affected instruction reaches the comparator. The processor then raises an exception through the existing exception path.

3.3 Security Rationale

The staggered schedule is the key difference from same-cycle duplication. If both copies of the same instruction are fetched in one cycle, CFI can corrupt both copies under the same timing condition. The comparator would then receive matching corrupted values. SIFD avoids this common-mode failure by ensuring that the two copies of an instruction are fetched in different cycles.

To bypass SIFD through instruction fetch, an attacker must maintain consistency across the staggered stream. A single-cycle disturbance corrupts two adjacent instruction fetches, but each is checked

against a copy from another cycle. Even a disturbance repeated over consecutive cycles cannot corrupt every instruction consistently: the instruction at the edge of the disturbance keeps one clean copy and is detected. Bypassing SIFD therefore requires confining a matched disturbance to an interior window without perturbing its boundary, which is stricter than inducing a single instruction skip.

A second bypass strategy is to inject a fault after the comparison point. SIFD does not rule out this strategy. The evaluation in Section 5 therefore includes EMFI, which can produce more localized effects than CFI.

4 Implementation

We implemented SIFD on a custom RISC-V processor. The processor implements RV32I with Zmmul, namely the base integer instruction set plus integer multiplication. Figure 4 shows the protected processor.

The baseline core has five stages: instruction fetch, instruction decode, execute, memory access (MA), and write back (WB). Applying SIFD adds two fetch-alignment stages, resulting in a seven-stage pipeline: IF1, IF2, IF3, ID, EX, MA, and WB. IF1 performs two instruction-memory accesses using PC_t and PC_{t+1} . IF2 aligns the staggered path. IF3 performs the instruction comparison and forwards a valid instruction to the decode stage. This placement differs from the conceptual design in Section 3, which locates the comparator in the ID stage. Because the ID stage already combines instruction decoding with register-file access, folding the comparison into it would increase its logic depth and delay. A dedicated comparison stage, IF3, keeps the per-stage delay balanced across the pipeline.

The processor fetches instructions from an instruction tightly coupled memory (ITCM) and accesses data through a data tightly coupled memory (DTCM). Both are mapped to distributed memory rather than block RAM (BRAM). The implementation includes no caches or prefetching, and a pre-decoder is placed before the decode stage.

The critical path is IF1, which fetches an instruction from the ITCM based on the PC value and pre-decodes it. The logic added by SIFD consists mainly of the second fetch path, the second pre-decoder, pipeline registers, and the comparator.

The comparator itself handles only instruction mismatch detection. Alert delivery reuses the existing exception mechanism. When IF3 detects a mismatch, it forwards the invalid instruction value 0. The invalid instruction reaches the later pipeline stages and triggers a custom RISC-V exception in MA. The exception handler prints an error message and enters an infinite loop. This handler is sufficient for detection-oriented evaluation. Recovery is left to future work.

5 Experiments

We evaluated SIFD from two perspectives: implementation overhead and fault-attack resistance. Both the baseline core and the protected core were implemented on an Arty A7-35T FPGA board.

5.1 Performance Evaluation

Table 1 shows the synthesis and benchmark results. The circuits were synthesized using Vivado 2022.1. We used CoreMark [10] as

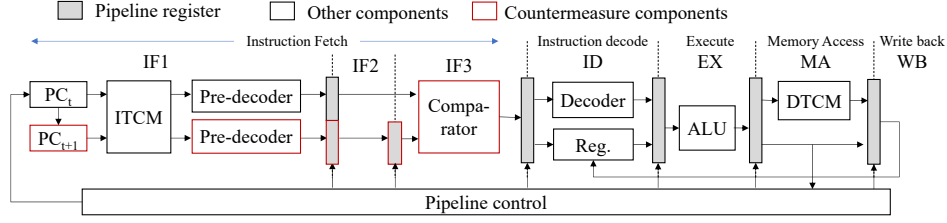


Figure 4: Seven-stage RISC-V processor with SIFD. For clarity, the ITCM and DTCM are drawn inside the processor, although they lie outside the core boundary in the implementation.

Table 1: Performance evaluation results of Staggered Instruction-Fetch Duplication.

	Baseline core	Protected core	Overhead
LUT	1,926	2,238	16.2%
FF	1,638	2,128	29.9%
Slice	767	882	15.0%
BRAM	0	0	0.0%
DSP	4	4	0.0%
Fmax [MHz]	80	80	0.0%
CoreMark/MHz	2.089	1.796	14.0% ¹

¹This CoreMark/MHz reduction corresponds to a 16.3% increase in execution time.

the benchmark and report CoreMark/MHz. The maximum operating frequency was 80 MHz for both cores. We operated the FPGA at 50 MHz for stable fault-injection experiments.

SIFD increases LUTs by 16.2%, FFs by 29.9%, and slices by 15.0%. The FF overhead is the largest because SIFD adds pipeline registers for the staggered fetch path. DSP usage is unchanged, as SIFD adds no arithmetic unit. The maximum operating frequency also remains unchanged at 80 MHz.

The 14.0% CoreMark/MHz reduction comes mainly from the longer pipeline. Splitting instruction fetch increases the control-hazard penalty, and CoreMark is branch-intensive. The measured overhead is therefore consistent with a fetch-stage mechanism that adds pipeline depth but does not change the execution units.

5.2 Fault-Attack Resistance Evaluation

5.2.1 Experimental Setup. We used *VerifyPIN*, a PIN-authentication benchmark for evaluating fault-injection countermeasures [9]. *VerifyPIN* compares a user PIN against a fixed reference PIN. The user PIN is hardcoded to an incorrect value, so a fault-free run always rejects it. A fault attack aims to force acceptance of this incorrect PIN, for example by corrupting the compared data or the branch condition that governs the accept/reject decision. The target execution interval was observed through a GPIO signal, and the same signal was used as the trigger for fault injection. This setup allowed us to scan the relevant injection timings exhaustively.

Figure 5 shows the experimental setup. We used two injection methods. CFI was performed using a ChipWhisperer Husky. CFI directly manipulates the clock and provides precise timing control. EMFI was performed using a ChipSHOUTER. EMFI provides

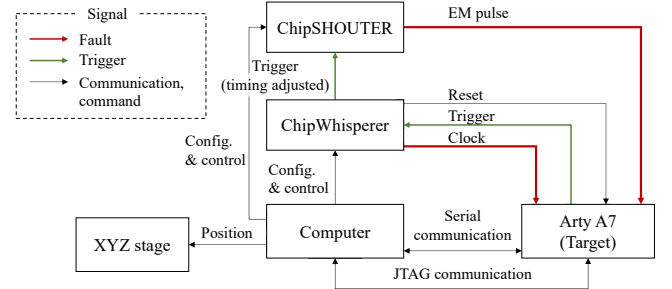


Figure 5: Experimental setup for fault-attack evaluation.

Table 2: CFI parameter settings. *glitch width* and *glitch offset* are in ChipWhisperer phase-shift steps; *ext_offset* is in clock cycles.

Parameter	Value	Step
<i>glitch width</i>	0–330	10
<i>glitch offset</i>	0–330	10
<i>ext_offset</i>	0–85	1 (baseline core)
	0–105	1 (protected core)

spatial selectivity and can induce local effects in specific regions of the FPGA. An XYZ stage controlled the injection position. The ChipWhisperer adjusted the injection timing relative to the trigger signal.

Tables 2 and 3 show the injection parameters. For CFI, *glitch width* and *glitch offset* set the shape and position of the glitch within the clock cycle. They are given as ChipWhisperer phase-shift steps, and we swept both over their full configurable range. *ext_offset* specifies the injection timing in clock cycles after the trigger¹. For EMFI, the Z coordinate was fixed. The pulse strength was controlled by the *voltage*. The XY scan covered the 15 mm × 15 mm package area with an 8 × 8 grid of injection positions, as shown in Fig. 6. For CFI, we performed 99,416 injections on the baseline core and 122,536 injections on the protected core. For EMFI, we performed 40,448 injections on the baseline core and 51,712 injections on the protected core.

¹The *ext_offset* range differs between the two cores because the protected core has additional pipeline stages. The EMFI ranges are slightly shorter than the CFI ranges because the trigger for EMFI passes through the ChipWhisperer for timing adjustment, which adds a small delay.

Table 3: EMFI parameter settings. The voltage range was selected from preliminary experiments, from the onset of observable fault effects to the range where crash/hang outcomes became frequent.

Parameter	Value	Step
<i>voltage</i>	380–450 V	10 V
<i>X</i>	0.50–14.50 mm	2.00 mm
<i>Y</i>	0.50–14.50 mm	2.00 mm
<i>ext_offset</i>	0–78	1 (baseline core)
	0–100	1 (protected core)

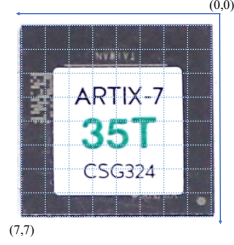


Figure 6: EMFI injection area.

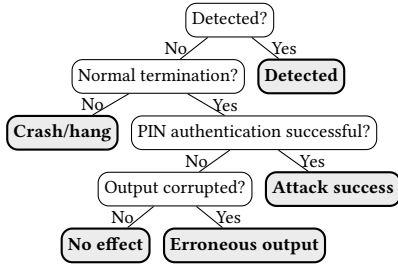


Figure 7: Classification of fault-injection results.

5.2.2 Classification of Fault-Injection Results. Figure 7 shows the classification used for the fault-injection results. A run is classified as Detected when the protected core raises the SIFD exception. If the run does not complete normally and no SIFD exception is observed, it is classified as Crash/hang. This category includes processor crashes, dead states, and runs that do not return a valid response. A normally terminated run is classified as Attack success when PIN authentication succeeds under an incorrect PIN. If authentication fails but the output is corrupted, the run is classified as Erroneous output. Otherwise, the run is classified as No effect.

5.2.3 Experimental Results. Table 4 shows the CFI and EMFI results. Figures 8 and 9 show the distributions of fault sensitivity under CFI and EMFI, respectively.

The results provide three observations. First, SIFD changes many CFI-induced fault effects into detections. On the baseline core, CFI produced 853 attack successes. On the protected core, only four attack successes remained, and 89,668 attempts were detected. This result is consistent with the design rationale: CFI disturbs clock timing globally, and SIFD is designed to expose cycle-level fetch

Table 4: Fault-attack resistance evaluation results.

(a) CFI.				
	Baseline core		Protected core	
Attack success	853	(0.86%)	4	(<0.01%)
Erroneous output	5,580	(5.61%)	10	(0.01%)
Crash/hang	800	(0.80%)	24	(0.02%)
No effect	92,183	(92.72%)	32,830	(26.79%)
Detected	0	(0.00%)	89,668	(73.18%)

(b) EMFI.				
	Baseline core		Protected core	
Attack success	128	(0.32%)	66	(0.13%)
Erroneous output	784	(1.94%)	174	(0.34%)
Crash/hang	6,698	(16.56%)	4,607	(8.91%)
No effect	32,838	(81.19%)	40,949	(79.19%)
Detected	0	(0.00%)	5,916	(11.44%)

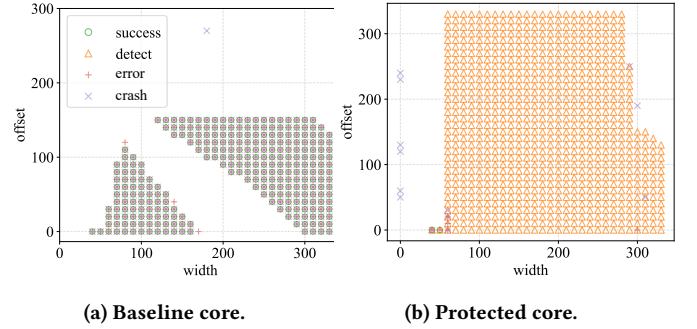


Figure 8: Fault-sensitivity distribution under CFI. The labels *success*, *detect*, *error*, and *crash* denote attack success, detected cases, erroneous output, and crash/hang, respectively. No-effect cases are omitted.

corruptions through staggered comparison. The four remaining attack successes are most likely faults that took effect after the instruction-fetch stage, beyond the reach of a fetch-stage comparison.

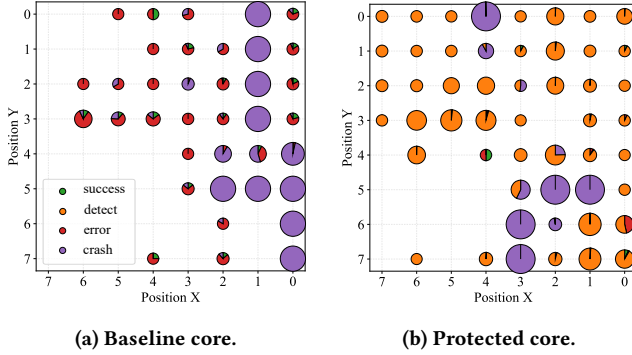
Second, SIFD also reduces harmful EMFI outcomes, but the reduction is smaller than under CFI. The attack success rate decreases from 0.32% to 0.13%, and the erroneous-output rate decreases from 1.94% to 0.34%. The protected core detects 11.44% of EMFI attempts. This behavior is consistent with EMFI spatial selectivity. EMFI can induce localized faults in circuit blocks or later pipeline stages that are not fully covered by a fetch-stage comparison.

Third, the detected regions overlap with the sensitive regions of the baseline core. In Fig. 8, detected cases appear over the fault-intensity region where the baseline core exhibits attack success, erroneous output, and crash/hang outcomes. In Fig. 9, detected cases are distributed across many injection positions. SIFD is not a physical sensor, but these results show that a fetch-stage logic monitor can provide sensor-like coverage for a broad set of fault-injection attempts in this implementation.

Table 5: Comparison with existing studies. Resource is the resource of the countermeasure-applied processor, and the overhead columns are relative to each design’s baseline.

Ref.	Year	Countermeasure	Class ¹	Resource ovh.	Resource	Exec. time ovh. ²	Device	Compiler ³	Enc. ⁴	Attack eval. ⁶
[19]	2012	TMR [12]	T	200%	2,150 slice	≈0%	Virtex-5	No	No	emul.
[19]	2012	Enhanced Lockstep	T	197%	1,664 slice	≈0%	Virtex-5	No	No	emul.
[5]	2024	Buffered TMR [4]	T	16% 9%	6,429 LUT 4,905 FF	≈+200%	Kintex-7	No	No	sim.
[5]	2024	Dynamic TMR	T	25% 12%	6,923 LUT 5,019 FF	≈+100%	Kintex-7	No	No	sim.
[8]	2017	SOFIA	R	28.2%	7,551 slice	13.7%	Virtex 6	No	Yes	none
[23]	2018	SCFP	R	47% ⁷	90 kGE	9.1%	ASIC 65 nm	Yes	Yes	none
[21]	2020	CONFIDAENT	R	N/A	N/A	20%–220% ⁵	FPGA	Yes	Yes	none
[1]	2022	Instruction Replay	R	11% 29%	809 LUT 389 FF	99%	Zynq-7020	No	No	sim.
[7]	2023	MAFIA CRC	R	6.5%	55 kGE	18.4%	ASIC FDSOI 22 nm	Yes	No	form.
[7]	2023	MAFIA CBC-MAC	R	23.8%	64 kGE	39.0%	ASIC FDSOI 22 nm	Yes	Yes	form.
Ours	2027	SIFD	R	16.2%	2,238 LUT	16.3%	Artix-7	No	No	phys.
				29.9%	2,128 FF					
				15.0%	882 slice					

¹T, fault tolerance; R, fault-attack resistance. ²≈ denotes an estimated value. ³Whether compiler modification is required. ⁴Whether encryption is used. ⁵CONFIDAENT has several variants, ranging from basic protection to stronger protection. ⁶Fault-behavior evaluation: phys., physical fault injection on hardware; emul., hardware bit-flip emulation; sim., simulation-based injection; form., formal verification; none, no fault-injection evaluation. ⁷SCFP accounts for 32% of a 90 kGE ASIC; the listed figure is the area overhead relative to the baseline core.

**Figure 9: Fault-sensitivity distribution under EMFI. The circle size represents the number of cases excluding no-effect cases. Each pie chart shows the composition of fault effects at the corresponding injection position. The labels follow Fig. 8.**

6 Comparison with Existing Studies

Table 5 compares SIFD with representative processor-level designs for fault tolerance and fault-attack resistance. We select entries by one criterion: processor-level designs that add redundancy or modify the processor for fault resistance and that report per-design hardware or execution-time overhead. This criterion admits two groups: redundancy baselines, which bound the cost of full replication, and processor-modification fault-attack countermeasures, the class to which SIFD belongs. The entries use different devices, processors, and benchmarks, so the table supports a qualitative design comparison rather than a controlled performance ranking.

Compared with lockstep execution and TMR, SIFD substantially reduces hardware overhead because it does not replicate execution units or complete processor state. In return, it covers a narrower range of faults: SIFD targets instruction-fetch faults, whereas full redundancy detects a wider range of transient processor faults.

The buffered and dynamic TMR variants report far larger execution-time overheads, roughly doubling to tripling execution time, because they interleave multiple threads on a single datapath rather than because of the redundancy itself.

Compared with processor-modification-based fault-attack countermeasures, SIFD needs no compiler modification and no cryptographic engine or key management, and its execution-time overhead is lower than MAFIA CRC, MAFIA CBC-MAC, and CONFIDAENT. SCFP and SOFIA report lower execution-time overhead but use encryption, and SCFP also needs compiler support. Instruction replay likewise needs no compiler or cryptography, but it protects the data flow rather than instruction fetch and reports a much larger execution-time overhead. Because SIFD duplicates only the instruction-fetch path, its added logic does not grow with core size. The relative overhead measured here on a small RV32I core is therefore a high estimate and is expected to decrease on larger processors. Overall, SIFD occupies a distinct point in the design space: a standalone instruction-fetch countermeasure with no compiler or cryptographic dependency.

The listed studies also differ in how fault behavior is evaluated. Most report simulation, formal verification, or overhead results, and few evaluate against physical fault injection. SIFD is evaluated with physical CFI and EMFI. Its detection rates in Section 5 are therefore measured under physical injection rather than estimated in simulation.

7 Conclusion

This paper proposed Staggered Instruction-Fetch Duplication (SIFD), a lightweight processor-level countermeasure that fetches each instruction twice across consecutive cycles to detect instruction-fetch corruptions without duplicating the processor core.

We implemented SIFD on an RV32I processor and evaluated it on an Artix-7 FPGA. It increases slice utilization by 15.0% and execution time by 16.3%, while reducing attack success to below 0.01% under CFI and to 0.13% under EMFI.

Future work spans three directions. First, broadening fault coverage by combining SIFD with complementary countermeasures, such as software-level checks or lightweight sensors, in a defense-in-depth design that covers post-decode faults without enlarging SIFD itself. Second, adding recovery beyond detection, for example through instruction-fetch triplication or checkpoint-based rollback after a detected mismatch. Third, applying SIFD to fetch front-ends with instruction caches, prefetching, or speculative fetch, which the tightly coupled memory implementation in this paper does not model.

References

- [1] Noura Ait Manssour, Vianney Lapotre, Guy Gogniat, and Arnaud Tisserand. 2022. Processor Extensions for Hardware Instruction Replay against Fault Injection Attacks. In *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 26–31.
- [2] Tamas Ban and David Hu. 2026. Physical attack mitigation in Trusted Firmware-M. https://trustedfirmware-m.readthedocs.io/en/latest/design_docs/tfm_physical_attack_mitigation.html. Accessed 13 July 2026.
- [3] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. 2006. The Sorcerer's Apprentice Guide to Fault Attacks. *Proc. IEEE* 94, 2 (2006), 370–382.
- [4] Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri. 2022. Design and evaluation of buffered triple modular redundancy in interleaved-multi-threading processors. *IEEE Access* 10 (2022), 126074–126088.
- [5] Marcello Barbirotta, Francesco Menichelli, Abdallah Cheikh, Antonio Mastrandrea, Marco Angioli, and Mauro Olivieri. 2024. Dynamic Triple Modular Redundancy in Interleaved Hardware Threads: An Alternative Solution to Lockstep Multi-Cores for Fault-Tolerant Systems. *IEEE Access* (2024).
- [6] Roua Boulifa, Giorgio Di Natale, and Paolo Maistri. 2025. Countermeasures Against Fault Injection Attacks in Processors: A Review. *Information* (2078-2489) 16, 4 (2025).
- [7] Thomas Chamelot, Damien Couroussé, and Karine Heydemann. 2023. MAFIA: Protecting the Microarchitecture of Embedded Systems Against Fault Injection Attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (2023), 4555–4568.
- [8] Ruan De Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. 2017. SOFIA: Software and control flow integrity architecture. *Computers & Security* 68 (2017), 16–35.
- [9] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. 2016. FISSC: A Fault Injection and Simulation Secure Collection. In *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*. 3–11.
- [10] Shay Gal-On and Markus Levy. 2009. *Exploring CoreMark: A Benchmark Maximizing Simplicity and Efficacy*. White Paper. Embedded Microprocessor Benchmark Consortium (EEMBC). Accessed 13 July 2026.
- [11] Wei He, Jakub Breier, Shivam Bhasin, Noriyuki Miura, and Makoto Nagata. 2017. An FPGA-compatible PLL-based sensor against fault injection attack. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 39–40.
- [12] Yoshihiro Ichinomiya, Shiro Tanoue, Motoki Amagasaki, Masahiro Iida, Morihiro Kuga, and Toshinori Sueyoshi. 2010. Improving the Robustness of a Softcore Processor against SEUs by Using TMR and Partial Reconfiguration. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 47–54.
- [13] Zahra Kazemi, David Hely, Mahdi Fazeli, and Vincent Beroulle. 2020. A Review on Evaluation and Configuration of Fault Injection Attack Instruments to Design Attack Resistant MCU-Based IoT Applications. *Electronics* 9, 7 (2020). <https://doi.org/10.3390/electronics9071153>
- [14] Nicolas Kühnapfel, Christian Werling, and Hans Niklas Jacob. 2023. Back in the Driver's Seat: Recovering Critical Data from Tesla Autopilot Using Voltage Glitching. https://media.ccc.de/v/37c3-12144-back_in_the_driver_s_seat_recovering_critical_data_from_tesla_autopilot_using_voltage_glitching. Accessed 18 January 2024.
- [15] Yifan Lu. 2019. Injecting Software Vulnerabilities with Voltage Glitching. <https://doi.org/10.48550/ARXIV.1903.08102>
- [16] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. 2013. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*. IEEE, 77–88.
- [17] Nicolas Moro, Karine Heydemann, Amine Dehbaoui, Bruno Robisson, and Emmanuelle Encrenaz. 2014. Experimental evaluation of two software countermeasures against fault attacks. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 112–117.
- [18] Roukoz Nabhan, Jean-Max Dutertre, Jean-Baptiste Rigaud, Jean-Luc Danger, and Laurent Sauvage. 2024. EM Fault Injection-Induced Clock Glitches: From Mechanism Analysis to Novel Sensor Design. In *2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 1–7.
- [19] Hung-Manh Pham, Sebastien Pillement, and Stanislaw J Piestrak. 2012. Low-Overhead Fault-Tolerance Technique for a Dynamically Reconfigurable Softcore Processor. *IEEE Trans. Comput.* 62, 6 (2012), 1179–1192.
- [20] Raelize. 2020. Fault Injection Reference Model (FIRM). <https://raelize.com/blog/raelize-fi-reference-model/>. Accessed 12 September 2024.
- [21] Olivier Savry, Mustapha El-Majhihi, and Thomas Hiscock. 2020. Confidaent: Control Flow protection with Instruction and Data Authenticated Encryption. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*. IEEE, 246–253.
- [22] Michael Tunstall and Olivier Benoit. 2007. Efficient use of random delays in embedded software. In *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems: First IFIP TC6/WG 8.8/WG 11.2 International Workshop, WISTP 2007, Heraklion, Crete, Greece, May 9-11, 2007. Proceedings 1*. Springer, 27–38.
- [23] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. 2018. Sponge-Based Control-Flow Protection for IoT Devices. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 214–226. <https://doi.org/10.1109/EuroSP.2018.00023>
- [24] Mario Werner, Erich Wenger, and Stefan Mangard. 2016. Protecting the control flow of embedded processors against fault attacks. In *Smart Card Research and Advanced Applications: 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers 14*. Springer, 161–176.
- [25] Nils Wiersma and Ramiro Pareja. 2017. Safety!= Security: On the resilience of ASIL-D certified microcontrollers against fault injection attacks. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 9–16.
- [26] Marc Witteman and Martijn Oostdijk. 2008. Secure application programming in the presence of side channel attacks. In *RSA conference*, Vol. 2008.
- [27] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. 2018. Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. *Journal of Hardware and Systems Security* 2, 2 (2018), 111–130.