Swift-Healer: Firmware-Reconfigurable Self-Healing for Remote Glitch-Injection on Autonomous Navigation Systems

Ali Suvizi
George Washington University
Washington, DC, USA
ali.suvizi@gwu.edu

Joshua Iwu
George Washington University
Washington, DC, USA
jiwu42@email.gwu.edu

Kostas Amberiadis
National Institute of Standards and Technology
Gaithersburg, MD, USA
kostas.amberiadis@nist.gov

Guru Venkataramani
George Washington University
Washington, DC, Germany
guruv@gwu.edu

Abstract

Autonomous Navigation Systems (ANS) incorporate many safety-critical functions, such as collision avoidance. Recent studies have shown how remote clock/voltage glitch injections pose an imminent threat to mission-sensitive modules in the autonomous navigation domain: timing/power perturbations in the perception stages can cascade into severe accuracy loss, and latency drift for downstream tasks. In this paper, we present *Swift-Healer*, a firmware-reconfigurable self-healing architecture that unifies prediction-detection modules and an automated healing unit to mitigate remote clock/voltage glitches, while satisfying the application latency constraints. Our solution leverages a chiplet-based architecture that offers isolation from compromised hardware modules, while enabling self-healing in the firmware management layer. We implement our design on a Zynq−7000 with a hardware accelerator, where *Swift-Healer* predicts glitches within ANS kernels up to two real-time loop iterations earlier ($\approx 0.06ms$), thereby giving abundant time for self-healing. When the prediction confidence is low, the reactive detector provides a fallback path for rapid fault detection.

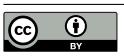
CCS Concepts

• **Computer systems organization** → **Firmware**; **Reliability**; **Reconfigurable computing**; • **Hardware** → **Failure recovery, maintenance and self-repair**; **Safety critical systems**.

Keywords

Self-Healing Systems, Remote Glitch Injection, Chiplet Architectures, Field Programmable Gate Arrays, Firmware

ACM Reference Format:
Ali Suvizi, Joshua Iwu, Kostas Amberiadis, and Guru Venkataramani. 2026. Swift-Healer: Firmware-Reconfigurable Self-Healing for Remote Glitch-Injection on Autonomous Navigation Systems. In *Great Lakes Symposium on VLSI 2026 (GLSVLSI '26), June 22–24, 2026, Canandaigua, NY, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3787109.3815273



GLSVLSI '26, Canandaigua, NY, USA

ACM ISBN 979-8-4007-2431-2/26/06
https://doi.org/10.1145/3787109.3815273

1 Introduction

Mission-critical applications with autonomous navigation abilities in the ground, aerial, and maritime domains, which demand high accuracy and strict real-time deadlines, making resilience a crucial aspect of their system design [8, 14, 20]. In many such applications, their execution pipeline stages (perception → planning → control) hinge on timely and correct perception [6, 10, 28]; any delay at this stage risks cascading failures downstream. In real-world deployments, *over-the-air* (OTA) updates have become a practical mechanism to sustain the deployed vehicles and minimize their service visits. OTA improves functionality, but by exposing the update pathways, it also broadens the attack surface. Other platform features, such as dynamic voltage and frequency scaling (DVFS), are used routinely to balance performance and battery life [26, 31, 34]. Yet, all of these features can also introduce avenues for adversaries to influence timing and power, if insufficiently protected [6].

Perception stage is highly sensitive to timing and power perturbations [32]. Clock/voltage glitches have recently been shown to be executed both physically and remotely [17, 29], facilitated by abusing OTA-enabled software voltage/frequency control surfaces [33]. Exploiting these interfaces, particularly during update windows, induces glitches on vision accelerators [34], leading to silent data corruption (SDC), and misclassification [5, 15]. Hence, self-healing is essential for sustained system operation. Unfortunately, prior defense methods are *reactive-only* and intervene only after a glitch has already manifested, relying on additional hardware sensors to raise alarms upon evidencing the attack manifestation. In many autonomous navigation pipelines, the time budget to decide and transition to a safe mode across the end−to−end pipeline is very tight (e.g., in the self-driving cars, this is under 100 ms), where depending on only reactive detection can be catastrophic [12, 18, 19, 34].

In this paper, we present *Swift-Healer*, a firmware-reconfigurable self-healing architecture that targets and heals clock/voltage glitch attacks against the perception stage in an *automated* fashion. The firmware layer provides unique visibility *across* system operations, while ensuring rapid healing through its direct access to the hardware. Swift-Healer predicts impending anomalies using telemetry data, thereby creating a pre-failure mitigation window, quickly detects residual timing and accuracy drifts, and executes self-healing in firmware. Swift-Healer offers a *chiplet-based solution*, providing frequency diversity and isolation from compromised modules for self-healing.

In summary, the key contributions of our paper are:

- We present *Swift-Healer*, an autonomous predict–detect–heal firmware controller that leverages hardware support to minimize latency and maintain real-time availability.
- Our design leverages system telemetry for *proactive* prediction and enables fallback reactive detection where needed, all while improving the ability to meet tight deadlines for decision making in mission-critical applications.
- Our design enables healing via firmware, which can continue operations even if the primary module gets compromised.
- Our Proof-of-concept implementation using Zynq–7000 SoC shows that *Swift-Healer* can predict glitches by about 0.06 ms in advance, thereby providing a sufficient time window for restoring the perception module to its original accuracy levels.

2 Background

According to ISO 26262, the international standard for functional safety in road vehicles, Autonomous Driving System (ADS) and robotics are mission-critical systems whose functions must satisfy hazard-driven safety goals within stringent timing constraints [9]. OTA updating is now a runtime necessity for ADS. Original Equipment Manufacturers (OEMs) deliver bug fixes and feature rollouts directly from cloud backends to vehicles over cellular/Wi-Fi, with updates scheduled and installed in-vehicle (e.g., Tesla's production OTA program [30]; Ford's SYNC "Power-Up" updates) [4, 13]. In parallel, DVFS is indispensable in ANS compute stacks. Heterogeneous and workload-variable perception requires on-the-fly power-performance trade-offs under tight battery budget, which modern SoCs expose via firmware governors and clock managers [1, 31]. Deadline-aware DVFS policies have been shown to preserve real-time guarantees while reducing energy consumption [22, 27, 31].

3 Threat Model

The perception stage of many mission-critical autonomous navigation systems is critical as it processes raw sensor streams under tight real-time deadlines. Perturbations at this stage can cause misclassification or latency drift, which may propagate through the pipeline and lead to deadline misses in a safety-critical loop [6, 32]. Our threat model targets *remote clock/voltage glitch injection* against the perception accelerator through software-accessible DVFS control surfaces. We assume a remote adversary with software-level access to voltage/frequency controls (e.g., through a compromised OTA agent). The attacker does not require physical access to manipulate v/f settings at runtime to induce harmful timing/power perturbations in the accelerator [16, 33, 34]. Prior work shows that such DVFS-driven perturbations can degrade accuracy, trigger SDC, and cause crashes in hardware accelerators [7, 20, 23]. As illustrated in Fig. 1, abusing the OTA/DVFS control path can degrade perception integrity and timeliness, leading to cascading downstream errors.

4 Design Overview of Swift-Healer

Swift-Healer architecture, as shown in Fig. 2-a, comprises three co-operating subsystems, making a closed loop of predict–detect–heal: (i) a **Monitor** that runs a lightweight proactive predictor and reactive detector over telemetry data and runtime feature sequences;

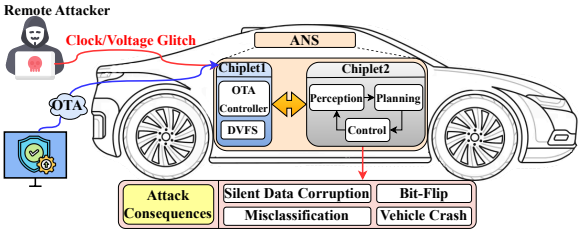


Figure 1: Remote glitch attacks affecting ANS perception

(ii) a **Firmware Controller** that implements and orchestrates self-healing; and (iii) a **Healer** module that implements methods for healing post-analysis.

4.1 Monitor

Glitch Predictor (Proactive): We collect sub-millisecond telemetry during live perception runs and align traces to accelerator *control cycles* (real-time loop iterations) for the training dataset. Each sample is represented by an 8-dimensional feature vector comprising: (1) frequency and voltage, (2) temperature, (3) accelerator latency/accuracy proxies, (4) DMA read/write bytes, (5) Advanced eXtensible Interface (AXI) performance counters, (6) `jitter`, (7) a canary delay-line, and (8) `pll_lock_toggle` events. We use raw values and first-order deltas, normalized to a benign baseline per bitstream, and feed them to a gated recurrent unit (GRU) model to forecast remote clock/voltage–glitch events. The GRU uses 8-sample windows, a 64-unit hidden state, and a two-control-cycle forecast horizon, and is trained offline with RMSProp, batch size 32, and MSE loss. During evaluation, a window is labeled as *glitch* if a glitch onset occurs within the next two control cycles, and *normal* otherwise. At runtime, the forecast is compared against predefined safety thresholds derived from board limits and refined through repeated benign-mode runs on the target platform to issue an early prediction alarm. To collect sufficient samples of the glitch cases, we oversample glitch instances across four main families, viz., overclock, undervolt, clock oscillation, and voltage oscillation via cross-varied onset times, durations, and amplitudes. The broad fault space is produced via (i) software-exposed DVFS perturbations and (ii) parameterized glitch injection with controllable onset, duration, and amplitude. Our representative offline training data span diverse temperatures, DVFS points, input scenes, accelerator configurations, and supply conditions. We reserve a portion of this dataset for testing under held-out stress conditions on the prototype platform. Our predictor gives an early signal of the event that might cross the threshold for voltage/clock glitching attacks.

Glitch Detector (Reactive): When the predictor's risk score falls below the calibrated threshold, the reactive detector compares live telemetry against calibrated nominal references. Monitored channels include: core voltage (V_{CCINT}) and its slope $\Delta V/\Delta t$; clock frequency, jitter, and PLL lock status from the clocking network; on-chip canary timing shift; AXI throughput and stall ratios; and accelerator latency and inference accuracy. Detection uses multi-sensor fusion with channel-specific thresholds. Our anomaly detection method strengthens its confidence by spotting correlated deviations involving multiple channels, such as a voltage droop correlated highly with a clock frequency deviation, a canary delay excursion accompanied by a latency spike. Such events are tagged as anomalies and lead to *automatically* trigger self-healing module.

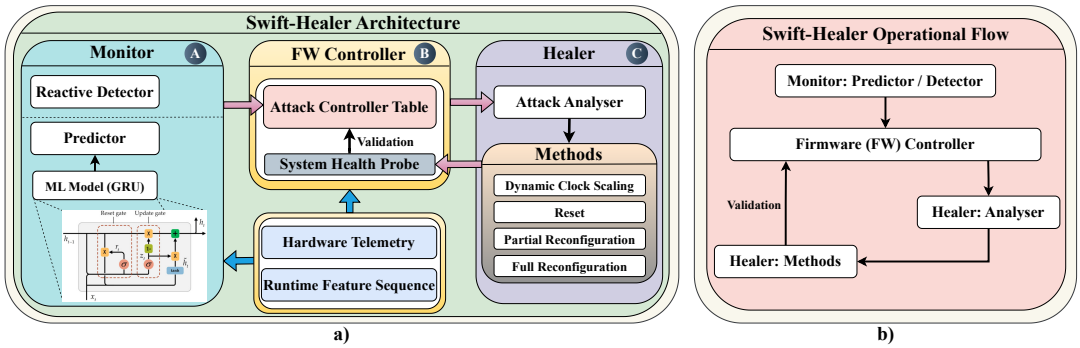


Figure 2: a) Swift-Healer architecture: a firmware-level Self-Healing framework that incorporates Monitor, Controller, and Healer (blue arrows indicate telemetry data flow; red arrows indicate control signals). b) Operational flow diagram between the key modules at runtime.

4.2 Firmware Controller

The *Firmware (FW) Controller* orchestrates the coordination between the Monitor and Healer modules at runtime. It boots the system by enabling the Monitor and then streaming the board- and application-level telemetry to it, while keeping the Healer dormant. When the Monitor performs a *predict* or *detect* action, the Controller *automatically* activates the Healer. Upon activation, the *Analyser* characterizes the fault (type, domain, affected features) using the event metadata and recent telemetry context. The controller then consults an *Attack Controller Table (ACT)*, using a Look-up table that maps the input vectors (attack class, affected features, system constraints) to an efficiency-ordered set of healing actions, and selects the most cost-effective corrective action that satisfies time deadline and power constraints. After execution, a *System Health Probe* validates recovery by checking post-healing system health indicators. After this point, the system returns to monitoring-only mode. During this process, the firmware controller logs the outcomes of our self-healing loop into the ACT (e.g., updated state, action priority, thresholds), thereby ensuring that ineffective methods are not repeatedly triggered.

4.3 Healer

The *Healer* is invoked when the Firmware Controller receives a positively identified attack event from the Monitor block.

Analyser: Based on the event metadata (type, severity, confidence), the Analyser classifies the fault along three axes: *type* (timing vs. voltage signature), *domain* (clock island, fault location), and *severity* (number of affected features). Classification uses low-cost rules over correlated features (e.g., voltage droop plus loss of PLL lock indicates a power–clock coupled glitch).

Healer Methods: Our Healer design currently supports four firmware-level methods ordered by increasing complexity:

(1) **Dynamic Clock Scaling (DCS)**: Shift the affected clock to a safe point, verify stability with a lightweight health probe, and gradually restore frequency using profiling results.
(2) **Reset**: Assert a targeted reset with integrity checks.
(3) **Partial Reconfiguration** (PR): Decouple the target region to isolate it and prevent fault propagation, stream a signed and resilient bitstream from secure storage, reattach interfaces, and run a post-PR health check.
(4) **Full Reconfiguration** (FR): Reload the bitstream to return to a clean baseline when multiple regions are corrupted.

We assume that our *Healer* methods are pre-checked rigorously for correctness using commercial/open-source tools in the market for FPGA modules. We also note that there are opportunities to dynamically order the healing methods based on system conditions and attack severity, and will consider this aspect in our future work.

4.4 Swift-Healer Operational Flow

Swift-Healer runs as a continuous, firmware-only loop *as shown in Fig. 2-b*: *Predict → Detect → Analysis → Heal → Validate*. During steady state, only the Monitor is active. The Analyser classifies the fault, and the Firmware Controller indexes into the ACT to select the appropriate healing action. An independent system health probe module confirms recovery from the glitching attack. This closed-loop design provides an *automated framework* for early warning, rapid healing, and post-healing assurance. Our Swift-Healer framework preserves availability for mission-critical operations.

5 Experimental Setup

Platform (Hardware): We implement Swift-Healer on an Xilinx Zynq-7000 SoC (Z-7010), which provides a compact heterogeneous platform with a processing system (PS) and programmable logic (PL) fabric. In our prototype, the PS executes the firmware controller and monitoring tasks, while the PL hosts the perception accelerator inside a designated partially reconfigurable region. This prototype is not intended to replicate the full complexity of a production automotive SoC; rather, it provides the essential architectural elements required to evaluate Swift-Healer: heterogeneous compute/control partitioning, runtime telemetry collection, independent clock domains, and localized recovery through partial reconfiguration. Accordingly, our chiplet-style decomposition into a PS domain, an accelerator domain, and a recovery-support domain is used to validate the control loop and recovery mechanisms under realistic timing constraints on a resource-constrained platform.

Edge-Detector Accelerator: We use edge detection as a perception workload because it is latency-sensitive, widely used in the perception stage as a preliminary step for image processing [3, 24], and effective for exposing the impact of timing/power perturbations and recovery actions under strict real-time constraints. We implement a resource-efficient Sobel accelerator in AMD Vitis™ HLS and integrate it as a reusable Vivado PL IP core.

Firmware Controller: After integrating the PL design and chiplet-style interfaces in Vivado, we implement the PS software in *AMD Vitis™*. The PS configures the accelerator via AXI4-Lite, manages

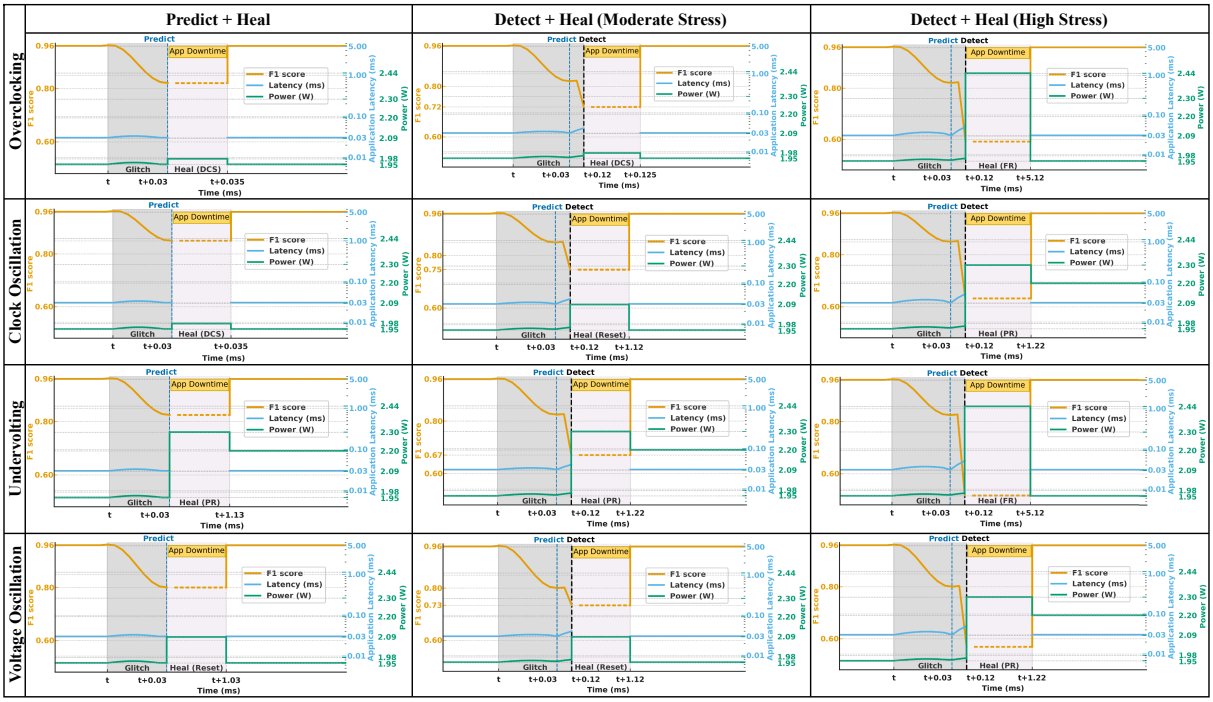


Figure 3: Evaluation of Swift-Healer across *four glitch families* (rows: overclocking, clock oscillation, undervolting, voltage oscillation) and *three healing scenarios* (columns: Predict + Heal; Detect + Heal—Moderate Stress; Detect + Heal—High Stress). Each panel shows F1 (left axis), and accelerator latency (ms) and power (W) on the right axis. The gray region marks the glitch interval; the blue marker denotes the *predict* alarm; the black dashed marker denotes the *detect* alarm; the pink bar/orange rectangle indicates application downtime during healing.

AXI4-Stream DMA descriptors to send/receive image frames, and timestamps transfers for per-frame latency accounting. A lightweight firmware stack samples board telemetry and exposes them to the *Monitor Block*, which runs the GRU-based *predictor* and the threshold-fusion *detector* described in §4.1. Confirmed events are delivered to the *Firmware Controller* (§4.2), which selects self-healing actions. The PS emulates OTA client by hosting the *DVFS* controller.

Glitch Injection: We emulate remote fault stimuli that perturb the timing/power integrity of the accelerator, consistent with software-exposed DVFS control abuse and unstable power/clocking.

Clock-path injections. To model frequency steps and oscillatory instabilities, we modulate the accelerator clock via a PR-safe clock path. A clock mux switches between nominal and perturbed frequencies, and a scheduler drives duty/period to create *clock oscillation* patterns. We also induce *overclock* episodes by brief excursions above the validated operating point and assert phase/frequency changes that momentarily de-assert PLL lock.

Voltage-path injections. To approximate rail disturbances, we use two controlled mechanisms: (i) software-exposed DVFS perturbations to create *undervolt* episodes and (ii) PL-generated dynamic load steps (toggling activity generators) that create *voltage oscillation* and droop transients on the accelerator's supply island.

Parameterization. Each injection episode is parameterized by onset t_0, duration Δt, and amplitude A (frequency offset or voltage delta). Oscillatory cases additionally specify a modulation period T_{mod} and duty. We cross–vary $\{t_0, \Delta t, A, T_{\text{mod}}\}$ to span short/long and weak/strong disturbances across four families: overclock, undervolt, clock oscillation, and voltage oscillation.

5.1 Experimental Results and Analysis

We evaluate Swift-Healer on a chiplet prototype running the edge-detector accelerator and the full firmware stack (Monitor, Controller, Healing). Experiments cover four main remote-glitch classes, including overclocking (a step increase above the validated clock), clock oscillation (periodically modulates the clock about nominal), undervolting (steps the rail below its nominal DVFS point), and voltage oscillation (periodic ripple on the rail) under varied onset times, durations, and intensities. We report average accelerator latency, self-healing latency, perception accuracy, and system power for both normal execution and execution with healer overhead.

Predictor/Detector Performance: The GRU predictor operates on 8–sample windows. Each window is a control–aligned multivariate slice. It is trained on 40,000 windows and tested on 10,000 with episode–wise splits and a held–out stress set covering benign runs and four main glitch families generated via software–exposed DVFS perturbations and parameterized injections. The model forecasts two control cycles (real–time loop iterations) ahead and attains F1= 0.87 (precision = 0.83, recall = 0.92). At runtime, the proactive predictor issues timely *predict* alarms for ∼ 92% of glitches (blue marker), consistent with its recall. Residual misses are confirmed by the reactive detector (black marker). The predictor is valuable because it shifts many events from post-failure repair to preemptive healing, reducing the corruption window.

Baseline and attack onset: Normally, the Edge detector runs at F1 ≈ 0.96 with 0.03 ms latency and 1.95 W power. After the attack connects to the DVFS path, telemetry shows oscillation in latency/power and a drop in accuracy; the predictor requires an

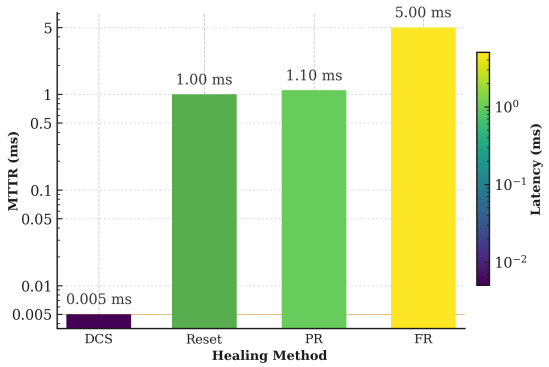


Figure 4: Healing latency (application downtime, MTTR)

initial warm-up to accumulate 8 samples; thereafter, it issues a new forecast every sample (≈0.03 ms) and emits a *predict* alarm when risk exceeds threshold. The timelines shown reflect post-warm-up behavior averaged over executions. If prediction is missed, the detector raises a *detect* alarm ∼0.03 ms after the glitch manifests.

Timeline Analysis and Overheads: Figure 3 arranges the three self-healing scenarios *by columns*: the **left** column shows *Predict+Heal*, the **middle** column shows *Detect+Heal (Moderate Stress)*, and the **right** column shows *Detect+Heal (High Stress)*.

Scenario 1: Predict + Heal. When the predictor raises a high-confidence alarm *before* the disturbance begins, the controller immediately initiates healing. For *clock-side* attacks, it applies DCS to stabilize timing. Downtime is minimal, yielding the shortest Mean Time to Repair (MTTR) ($t_{\mathrm{DCS}} \approx 0.005$ ms) with a small power bump to ∼1.98 W and accuracy near baseline. For *voltage-side* attacks, prediction indicates risk of state corruption or persistent rail instability. The controller therefore isolates the accelerator and invokes PR when the fault is localized ($t_{\mathrm{PR}} \approx 1.1$ ms, peak power ∼2.30 W), or a scoped Reset when PR is unnecessary or not applicable ($t_{\mathrm{RST}} \approx 1$ ms, ∼2.09 W). By acting preemptively, the system avoids in-flight state corruption, keeps availability within control deadlines, and preserves near-baseline accuracy and power.

Scenario 2: Detect + Heal (Moderate Stress). Wherever the predictor does not yield sufficient confidence to raise an early alarm, the glitch manifests, and the ***reactive detector*** confirms a fault classified as *Moderate Stress* (a short-lived, localized disturbance), which is evidenced by the edge-detector F1 dipping to ≈ 0.70. For *clock-side* disturbances, the controller first applies DCS to recover timing margin. If the anomalies persist, it escalates to PR to refresh the affected region and requalify the clock domain. For *voltage-side* disturbances, correlated rail and canary shifts indicate possible state corruption, so the controller selects PR to refresh the accelerator image. When corruption is limited to a transient state, a scoped *Reset* suffices. PR restores service with $t_{\mathrm{PR}} \approx 1.1$ ms, while DCS and Reset provide lower-latency repairs when structural refresh is unnecessary. Here, the policy minimizes downtime by selecting the lightest effective action, containing localized faults without over-repair and returning accuracy to spec within the control budget.

Scenario 3: Detect + Heal (High Stress). For sustained or coupled glitches, the event is classified as *High Stress* (a sustained or cross-domain disturbance), where the edge-detector F1 drops to ≈ 0.50 with pronounced latency/power oscillation; the Analyser flags high severity and the policy selects *heavy* healing actions

Table 1: FPGA resource comparison between the baseline Sobel accelerator and the full Swift-Healer design on Z-7010.

Resource Type	Available	Baseline	Swift-Healer
LUT	17600	13024	14030
LUTRAM	6000	930	1067
FF	35200	15932	17000
DSP	80	0	0
BRAM	60	9	14
BUFGCTRL	32	0	7

only. If corruption is *localized* to accelerator region, controller performs PR. If corruption is *widespread* (multiple regions or persistent rail/clock instability), the controller escalates to FR, yielding the longest MTTR, $t_{\mathrm{FR}} \approx 5$ ms, and a peak power ∼2.44 W. Under high severity, only structural repairs (PR/FR) restore correctness.

Overhead analysis: Fig. 4 summarizes the PS-side runtime overhead of Swift-Healer during recovery, reported as healing latency or application downtime (MTTR) after Monitor/Controller initiates DCS, Reset, PR, or FR. DCS is fastest because it performs an in-fabric clock step with only a register write and short PLL settle, requiring no data movement. Reset incurs ∼1 ms to quiesce interfaces, drain DMA, and reinitialize the accelerator. PR adds bitstream transfer and decoupler/isolation handshakes, yielding ∼1.1 ms while keeping the rest of the system live. FR is slowest because it reconfigures the full device and replays portions of the boot/config chain. The resulting timing hierarchy aligns with ANS constraints: lightweight actions preserve loop slack under moderate stress, while heavier actions are reserved for high-stress corruption where brief availability pauses are safer than continued operation under corruption.

FPGA Resource Footprint: We report the PL-side hardware area overhead of Swift-Healer on the Z-7010. Table 1 summarizes the resources of the baseline accelerator and the additional logic required by Swift-Healer, including the clock wizard, DMA control logic, and PR/FR decoupling infrastructure. To avoid conflating hardware and software overheads, the table reports only PL resources; lightweight PS components are not included in these counts. The reported resources cover the main logic and memory structures: LUT and FF for control and datapaths, LUTRAM for small memories, DSP for fixed-point arithmetic, BRAM for on-chip storage, and BUFGCTRL for global clock management across the PS, accelerator, and healing partition. In terms of power, the chiplet-oriented partition (PS, accelerator chiplet, and a dedicated healing chiplet) adds a small steady clocking/control footprint. Its primary benefit is containment and frequency diversity: faults do not propagate across chiplets, and the healing partition can operate at a more resilient clock during recovery. After a PR event, the *resilient* accelerator image (with parity checks and temporal hardening) raises steady-state power relative to baseline—an availability–power trade-off that safeguards perception under disturbances (see Sec. 5).

6 Related Work

Prior defenses against fault injection in embedded/automotive systems mainly rely on added sensors, monitors, microarchitectural checkers, resets, or redundancy, each incurring latency, energy, area, or coverage trade-offs [18, 28, 29]. Surveys further show that remote over/undervolting and clock manipulation through software-visible

power-management interfaces are increasing, while most countermeasures still emphasize detection rather than recovery [11].

Hardware monitors such as ring-oscillator sensors, time-to-digital voltage/clock sensors, path-delay canaries, and power-integrity sentinels can detect glitches, but they add area/power overhead and typically depend on reset-centric remediation [12, 25]. Microarchitectural techniques such as *MAFIA* protect CPU control flows but are processor-centric and do not cover heterogeneous accelerator fabrics used in ANS perception [2]. Similarly, *FTC* integrates FIA detection with prevention policies, but remains monitor-driven and can constrain runtime performance–power flexibility [21]. In contrast, Swift-Healer combines proactive prediction, fallback reactive confirmation, and ACT-driven corrective healing, enabling runtime recovery rather than post-manifestation detection alone.

7 Conclusion

We introduced *Swift-Healer*, a firmware-reconfigurable self-healing architecture for ANS perception under remote clock/voltage glitches. Swift-Healer combines proactive prediction, fallback reactive detection, and ACT-driven healing to restore operation in an automatic loop. Its chiplet-oriented partition provides frequency diversity and isolation for localized recovery, and the Zynq−7000 implementation shows that earlier intervention reduces corruption exposure compared with detection-only recovery.

Acknowledgments

This work is supported by the Office of Naval Research under Grant N00014-24-1-2046. The opinions, recommendations, findings, and conclusions in this article do not necessarily reflect the views or policies of NIST or the United States Government.

References

[1] Morteza Adelkhani, Ali Suvizi, Farzaneh Arzaghi, Sara Zamani, Mohammad Salehi, and Muhammad Shafique. 2025. LPCR: Information-Theory-Based Low-Power Code Reordering for Serial Links in Network-on-Chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025), 4155–4166.

[2] 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.

[3] Dario Ćorić, Ivan Kaštelan, and Nebojša Pjevalica. 2021. Implementation of different image edge detection algorithms on a real embedded ADAS platform. In *2021 Zooming Innovation in Consumer Technologies Conference*. IEEE, 193–197.

[4] Cyient. [n. d.]. How OTA Accelerates Software-Defined Vehicle Transformation. https://www.cyient.com/hubfs/Whitepaper/ota/Whitepaper_OTA.pdf

[5] Shijin Duan, Wenhao Wang, Yukui Luo, and Xiaolin Xu. 2021. A survey of recent attacks and mitigation on FPGA systems. In *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 284–289.

[6] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, 385–396.

[7] Mahyar Ghazanfari, Iman Sharifi, Peng Wei, Abenezer Taye, Bryan Ward, Xenofon Koutsoukos, Gautam Biswas, Mahsa Ghasemi, Vijay Gupta, Hyeong Tae Kim, et al. 2026. A Survey of Security Challenges and Solutions for Advanced Air Mobility and eVTOL Aircraft. In *AIAA SCITECH 2026 Forum*. 2891.

[8] Kailash Gogineni, Ali Suvizi, and Guru Venkataramani. 2025. Llms on a budget: System-level approaches to power-efficient and scalable fine-tuning. *IEEE Open Journal of the Computer Society* (2025).

[9] International Organization for Standardization. 2011. Road vehicles—Functional safety (ISO 26262). ISO Standard ISO 26262:2011.

[10] Taminul Islam, Md Alif Sheakh, Anjuman Naher Jui, Omar Sharif, and Md Zobaer Hasan. 2023. A review of cyber attacks on sensors and perception systems in autonomous vehicle. *Journal of Economy and Technology* 1 (2023), 242–258.

[11] Kyounggon Kim, Jun Seok Kim, Seonghoon Jeong, Jo-Hee Park, and Huy Kang Kim. 2021. Cybersecurity for autonomous vehicles: Review of attacks and defense. *Computers & Security* 103 (2021), 102150.

[12] Gwenn Le Gonidec, Guillaume Bouffard, Jean-Christophe Prevotet, and Maria Méndez Real. 2025. Do Not Trust Power Management: A Survey on Internal Energy-based Attacks Circumventing Trusted Execution Environments Security Properties. *ACM Transactions on Embedded Computing Systems* (2025).

[13] Beibei Li, Wei Hu, Lemei Da, Yibing Wu, Xinxin Wang, Yiwei Li, and Chaoxuan Yuan. 2024. Over-the-air upgrading for enhancing security of intelligent connected vehicles: a survey. *Artificial Intelligence Review* 57, 11 (2024), 314.

[14] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. 2018. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 751–766.

[15] Wenye Liu, Chip-Hong Chang, Fan Zhang, and Xiaoxuan Lou. 2020. Imperceptible misclassification attack on deep learning accelerator by glitch injection. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[16] Shahid Mahmood, Hoang Nga Nguyen, and Siraj Ahmed Shaikh. 2022. Systematic threat assessment and security testing of automotive over-the-air (OTA) updates. *Vehicular Communications* 35 (2022), 100468.

[17] Dina G Mahmoud, Vincent Lenders, and Mirjana Stojilović. 2022. Electrical-level attacks on CPUs, FPGAs, and GPUs: Survey and implications in the heterogeneous era. *ACM Computing Surveys (CSUR)* 55, 3 (2022), 1–40.

[18] Amit Mazumder Shuvo, Tao Zhang, Farimah Farahmandi, and Mark Tehranipoor. 2023. A Comprehensive Survey on Non-Invasive Fault Injection Attacks. Cryptology ePrint Archive, Paper 2023/1769. https://eprint.iacr.org/2023/1769

[19] Shayan Moini, Aleksa Deric, Xiang Li, George Provelengios, and Daniel Holcomb. 2022. Voltage sensor implementations for remote power attacks on FPGAs. *ACM Transactions on Reconfigurable Technology and Systems* 16, 1 (2022), 1–21.

[20] Raymond Muller, Ruoyu Song, Chenyi Wang, Yuxia Zhan, Jean-Phillipe Monteuuis, Yanmao Man, Ming Li, Ryan Gerdes, Jonathan Petit, and Z. Berkay Celik. 2025. Investigating Physical Latency Attacks Against Camera-Based Perception. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 4588–4605.

[21] Md Rafid Muttaki, Md Habibur Rahman, Akshay Kulkarni, Mark Tehranipoor, and Farimah Farahmandi. 2024. Ftc: A universal framework for fault-injection attack detection and prevention. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 32, 7 (2024), 1311–1324.

[22] Shakiba Rezaie, Ehsan Mousavi Khaneghah, Mario Francesco Pavone, Mahdi Hajirajabi, Reza Safari, Ali Suvizi, and Amirhossein Reyhani Showkatabad. 2025. OutExaRD: Enhancing Resource Discovery for Support Dynamic and Interactive Events on Out-of-Distributed Exascale Systems. *International Journal of Networked and Distributed Computing* 13, 1 (2025), 18.

[23] Bodo Selmke, Florian Hauschild, and Johannes Obermaier. 2019. Peak Clock: Fault Injection into PLL-Based Systems via Clock Manipulation. In *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop (ASHES'19)*. Association for Computing Machinery, New York, NY, USA, 85–94.

[24] Wissam Sleiman, Mohaiminul Haque, Mohammad Saiful Amin, Pedram Beigi, Michel Khoueiry, Seungmo Kang, and Samer Hamdar. 2026. Diffusion Process-Based Model for Network Trajectory Propagation: Formulation, Data Mining, and Cross-Comparative Analysis. *IEEE Transactions on Intelligent Transportation Systems* (2026).

[25] Ali Suvizi, Ruhollah Ahmadi, Morteza Saheb Zamani, and Mohammad Reza Meybodi. 2025. A parallel computational approach for energy-efficient hydraulic analysis of water distribution networks using learning automata. *Sustainable Computing: Informatics and Systems* 46 (2025), 101135.

[26] Ali Suvizi, Azim Farghadan, and Morteza Saheb Zamani. 2023. A parallel computing architecture based on cellular automata for hydraulic analysis of water distribution networks. *J. Parallel and Distrib. Comput.* 178 (2023), 11–28.

[27] Ali Suvizi, Suresh Subramaniam, Tian Lan, and Guru Venkataramani. 2024. Exploring in-memory accelerators and fpgas for latency-sensitive dnn inference on edge servers. In *2024 IEEE Cloud Summit*. IEEE, 1–6.

[28] Ali Suvizi and Guru Venkataramani. 2025. Auto-Healer: Self-Healing Hardware for Perception Stage Faults in Autonomous Driving Systems. In *Proceedings of the 39th ACM International Conference on Supercomputing*. ACM, 1064–1078.

[29] Mahsa Tahghigh and Hassan Salmani. 2024. Detecting Hardware Trojans in Manufactured Chips without Reference: A GMM-Based Approach. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 1–7.

[30] Tesla. [n. d.]. Software Updates. www.tesla.com/support/software-updates.

[31] Guru Prasadh Venkataramani and Sanjay Rekhi. 2026. Firmware-Based Monitoring for Bus-Based Computer Systems. (2026).

[32] Chenyi Wang, Raymond Muller, Ruoyu Song, Jean-Philippe Monteuuis, Jonathan Petit, Yanmao Man, Ryan Gerdes, Z Berkay Celik, and Ming Li. 2025. From Threat to Trust: Exploiting Attention Mechanisms for Attacks and Defenses in Cooperative Perception. In *34th USENIX Security Symposium*. 7387–7406.

[33] Junge Xu, Bohan Xuan, Anlin Liu, Mo Sun, Fan Zhang, Zeke Wang, and Kui Ren. 2022. Terminator on SkyNet: a practical DVFS attack on DNN hardware IP for UAV object detection. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. Association for Computing Machinery, New York, NY, USA, 685–690.

[34] Junge Xu, Fan Zhang, Wenguang Jin, Kun Yang, Zeke Wang, Weixiong Jiang, and Yajun Ha. 2025. A Deep Investigation on Stealthy DVFS Fault Injection Attacks at DNN Hardware Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44, 1 (2025), 39–51.