Fault Injection Attacks and Countermeasures on TinyML Algorithms

Anthony Etim*
Yale University
anthony.etim@yale.edu

Srilalith Nampally*
Virginia Tech
nsrilalith@vt.edu

Aubtin Rasouli
Virginia Tech
aubtinr@vt.edu

Dustin Mazza
Virginia Tech
djmazza@vt.edu

Krishna Chilakapati
Virginia Tech
krishnapc20@vt.edu

Tinghung Chiu
Virginia Tech
kennychiu0818@vt.edu

Ferhat Erata
Yale University
ferhat.erata@yale.edu

Leyla Nazhandali
Virginia Tech
leyla@vt.edu

Wenjie Xiong
Virginia Tech
wenjiex@vt.edu

Jakub Szefer
Northwestern University
jakub.szefer
@northwestern.edu

*Anthony Etim and Srilalith Nampally contributed equally to this work.

***Abstract*—Tiny Machine Learning (TinyML) algorithms, designed to operate on constrained devices such as those found in Internet of Things (IoT) systems, are vulnerable to adversarial threats, including fault injection attacks. These attacks exploit physical means to induce errors in computation, compromising the reliability of the device and the TinyML models running on it. This work investigates the operation of TinyML models under fault injection attacks. Through systematic experimentation with voltage glitching and EM fault injection attacks on microcontrollers, this work identifies configurations that adversaries can exploit to induce faults without triggering a system reset, thus focusing on finding the more stealthy attacks. This study analyzes four types of TinyML models, and demonstrates that all four evaluated TinyML models will generate inference outputs with reduced accuracy under the two types of fault injection attacks. Further, in some instances, this work shows that it may be feasible for the attackers to use the faults to cause inference operations to return a predictable output, not just random incorrect inference results. This highlights the need for more robust fault injection protection mechanisms in TinyML implementations. In order to provide one such protection, this work demonstrates the use of Randomized Self-Reduction (RSR) schemes and majority voting for intermediate values as a means to protect the TinyML models.**

***Index Terms*—Fault Injection Attacks and Countermeasures, Machine Learning, TinyML**

I. Introduction

Tiny Machine Learning (TinyML) runs inference directly on resource-constrained embedded devices [1], improving latency, bandwidth, and privacy. Unlike when using server devices in the cloud, TinyML devices are often physically accessible in the field [2], making them attractive targets for fault injection attacks [3], which are deliberate transient perturbations (e.g., voltage glitches or electromagnetic pulses) that corrupt computations or control flow. Most prior fault-injection research targets cryptographic implementations, e.g., [2], [4], [5], [6], [7], while analogous risks for TinyML workloads remain poorly understood.

In this work, we evaluate voltage and electromagnetic fault injection attacks on four representative TinyML models running on microcontrollers, targeting core operations such as matrix-vector multiplication and non-linear activation functions. We show that small, well-timed faults can cause perturbations in intermediate data, which then propagate to the output and result in large errors in the TinyML output, leading to misclassifications. Further, in some settings, controllable misclassifications can be achieved, not just random ones.

Our study focuses on two target boards and two types of fault injection attacks. First, we apply voltage glitching to a Riscure Pinata board with a STM32F417IG microcontroller target and study when glitches induce effective, non-reset faults across the four TinyML models. Next, we demonstrate voltage-glitch attacks using a ChipWhisperer-Husky on a SAM4S microcontroller target enabling a cross-platform comparison of fault behaviors and success rates. Further, we perform electromagnetic (EM) fault injection on the Riscure Pinata board and study how faults are sensitive to probe placement and timing.

Across these experiments, we find that carefully tuned voltage glitches can steer TinyML models toward fixed-label outputs by inducing bit flips in intermediate values. EM fault injection can also produce the same fixed-label misprediction, but only within narrow fault injection-time windows. A larger share of EM injection attempts also result in device resets, which makes EM fault injection less stealthy than voltage glitching. Overall, voltage glitching yields more frequent and structured corruptions in intermediate values, is less sensitive to the physical setup than EM fault injection, and does not have problems due to EM probe positioning that EM fault injection has.

To mitigate these attacks, we propose a lightweight countermeasure based on Randomized Self-Reduction (RSR) schemes. RSRs operate by: randomizing function inputs, evaluating multiple equivalent function instances, and then recombining results via majority voting with median-based filtering while respecting the limited compute and memory budgets of microcontrollers.

A. Contributions

The contributions of this work are:

- The first systematic evaluation of fault injection on four TinyML algorithms (FastGRNN, TinyCNN, Logistic Regression, and Wake Word) running on microcontrollers.
- An empirical characterization of voltage and EM fault injection on TinyML workloads using commercially available equipment, including Riscure Pinata and ChipWhisperer-Husky platforms.
- An analysis of how fault parameters (fault trigger timing, fault voltage or EM strength, and fault duration) affect prediction accuracy and fault patterns in the TinyML models.
- A detailed study of how injected faults perturb intermediate values and how these errors propagate to final outputs, including configurations that yield predictable, attacker-controlled inference outcomes rather than random mispredictions.
- A novel application of Randomized Self-Reductions (RSRs) to harden TinyML workloads by designing and evaluating RSR-based protections for core ML operations such as sigmoid, matrix-vector and matrix-matrix multiplication, and convolution.

B. Code Availability

The code and evaluation artifacts used in this work are available at https://github.com/caslab-code/ml-fault-attacks-on-tinyml.

II. Background and Related Work

This section provides information on Tiny Machine Learning (TinyML) algorithms as well as Fault Injection attacks on machine learning algorithms.

A. TinyML Algorithms

TinyML enables deployment of machine-learning models on resource-constrained devices such as microcontrollers [1]. To meet limited memory and compute budgets, TinyML relies on techniques such as quantization, pruning, and compact architectures for efficient edge inference.

Quantization reduces parameter and compute precision (e.g., `float` to `int8`), lowering memory footprint and computational cost for embedded targets. While quantization can slightly reduce accuracy, post-training quantization and quantization-aware training often preserve performance. In our experiments, we use TensorFlow Lite [8] post-training quantization and CMSIS-NN [9] kernels to meet the strict on-device memory and storage constraints while maintaining good accuracy.

B. Fault Injection Attacks

Fault injection attacks intentionally induce hardware-level errors to manipulate execution or compromise integrity [3]. Common techniques include electromagnetic interference [10], voltage glitching [4], clock glitching [11], and laser-induced faults [12]. Such faults can enable security bypasses or result in the disruption of normal operation.

Of these, voltage glitching and EM fault injection are the most practical for field attacks on embedded devices, as both induce transient faults without requiring chip decapsulation [13]. While extensively studied against cryptographic implementations [5], their effects on ML inference remain less explored.

C. Fault Injection Attacks on ML

Fault injection attacks against machine learning induce transient errors during training or inference to corrupt computations and cause mispredictions, impacting safety-critical applications. Prior work spans Rowhammer-induced bit flips in weights [14], [15], [16], laser injections on activation functions [12], DVFS-induced GPU faults [17], clock glitches in convolutional layers [18], and bit faults in quantized NN accelerators [19]. Some attacks (e.g., DeepHammer [15] and DeepVenom [14]) use carefully chosen bit flips to cause large accuracy drops or inject trojans, while others (e.g., Terminal Brain Damage [20]) study the effects of random corruptions. Additionally, prior work on fault attacks against DNN inference [21], [22] further motivates the investigation of TinyML-specific vulnerabilities. These studies establish the feasibility of fault attacks on ML but largely target bigger platforms and full-scale models rather than highly resource-constrained microcontrollers, which is what we target.

On the embedded side, TinyML libraries enable neural network inference under strict memory or energy budgets [23], which changes both the attack surface and feasible defenses. Recent work shows that embedded deployments of ML can leak or be manipulated via fault and side-channel mechanisms, including parameter or input recovery and practical EM or power-based setups [24], [25], [26], [20], [17], [27]. However, many of these evaluations study a single platform and a single model, and they do not cover systematic fault characterization across multiple microcontroller setups.

D. Randomized Self-Reduction and Self-Correctness

Randomized Self-Reduction (RSR) schemes rely on randomized self-reducibility, which captures functions for which computing $f(x)$ can be reduced to evaluating f on a few randomly related inputs plus a cheap combining step. For example, for the linear function $f_c(x) = c \cdot x$, choosing a random offset r gives:

$$f_c(x) = c(x - r) + cr = f_c(x - r) + f_c(r),$$

so a single evaluation on x is replaced by two evaluations on random points $x - r$ and r and one addition.

Definition 1 (Randomized Self-Reducibility [28], [29])**.** *A function $f : \mathcal{D} \to \mathcal{R}$ is c-randomly self-reducible if there exist random variables $a_1, \ldots, a_k$ (with $k \leq c$) and a combining function F computable in time $O(T_f)$ such that for every $x \in \mathcal{D}$,*

$$F\big(x,\, a_1, \ldots, a_k;\; f(a_1), \ldots, f(a_k)\big) \;=\; f(x),$$

where the a_i are drawn (possibly dependently) from a distribution independent of f.

Blum et al. [28] show that any RSR function admits a *self-corrector*: given a black-box program P that computes f correctly on all but an ϵ-fraction of inputs, one can query P on a small number of random points and combine the answers so that the new procedure outputs the correct value $f(x)$ on *every* input with high probability.

Erata et al. [30], [31] applied these ideas to protect arithmetic kernels and to learn RSRs for nonlinear real-valued functions. Building on this, we implement RSR schemes for TinyML primitives: sigmoid, matrix-vector and matrix-matrix multiplication, and convolution (via a matrix-vector formulation), as described in Section VI.

III. Threat Model

In this work, we consider an active adversary who deliberately injects transient faults into a TinyML system to compromise its inference outcomes. For example, an outdoor motion-detection IoT node that runs a TinyML classifier to decide whether to trigger an alarm, where injected faults can suppress true alerts or induce false alarms. We focus on voltage glitching attacks and electromagnetic (EM) fault injection attacks, where the adversary perturbs the supply voltage or injects electromagnetic interference into the device, respectively. Consequently, we assume the adversary has physical access to the target device, which is realistic for many TinyML deployments (embedded devices, robots, IoT nodes, and other small systems located outside physically secure facilities).

On the software side, we assume the adversary knows the TinyML algorithm and its implementation (e.g., firmware or binary) and can synchronize fault injection with model execution. In particular, the adversary can choose the glitch start time, duration, and target voltage (for voltage attacks) or EM pulse parameters (for EM attacks), when a new input is processed by the device.

Our threat model is consistent with prior fault-injection work on ML [15], in that the adversary perturbs computations during inference rather than retraining or modifying the model. Unlike Rowhammer-based attacks that require executing malicious code on the device to flip bits in DRAM, we consider a more accessible vector: external physical fault injection via voltage or EM glitches. A representative scenario is an unattended edge node that receives and classifies images from a secure drone: the attacker cannot alter the encrypted input stream but has physical access to the node, enabling them to inject faults and force a misclassification (e.g., classifying a "threat" as "benign").

IV. Attack Methodology and Implementation

This section describes how we *induce*, *measure*, and *attribute* faults in TinyML inference across three different setups: (i) **Pinata board voltage glitching**, (ii) **ChipWhisperer-Husky voltage glitching**, and (iii) **Pinata board EM fault injection**. Across all setups, the firmware exposes repeatable trigger points inside each model, and each trial is classified as *Correct*, *Misprediction*, or *Reset/Hang*.

TABLE I: FastGRNN Model Architecture

No.	Layer	Input Shape	Output Shape	Fused Activation	ROI
-	Input	(1, 16, 16)	-	n/a	
1	Recurrent Layer	(1, 16, 16)	(1, 32)	Sigmoid and Tanh	✓
2	Fully Connected	(1, 32)	(1, 10)	n/a	
-	Output	-	(1, 10)	n/a	

A. Platforms and Hardware Setup

1) Pinata Board Target (STM32F417IG): Figure 1(a-c) shows the Riscure Pinata board used for both voltage glitching and EM fault injection. The board hosts an STM32F417IG (ARM Cortex-M4F) microcontroller target, representative of Microcontroller Units (MCUs) used for embedded TinyML deployments.

a) Voltage Glitching Setup: For voltage fault injection, we use a Spider VCC glitcher with a voltage amplifier to inject controlled VDD perturbations during inference. A PicoScope monitors trigger and supply rails for characterization, while a controller PC communicates with the target over UART to send inputs, collect predictions, and program glitch parameters via Python.

b) EM Fault Injection Setup: For EM faults, an EM probe is positioned over the MCU package and driven by an EM fault injection pulse generator. The controller triggers a pulse with a programmable delay relative to a firmware-defined trigger pin, while an oscilloscope monitors the trigger and probe coil current for alignment. A relay-based reset path automates recovery from faults, and a buck converter regulates the voltage supply to the board.

2) ChipWhisperer-Husky Target (CW312T-SAM4S): Figure 1d shows our ChipWhisperer-Husky setup for glitching and capture of output from a CW313 baseboard with a CW312T-SAM4S microcontroller target (ATSAM4S2A, Cortex-M4).

a) Voltage Glitching Setup: The CW313 powers the target, provides UART connectivity through I/O pins TIO1 and TIO2, and routes a trigger signal from GPIO4 or nRSTOUT pins back to the ChipWhisperer-Husky to align glitches with regions of interest. During the experiments, we sweep ChipWhisperer-Husky glitch parameters (e.g., width, offset, external offset, repetition) and classify outcomes using the same trial categories as for the Pinata board.

B. Target TinyML Models

We target four TinyML workloads spanning tabular, audio, and vision inference: FastGRNN, a TinyCNN, Logistic Regression and Wake Word detection. Together they cover `float` and quantized `int8` inference, core compute kernels (matrix-vector and matrix-matrix multiplication, dense layers, convolution, pooling), and different control-flow patterns.

FastGRNN: FastGRNN is a TinyML recurrent model that reduces prediction cost while maintaining accuracy through a compact gating structure [32]. We use the C implementation

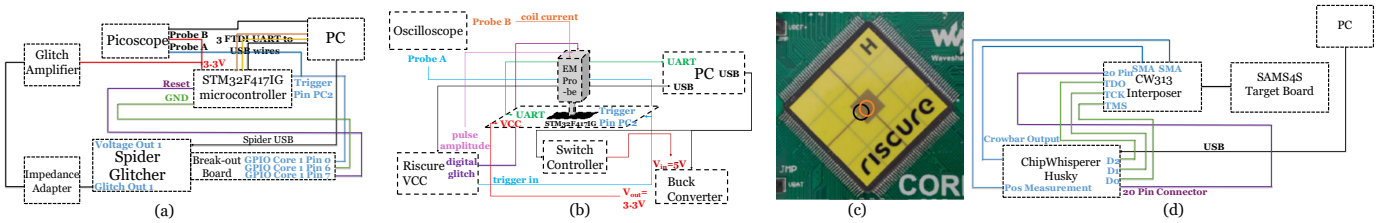


Fig. 1: Hardware setup for (a) Pinata board voltage glitching, (b) Pinata board EM fault injection, (c) probe placements used for EM fault injection on the MCU package (Probe Position 1 - Black, Probe Position 2 - Orange) and (d) ChipWhisperer-Husky SAM4S target voltage glitching experiments.

TABLE II: TinyCNN Model Architecture

No.	Layer	Input Shape	Output Shape	Fused Activation	ROI
-	Input	(1, 8, 8, 1)	-	n/a	
1	Convolution	(1, 8, 8, 1)	(1, 6, 6, 8)	n/a	✓
2	Maxpool	(1, 6, 6, 8)	(1, 3, 3, 8)	n/a	
-	Reshape	(1, 3, 3, 8)	(1, 72)	n/a	
3	Fully Connected	(1, 72)	(1, 10)	n/a	
-	Output	-	(1, 10)	n/a	

TABLE III: Logistic Regression Model Architecture

No.	Layer	Input Shape	Output Shape	Fused Activation	ROI
-	Input	(1, 11)	-	n/a	
1	Fully Connected	(1, 11)	(1, 1)	Sigmoid	✓
-	Output	-	(1, 1)	n/a	

from Microsoft EdgeML [33]. The input is processed as a sequence of 16 time steps with 16 features per step.

TinyCNN: TinyCNN is a compact CNN for MNIST digit classification [34], [35], quantized to `int8` using TensorFlow Lite [36] and deployed with CMSIS-NN [37].

Logistic Regression: Logistic Regression is a standard linear classifier for binary outcomes, valued for its simplicity and interpretability [38]. We implement a custom lightweight C inference path.

Wake Word: Wake Word detection (also called keyword spotting) is a common always-on workload in voice interfaces. We follow the MFCC-based pipeline and network described in [39], quantize to `int8` using TensorFlow Lite [36], and deploy using CMSIS-NN [37].

C. Firmware Deployment, Triggering, and Trial Classification

All models are deployed as on-device C inference binaries. The controller selects a model over UART, sends one input sample, and reads back the predicted label (and, when applicable, intermediate values for the analysis in Section V-D). The code of each model is modified to trigger the voltage or EM glitches around a *Region of Interest* (ROI), typically the first compute-intensive layer (e.g., first convolution or first fully connected layer).

Each trial is classified as: (i) *Correct* if the returned label matches ground truth, (ii) *Misprediction* if the returned label is valid but incorrect, and (iii) *Reset/Hang* if the device fails to respond within a timeout or reboots mid-trial. We treat reset and hang identically (both are non-responses within the timeout) and categorize them as a single *Reset/Hang* outcome. We report the *reset rate* (resets + hangs / total trials), and standard end-to-end accuracy computed over valid (non-reset) return values.

TABLE IV: Wake Word Detection Model Architecture

No.	Layer	Input Shape	Output Shape	Fused Activation	ROI
-	Input	(1, 40)	-	n/a	
1	Fully Connected	(1, 40)	(1, 256)	n/a	✓
2	Fully Connected	(1, 256)	(1, 256)	n/a	
3	Fully Connected	(1, 256)	(1, 2)	n/a	
4	Softmax	(1, 2)	(1, 2)	n/a	
-	Output	-	(1, 2)	n/a	

D. Fault Injection Parameters

We describe the swept parameters for each platform and fault injection method.

a) Pinata Board Voltage Glitching: For Pinata board voltage glitching, the swept parameters are:

- **Target voltage** (`voltage`): the minimum VDD reached during the glitch (3.3 V to 0 V where 0 V denotes a full drop).
- **Glitch duration** (`duration`): how long the target voltage drop is applied (in nanoseconds).
- **Injection time** (`inject_time`): the delay between the glitch trigger and the actual glitch injection (in nanoseconds).

b) Pinata Board EM Fault Injection: For Pinata board EM fault injection, the swept parameters are:

- **Pulse amplitude** (`amplitude`): effective EM pulse amplitude, swept from -7.0 V to 3.5 V in 0.5 V increments.
- **Injection time** (`inject_time`): trigger-relative delay between the fault trigger and the EM pulse launch, swept over a coarse window of $[0, 100]$ ns in steps of 5.
- **Probe position** (`probe_pos`): physical placement of the EM probe over the MCU package; we evaluate the probe positions shown in Figure 1c.

c) ChipWhisperer-Husky Voltage Glitching: For the ChipWhisperer-Husky, the swept parameters are:

- **Glitch Width** (`width`): width of a single glitch pulse, expressed in phase-shift steps, swept over $[0, 4000]$.
- **Glitch Offset** (`offset`): sub-cycle phase offset relative to the victim clock edge, expressed in phase-shift steps, swept over $[0, 4000]$.
- **External Offset** (`ext_offset`): number of victim clock cycles from trigger to the first pulse, swept over $[0, E_{max}]$.

- **Repeat** (`repeat`): number of pulses per trigger (voltage glitching), swept over number of repeats in range $[1, 5]$.

E. Parameter Search and Injection-Time Sweeps

Fault injection effectiveness depends jointly on (i) *fault strength* (e.g., voltage drop or EM pulse amplitude), (ii) *fault duration*, and (iii) *fault timing* relative to a vulnerable computation. To make results comparable across models and across platforms, we adopt a consistent two-stage procedure.

a) Stage 1: Fault Injection Parameter Sweep (strength × duration): To identify effective non-reset-inducing fault settings, we first perform a coarse search over the platform-specific fault parameters. For each tested configuration, we execute repeated trials and record the frequencies of *Correct*, *Misprediction*, and *Reset/Hang*. We then use the observed outcomes to identify regions that are mostly correct, regions where mispredictions emerge without dominating resets, and regions dominated by reset/hang.

For Pinata board voltage glitching, this coarse search is performed over target voltage and glitch duration. For Pinata board EM fault injection, we follow the same selection logic but sweep probe-drive amplitude and trigger-relative injection offset at each probe position (Figure 1c), repeating each setting 50 times, and then retain one or two settings that maximize mispredictions while avoiding reset-dominated regimes.

For ChipWhisperer-Husky voltage glitching, we use Optuna [40] to search the four-parameter glitch configuration space. Each Optuna trial samples a configuration $\{\texttt{width}, \texttt{repeat}, \texttt{offset}, \texttt{ext\_offset}\}$, executes a batch of inferences, and receives a scalar objective score. In our experiments, we simply maximize the number of observed faults (mispredictions), while discarding configurations that lead to pathological reset-dominated behavior.

After the Stage 1 search, we select one or two operating points that maximize mispredictions subject to a reset-rate cap. These operating points are then used for further analysis.

b) Stage 2: Injection Timing Sweep (Normalized Injection Time): After Stage 1, we select one or two of the most effective `voltage` and `duration` injection settings (for Pinata board voltage glitching), one or two most effective EM `amplitude` and `inject_time` pairings (for Pinata board EM fault injection), or one or more best-found glitch configurations (for ChipWhisperer-Husky voltage glitching). Then, we sweep the `inject_time` (for Pinata board glitching) or `ext_offset` (for ChipWhisperer-Husky voltage glitching) within the Region Of Interest (ROI) in a finer-grain manner. Across all models, we target the earliest compute-heavy layer as ROI (e.g., the first convolution or first fully connected layer), which also provides a stable trigger boundary. We define a normalized injection time $t \in [0, 1]$ with $t = 0$ at ROI entry (fault trigger rising edge) and $t = 1$ at ROI exit (fault trigger falling edge). This normalization makes timing sweeps comparable even when ROI runtimes differ across models. The sweep aims to find the most reliable non-reset-inducing faults.

V. Experimental Results

This section presents the evaluation results and our findings on effects of fault injection (Voltage and EM) on the various TinyML algorithms.

A. Pinata Board Voltage Glitching Results

We first evaluate voltage glitching on the Pinata board by (i) sweeping glitch `voltage` and `duration` to identify regions with large amounts of faults and few resets/hangs, and then (ii) sweeping `inject_time` t_i within the ROI to characterize timing sensitivity (Section IV-E). Unless stated otherwise, we report (a) *accuracy* computed over valid (non-reset) returns and (b) *reset rate* as the number of resets/hangs observed for each configuration over the evaluated input set.

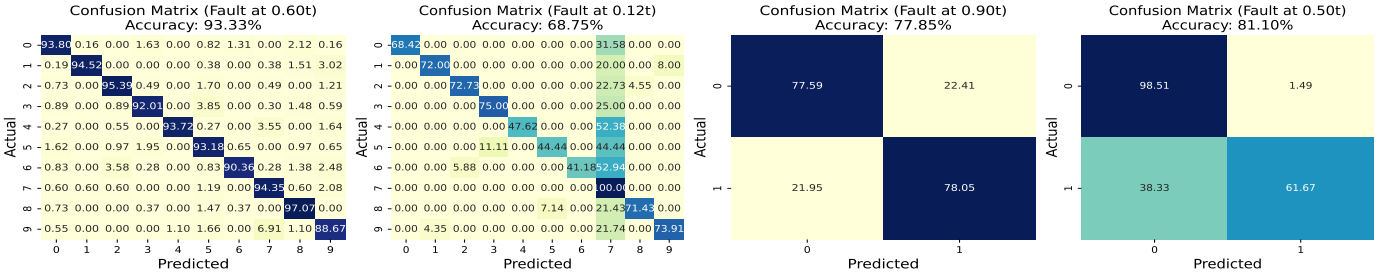
1) FastGRNN: Table I defines the FastGRNN architecture. We inject faults inside the matrix-vector computation of the first compute-heavy layer. We observe a broad vulnerable region at low target voltages and mid-range durations, with the highest fault counts at approximately 0 V and ∼75 ns duration.

Based on these sweeps, we select 0 V and 75 ns and then sweep injection time. Figure 2a shows confusion matrices for the selected t_i values. We observe that $t_i = 0.6t_s$ (where, t_s is the time representative of the ROI) yields the lowest accuracy among the shown timings: the baseline accuracy is 98.83% and drops to 93.33% under attack at $t_i = 0.6t_s$. For $t_i = 0$, the model often mispredicts a fixed label, yielding an accuracy of 15.38%. This timing coincides with firmware setup and UART initialization, so the effect likely reflects glitches landing on initialization or I/O code rather than on the matrix-vector kernel itself; nevertheless, the outcome is a repeatable, attacker-controllable failure mode.

2) TinyCNN: Table II defines the TinyCNN [35]. We inject faults during the first convolution layer. From the Stage 1 sweep, we select 0 V and 80 ns, and then perform the Stage 2 fine timing sweep over t_i. Figure 2b shows $t_i = 0.12t_s$ having only 68.75% accuracy, while the baseline accuracy is 99%. At very early injection times (around $t_i = 0$), the model frequently mispredicts a fixed output label (commonly 0 or 7), yielding an accuracy as low as 9.71%. As with FastGRNN, we interpret early timings cautiously because they can overlap with setup and UART initialization; nevertheless, the effect illustrates that small timing shifts can flip the model into highly predictable failure modes.

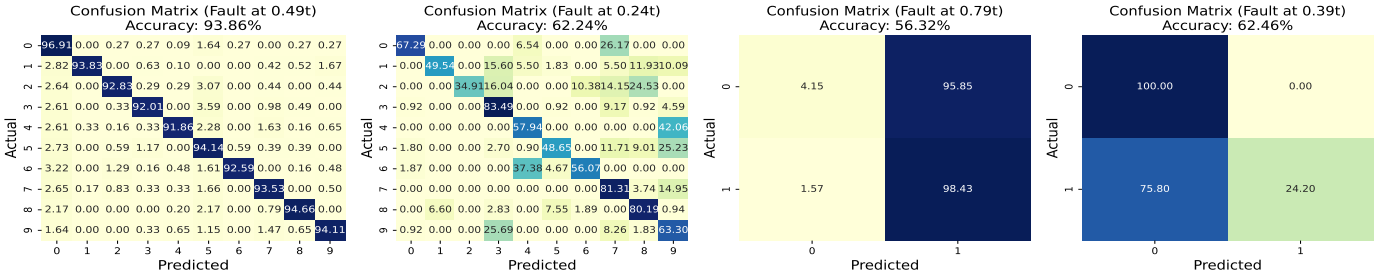
3) Logistic Regression: Table III defines the logistic regression model. We inject faults during the core weight-feature multiply-accumulate computation. From the sweep, we select 0.9 V and 85 ns as a high-fault and low-reset operating point and then sweep injection time. Figure 2c shows confusion matrices for the representative timings. The baseline accuracy is 85.19%, and drops to 77.85% under the most effective shown timing ($t_i = 0.9t_s$).

4) Wake Word: Table IV defines the Wake Word network [39]. We inject faults during the first dense layer computation. Based on the sweep, we select 0.9 V and 90 ns and then sweep injection time. Figure 2d shows confusion matrices for



(a) FastGRNN (b) TinyCNN (c) Logistic Regression (d) Wake Word

Fig. 2: Selected Pinata Voltage Glitching Confusion matrices for all the models.



(a) FastGRNN (b) TinyCNN (c) Logistic Regression (d) WakeWord

Fig. 3: Selected ChipWhisperer Voltage Glitching Confusion matrices for all the models.

representative timings. The baseline accuracy is 87.38%, and drops to 81.10% under attack at $t_i = 0.5t_s$.

B. ChipWhisperer-Husky Voltage Glitching Results

We next demonstrate voltage-glitch fault injection on the ChipWhisperer-Husky. As with the Pinata board experiments, each trial is labeled as *Correct*, *Misprediction*, or *Reset/Hang*, and we report accuracy over valid returns alongside reset rate when applicable.

1) FastGRNN: For FastGRNN, the selected Husky configuration yields a modest but consistent degradation at mid-ROI timing. At $t_i = 0.4876t_s$, accuracy is 93.86% as seen in Figure 3a. This degradation is smaller than what we observe for TinyCNN under the same Husky setup suggesting that FastGRNN is more robust than TinyCNN against ChipWhisperer-Husky crowbar glitches in our experiments.

2) TinyCNN: For the TinyCNN, ChipWhisperer-Husky glitching can induce substantially stronger disruption, but at the cost of a higher reset rate. At a very early timing ($t_i = 0.004t_s$), we observe 37.06% accuracy with a 63.45% reset rate. At a later timing ($t_i = 0.24t_s$), accuracy partially recovers to 62.24% with mispredictions biased toward a few dominant labels (notably 3, 7, and 9), as shown in Figure 3b. As in the Pinata board voltage results, we interpret early timings cautiously; nevertheless, the result demonstrates that ChipWhisperer-Husky early glitches can simultaneously (i) sharply reduce accuracy and (ii) drive the device into reset/hang regimes for CNN-style workloads.

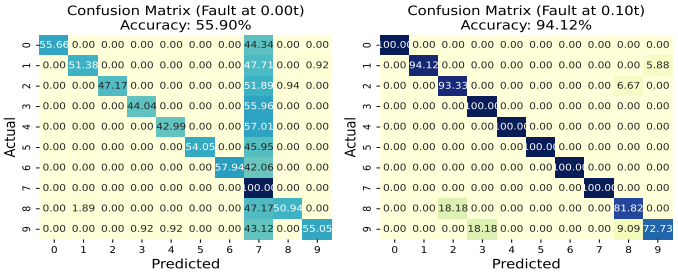
3) Logistic Regression: For Logistic Regression, ChipWhisperer-Husky voltage glitches can bias the classifier toward a dominant output. In Figure 3c ($t_i = 0.789t_s$), the model predicts class "1" for the vast majority of samples, yielding 56.32% accuracy. This behavior is consistent with the observation from the Pinata board voltage-glitching results (Section V-A) that fault injection may produce *structured*, attacker-useful failure modes (e.g., output bias or collapse) rather than purely random errors.

4) Wake Word : For the Wake Word model, ChipWhisperer-Husky glitching similarly biases the output distribution. At $t_i = 0.387t_s$, accuracy is 62.46% as seen in Figure 3d, with a strong tendency to predict the negative class (class "0") even when the true label is "1". Together, the Logistic Regression and Wake Word results reinforce the observation that even simple voltage-glitch attacks can yield predictable misclassification structure, which is particularly concerning for always-on binary decision workloads (e.g., keyword spotting).

C. Pinata Board EM Fault Injection Results

EM fault injection introduces a different fault mechanism than VDD drops: instead of directly perturbing the supply rail, an EM pulse induces local transient currents that can corrupt internal state (e.g., registers, buses, SRAM). We evaluate EM fault injection on the same Pinata board target by sweeping pulse amplitude and injection time and repeating sweeps across two probe placements in Figure 1c.

a) Probe Position 1: At Probe Position 1, we observe a noticeable number of isolated prediction faults for TinyCNN. At $t_i = 0$, the model tends to mispredict a dominant label (often "7"), visible in Figure 4. Figure 4 shows that at $t_i = 0$, accuracy drops to 55.90%, while at $t_i = 0.1t_s$, it recovers to 94.12%. Figure 5b shows the same metrics for EM fault



(a) $t = 0.0$ (b) $t = 0.1$

Fig. 4: TinyCNN confusion matrices under EM fault injection (Probe Position 1) for pulse amplitude 3.5 V and two injection times.

injection at Probe Position 1: non-reset errors peak at the earliest injection times and shift toward resets later.

b) Probe Position 2: At Probe Position 2 (only a few millimeters away), EM fault injection behavior changes qualitatively. We rarely observe isolated single-inference mispredictions; instead, when a fault does occur it often corrupts *all subsequent predictions* until the board hard resets through a power cycle (persistent corruption). This behavior appears intermittently at several injection times (e.g., $t \in \{0.30, 0.35, 0.55, 0.85\}$ in our sweeps), is not repeatable, and is compatible with corruption of architectural state such as registers or SRAM. We classify these as persistent state corruption because, once triggered, subsequent inferences continue to return incorrect outputs, and the behavior persists only until the board is reset.

Overall, EM fault injection results highlight strong *spatial sensitivity*: small probe-placement changes shift outcomes from prediction-specific faults to unstable but high-impact persistent corruption.

D. Fault Analysis and Cross-Modality Results

Beyond end-to-end accuracy and confusion matrices, we analyze how faults manifest inside the model and how outcomes differ across *voltage glitching* and *EM fault injection*. Unless stated otherwise, the analyses in this section use the same ROI triggering and trial classifier described in Section IV.

1) Voltage Fault Propagation in TinyCNN Intermediate Values: To understand *why* some glitches change intermediate activations but do not change the predicted class, we record the layer outputs produced during inference under fault injection. The TinyCNN (Table II) has three main compute stages: convolution, maxpool, and a final fully connected layer. The corresponding layer outputs have sizes 288 B (convolution), 72 B (maxpool), and 10 B (fully connected) in our `int8` implementation.

Figure 5a summarizes four metrics as a function of injection time t: (1) *Accuracy* computed over valid (non-reset) responses; (2) *Intermediate-Value Error Rate*, the fraction of non-reset trials where any recorded layer output differs from the no-fault baseline; (3) *Reset Rate*, the fraction of trials that reboot/hang; and (4) *Effective Error Rate*, the fraction of intermediate errors that actually induce a misclassification. A key observation is that intermediate-value corruption is common whenever the device survives the glitch, but only a subset of corruptions propagate to the final decision. Faults injected early within the ROI corrupt values that still feed many subsequent MAC operations in the same convolution, so a single glitch can contaminate a wide span of output bytes. Faults injected late within the ROI arrive after most of the layer's output has already been computed and committed, so they affect fewer output values and are more often overwhelmed by the classifier's margin at argmax. This highlights that end-to-end impact depends on *where* and *when* the corruption lands, not just whether any bit flips occur.

2) Bit-Level Effects in TinyCNN Convolution Outputs: Figure 6 visualizes bit-flip intensity in the convolution output vector (288 `int8` values) under voltage glitching. The x-axis indexes output bytes; the y-axis sweeps injection time; color intensity indicates the average number of flipped bits per byte relative to the no-fault output. We observe that the earliest affected bytes shift with increasing t, consistent with faults landing on different dynamic instruction windows. When comparing Figure 6 with Figure 5a, we observe that higher effective error tends to co-occur with broader and denser bit-flip regions, suggesting that widespread corruption of early-layer activations is more likely to change the final argmax decision. The early convolution output (288 B) feeds later nonlinear and aggregation stages, so broad corruption can propagate to many downstream logits. The final fully connected layer output (10 B) is closer to argmax but much smaller; a few localized flips may not change the winning logit if the classification margin is large. That is, within one layer (in this case the convolution layer), injecting early helps to change the final output.

E. Discussion

Our experiments show that embedded ML fault behavior is jointly shaped by (i) *where* the fault lands (layer/kernel), (ii) *when* it lands (timing within the ROI), and (iii) the *fault mechanism* (VDD drop vs. EM fault injection). Below we summarize the most salient cross-model and cross-modality observations.

1) Timing Sensitivity and "Predictable" Failure Modes: Across voltage glitching experiments on the Pinata board, we observe that effective errors are concentrated in specific timing windows within the early portion of the ROI (Figure 5a). At these timings, faults corrupt intermediate activations before subsequent computation can overwrite them, increasing the likelihood of altering the final argmax decision. At very early injections (around $t = 0$), we frequently observe highly *predictable* output behaviors (e.g., FastGRNN mispredicts a fixed class; TinyCNN mispredicts labels such as 0 or 7). Because $t = 0$ lies near firmware setup boundaries, these outcomes may reflect faults in initialization or I/O code rather than purely in-model computation; nevertheless, from an attacker's perspective, they still represent a practical avenue for forcing *repeatable* misbehavior.

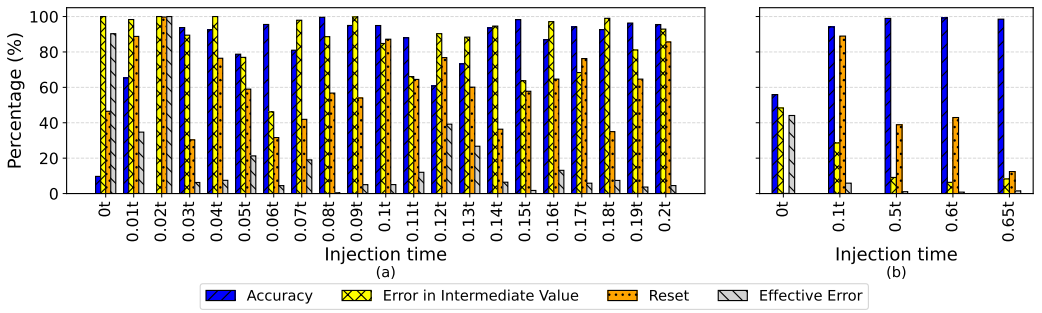


Fig. 5: Fault analysis of TinyCNN under voltage glitching and EM fault injection. (a) shows TinyCNN voltage-glitch analysis across different injection time: accuracy, intermediate-value error rate, reset rate and effective error rate. (b) shows Probe Position 1 EM fault injection analysis on TinyCNN using the same metrics.

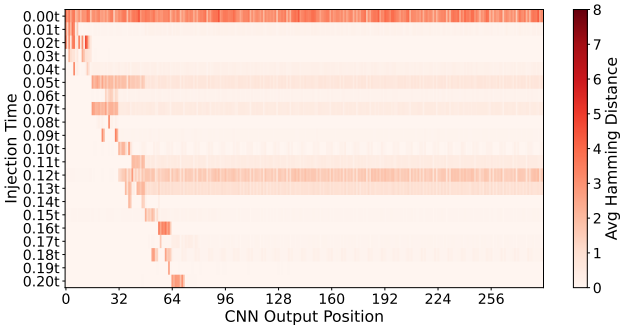


Fig. 6: Average number and position of bit flips in the TinyCNN convolution layer output as a function of injection time t.

2) Impact of Model Structure: The TinyCNN is the most vulnerable among the four workloads: small timing changes can transition between mild accuracy degradation and predictable faulty output (e.g., a drop below 20% accuracy with the model mispredicting a single class regardless of input). Logistic Regression and Wake Word sit in the middle, with accuracy drops of roughly 6–8% under the best-found voltage-glitch parameters. FastGRNN is the most robust of the four under our attacks, but it still admits regimes where glitches induce systematic misclassification for specific timing windows.

3) Intermediate Analysis Insights: End-to-end confusion matrices alone cannot distinguish between (a) faults that perturb internal activations but are later "masked" and (b) faults that propagate to the final decision. Our TinyCNN analysis shows that intermediate corruption can be frequent among non-reset trials, while effective errors are concentrated in specific timing windows (Figure 5a). Bit-level heatmaps (Figure 6) further suggest that larger and more spatially distributed corruption in early-layer activations correlates with higher effective error, supporting the intuition that early-ROI corruption contaminates a wider span of convolution outputs, which is more likely to perturb downstream logits enough to flip the argmax.

4) EM Fault Injection vs. Voltage Glitching on the Pinata Board: EM fault injection is viable but noticeably more sensitive to spatial and mechanical conditions than VDD drops. Probe Position 1 produces mostly prediction-specific faults on TinyCNN, whereas Probe Position 2 produces rare but *persistent corruption* events that corrupt all subsequent predictions until reset. This shows a tradeoff: voltage glitching is easier to parameterize and reproduce, while EM fault injection can reach fault modes (e.g., persistent state corruption) that voltage glitching cannot, though these EM fault injection modes are harder to reproduce.

VI. Protection using RSR Schemes

Randomized Self-Reduction (RSR) schemes provide a generic way to make a computation robust to random faults by evaluating randomized perturbations of the inputs and recombining the results. Erata et al. [30], [31] first applied RSR to cryptographic kernels such as modular multiplication, polynomial multiplication, and number-theoretic transforms. In this work, we adapt the same core idea to TinyML inference and focus on three dominant primitive types: (i) logistic regression (single inner product plus sigmoid), (ii) vector-matrix multiplication (used in fully connected and recurrent layers), and (iii) convolutions implemented via matrix-vector products and `im2col`. In all cases we keep the model architectures fixed and wrap the critical linear algebra kernels with a small number of RSR repetitions and a median aggregation step.

RSR is designed to tolerate transient, non-persistent computation faults during protected kernel execution. It does not guarantee recovery from persistent architectural state corruption (e.g., corrupted SRAM/register state across inferences). For persistent faults, complementary system-level defenses such as canaries and checksums on model state or state integrity checks would be required.

A. Logistic Regression Protection

Logistic regression computes a scalar score followed by a sigmoid nonlinearity,

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x} + b), \text{ where } \sigma(z) = \frac{1}{1+e^{-z}},$$

Algorithm 1 Self-Correcting Logistic Regression Algorithm

Require: Weights vector $\mathbf{w}$, input vector $\mathbf{x}$, bias $\mathbf{b}$, size n, confidence parameter β
Ensure: Corrected prediction $\hat{y}$
Definitions:
    `sigmoid`: Computes the sigmoid function, $\sigma(z) = \frac{1}{1+e^{-z}}$.
    `linear_combination`: Computes $\mathbf{w}^\top \mathbf{x} + \mathbf{b}$.
    `predict`: Combines `linear_combination` and `sigmoid` to predict $P(y = 1 \mid \mathbf{x})$.
    `rsrprop`(p, q): Computes $\frac{p\,(1-q)}{-2p\,q + p + q}$, implementing the sigmoid RSR identity that yields $\sigma(a - b)$ from $\sigma(a)$ and $\sigma(b)$.
    `calculate_median`: Returns the median of a set of values.
Calculate $N \leftarrow \lceil \log(1/\beta) \rceil$ ▷ Determine the number of iterations based on confidence parameter
Allocate space for N corrected values, `corrected_values`.
for $i = 1, \ldots, N$ **do**
    Generate random vector $\mathbf{r} \leftarrow [r_1, \ldots, r_n]$ with $r_i \sim \mathcal{U}(-1, 1)$
    Compute perturbed input $\mathbf{x_r} \leftarrow \mathbf{x} + \mathbf{r}$
    Predict:
        `pred_x_r` $\leftarrow$ `predict`$(\mathbf{w}, \mathbf{x_r}, b)$ ▷ Uses `linear_combination` and `sigmoid`
        `pred_r` $\leftarrow$ `predict`$(\mathbf{w}, \mathbf{r}, 0)$ ▷ Uses `linear_combination` and `sigmoid`
    Compute corrected value:
        `corrected_values`$[i]$ $\leftarrow$ `rsrprop`(`pred_x_r`, `pred_r`) ▷ Applies sigmoid RSR
end for
Compute the median of `corrected_values` using `calculate_median`.
return The median as the corrected prediction $\hat{y}$

where $\mathbf{w}$ and $\mathbf{x}$ are the weight and input vectors, b is the bias, and $\hat{y}$ is the predicted probability. A single transient fault in the inner product or in the evaluation of σ can therefore corrupt the final decision.

Our protection relies on two RSR identities. First, the inner product is linear, so $\mathbf{w}^\top(\mathbf{x} + \mathbf{r}) - \mathbf{w}^\top \mathbf{r} = \mathbf{w}^\top \mathbf{x}$. Second, the sigmoid satisfies a multiplicative RSR property,

$$\sigma(a - b) \;=\; \frac{\sigma(a)\,(1 - \sigma(b))}{-2\sigma(a)\,\sigma(b) + \sigma(a) + \sigma(b)},$$

which allows recovering $\sigma(z)$ from $\sigma(z + s)$ and $\sigma(s)$.

We draw a random perturbation vector $\mathbf{r}$ and evaluate the logistic model on both $\mathbf{x} + \mathbf{r}$ and $\mathbf{r}$ to obtain

$$\texttt{pred\_x\_r} = \sigma\big(\mathbf{w}^\top(\mathbf{x} + \mathbf{r}) + b\big), \quad \texttt{pred\_r} = \sigma\big(\mathbf{w}^\top \mathbf{r}\big).$$

Linearity of the inner product gives $\mathbf{w}^\top(\mathbf{x}) + b = (\mathbf{w}^\top(\mathbf{x} + \mathbf{r}) + b) - \mathbf{w}^\top \mathbf{r}$, so the sigmoid argument in `pred_x_r` decomposes into the unprotected pre-activation $\mathbf{w}^\top \mathbf{x} + b$ plus a random offset whose sigmoid-shifted counterpart is captured by `pred_r`. Applying the sigmoid RSR identity stated above to `pred_x_r` and `pred_r` then recovers $\sigma(\mathbf{w}^\top \mathbf{x} + b)$ without ever computing the unprotected inner product directly. Algorithm 1 summarizes the protected inference procedure in our implementation. `rsrprop` implements the sigmoid RSR property.

Our implementation uses the same fixed-point representation as the unprotected TinyML models, so the protected logistic layer can be dropped in place of the original one with only a modest increase in code size and latency.

B. Matrix-Vector Multiplication Protection

Matrix-vector products,

$$\hat{\mathbf{y}} = \mathbf{v}\mathbf{W},$$

dominate the compute cost in our FastGRNN and Wake Word models. Here $\mathbf{v} \in \mathbb{Z}^{1\times n}$ is an input vector, $\mathbf{W} \in \mathbb{Z}^{n\times p}$ is a weight matrix, and $\hat{\mathbf{y}} \in \mathbb{Z}^{1\times p}$ is the output. To protect this primitive we again decompose both operands into random and perturbed components,

$$\mathbf{v} = \mathbf{v}_{\text{rand}} + \mathbf{v}_{\text{pert}}, \qquad \mathbf{W} = \mathbf{W}_{\text{rand}} + \mathbf{W}_{\text{pert}},$$

and expand the product into four terms:

$$\mathbf{v}\mathbf{W} = \mathbf{v}_{\text{rand}}\mathbf{W}_{\text{rand}} + \mathbf{v}_{\text{pert}}\mathbf{W}_{\text{rand}} + \mathbf{v}_{\text{rand}}\mathbf{W}_{\text{pert}} + \mathbf{v}_{\text{pert}}\mathbf{W}_{\text{pert}}$$

This decomposition follows directly from the bilinearity of matrix multiplication: since $\mathbf{v} = \mathbf{v}_{\text{rand}} + \mathbf{v}_{\text{pert}}$ and $\mathbf{W} = \mathbf{W}_{\text{rand}} + \mathbf{W}_{\text{pert}}$, the product expands by distributivity, and no RSR-specific identity is required beyond linearity.

Each RSR repetition computes these four matrix multiplications using independently sampled randomness and recombines them to obtain a candidate result for $\mathbf{v}\mathbf{W}$. As in the logistic case we then aggregate T repetitions with an element-wise median, which filters out outliers caused by faults. To balance robustness and overhead, our C implementation exposes several matrix-multiplication backends that trade off precision and speed (e.g., 8-bit vs. 16-bit accumulators); the protected layer simply calls the appropriate backend inside the RSR loop. Algorithm 2 gives the abstract procedure that is shared across all TinyML models in this paper.

C. Convolution Protection

Convolutions in TinyCNN can be written as a sequence of vector–matrix products by applying an `im2col` transformation: each spatial patch is flattened into a vector, and each output channel corresponds to one column of a weight matrix. This allows us to reuse the protected vector–matrix primitive for convolution with minimal changes to the model code.

Let A denote the patch vector and B the corresponding weight column. We apply the same RSR decomposition

$$A = A_{\text{rand}} + A_{\text{pert}}, \qquad B = B_{\text{rand}} + B_{\text{pert}},$$

which yields four partial products (RR, PR, RP, PP) whose sum equals the original inner product. Algorithm 3 shows how the TinyCNN implementation uses this protected inner product at the patch level, wrapping the baseline convolution in a lightweight self-correcting loop without changing the network architecture.

Algorithm 2 Protected Vector-Matrix Multiplication via Self-Correcting Matrix Multiplication

Require: Input vector $\mathbf{v} \in \mathbb{Z}^{1\times n}$, weight matrix $\mathbf{W} \in \mathbb{Z}^{n\times p}$, number of iterations T
Ensure: Protected output vector $\hat{\mathbf{y}} \in \mathbb{Z}^p$
Helper Functions:
`allocate_matrix()` / `free_matrix()`: Memory allocation utilities
`random_matrix()`: Fills a matrix with uniformly random integers
`subtract_with_offset`$(\mathbf{X}, \mathbf{X}_{\text{rand}})$: Returns $\mathbf{X} - \mathbf{X}_{\text{rand}}$, adjusting for offsets
`matrix_mul_path1` to `path4`: Four matrix multiplication routines with varying input precision
`majority_vote()`: Applies element-wise majority vote (e.g., median) across repeated outputs
Convert $\mathbf{v}$ to matrix form $\mathbf{A} \in \mathbb{Z}^{1\times n}$, transpose $\mathbf{W}$ to get $\mathbf{B} \in \mathbb{Z}^{n\times p}$
$T \leftarrow 5$ ▷ Number of self-correcting repetitions
Allocate temporary matrices `temp1`, `temp2`, `temp3`, `temp4` $\in \mathbb{Z}^{1\times p}$
Initialize list `answers` of size T to hold result matrices
for $t = 1$ to T **do**
  Sample random matrices $\mathbf{A}_{\text{rand}}, \mathbf{B}_{\text{rand}}$
  Compute $\mathbf{A}_{\text{pert}} \leftarrow \mathbf{A} - \mathbf{A}_{\text{rand}}$ (with input offset)
  Compute $\mathbf{B}_{\text{pert}} \leftarrow \mathbf{B} - \mathbf{B}_{\text{rand}}$ (with weight offset)
  `temp1` $\leftarrow$ `matmul`$(\mathbf{A}_{\text{rand}}, \mathbf{B}_{\text{rand}})$
  `temp2` $\leftarrow$ `matmul`$(\mathbf{A}_{\text{pert}}, \mathbf{B}_{\text{rand}})$
  `temp3` $\leftarrow$ `matmul`$(\mathbf{A}_{\text{rand}}, \mathbf{B}_{\text{pert}})$
  `temp4` $\leftarrow$ `matmul`$(\mathbf{A}_{\text{pert}}, \mathbf{B}_{\text{pert}})$
  `answers`$[t]$ $\leftarrow$ `temp1` + `temp2` + `temp3` + `temp4`
end for
$\hat{\mathbf{y}} \leftarrow$ `majority_vote(answers)`
return $\hat{\mathbf{y}}$

Algorithm 3 Protected convolution via patch-level self-correction.

Allocate buffers $A, B \in \mathbb{Z}^M$
for $h = 0, \ldots, H_{\text{out}} - 1$ **do**
  for $w = 0, \ldots, W_{\text{out}} - 1$ **do**
    for $o = 0, \ldots, C_{\text{out}} - 1$ **do**
      Build patch vector A via im2col
      Extract weight column B from K
      ret $\leftarrow$ `protectedVecmat`$(A, B, \ldots)$
      $\hat{Y}_{o,h,w} \leftarrow$ ret
    end for
  end for
end for
Free buffers A, B

D. Protection Effectiveness

Table V compares the clean (no-fault) accuracy of the original TinyML models to their RSR-protected counterparts. Across all four workloads (FastGRNN, Logistic Regression, Wake Word, and TinyCNN), the accuracy drop from enabling protection is below 1%, confirming that the added randomization and aggregation do not significantly distort the learned decision boundaries. For the models where F1 score is also reported, the protected versions maintain nearly identical or slightly improved F1 values compared to the unprotected baselines, indicating that protection preserves not only overall accuracy but also the balance between precision and recall.

TABLE V: Accuracy and F1 Score of Unprotected and Protected Models

Model	Metric	Baseline	Faulted	Protected
FastGRNN	Accuracy	98.83	15.38–98.83	98.75
Logistic Regression	Accuracy	85.19	77.85–85.20	85.24
	F1 Score	0.8522	0.7787–0.8509	0.8526
Wake Word	Accuracy	87.38	81.10–91.23	87.36
	F1 Score	0.8540	0.7562–0.9018	0.8571
TinyCNN	Accuracy	99.07	9.71–98.36	98.80

TABLE VI: Protection Overhead

Model	Unprotected Cycles	Protected Cycles	Overhead
FastGRNN	53,196	1,791,081	34×
Logistic Regression	37,268	215,035	6×
Wake Word	4,837,353	12,175,727	2.5×
TinyCNN	4,395,603	6,993,419	1.6×

E. Protection Overhead

Table VI summarizes the runtime overhead of our protected implementations. While the RSR schemes add a multiplicative factor from repeated evaluations, the absolute cost remains within the budget of typical embedded targets, especially for security-sensitive deployments. FastGRNN's higher overhead reflects that its protected matrix-vector operation is invoked once per time step, so RSR cost compounds across the recurrence.

VII. Conclusion

Through systematic voltage glitch and EM fault injection experiments, we demonstrated the susceptibility of four TinyML models to compromised inference accuracy and, more alarmingly, the ability of attackers to manipulate model outputs to generate specific malicious labels with a high success rate. These findings emphasize the urgent need for robust fault-tolerance mechanisms in TinyML implementations. The proposed protections, leveraging random self-reducibility and majority voting for intermediate values, showed promising resilience against such attacks. This work lays a foundation for future research aimed at enhancing the security and reliability of TinyML systems in adversarial scenarios.

Acknowledgements

This work was supported in part by National Science Foundation grant 2245344. At Virginia Tech, this project is partially supported by Commonwealth Cybersecurity Initiative, by the National Science Foundation (NSF) under grants CCF-2153748, CNS-2442993, and by the Air Force Office of Scientific Research under award number FA9550-22-1-0548. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force. LLMs were used for editorial purposes, with all outputs inspected by the authors to ensure accuracy and originality.

REFERENCES

[1] Y. Abadade, A. Temouden, H. Bamoumen, N. Benamar, Y. Chtouki, and A. S. Hafid, "A comprehensive survey on tinyml," *IEEE Access*, 2023.

[2] A. Gangolli, Q. H. Mahmoud, and A. Azim, "A systematic review of fault injection attacks on iot systems," *Electronics*, vol. 11, no. 13, p. 2023, 2022.

[3] N. K. Salih, D. Satyanarayana, A. S. Alkalbani, and R. Gopal, "A survey on software/hardware fault injection tools and techniques," in *2022 IEEE Symposium on Industrial Electronics & Applications (ISIEA)*, pp. 1–7, 2022.

[4] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald, and F. D. Garcia, "{VoltPillager}: Hardware-based fault injection attacks against intel {SGX} enclaves using the {SVID} voltage scaling interface," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 699–716, 2021.

[5] T. Chiu and W. Xiong, "Sok: Fault injection attacks on cryptosystems," in *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*, pp. 64–72, 2023.

[6] L. L. Mankali, M. Nabeel, F. Raees, M. Maniatakos, O. Sinanoglu, and J. Knechtel, "{GlitchFHE}: Attacking fully homomorphic encryption using fault injection," in *34th USENIX Security Symposium (USENIX Security 25)*, pp. 8481–8500, 2025.

[7] Y. Wu, Q. Wang, and M. T. Arafin, "Shafi: Securing hash-based post-quantum cryptography from hardware fault injection attacks," in *2025 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pp. 1–4, IEEE, 2025.

[8] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of machine learning and systems*, vol. 3, pp. 800–811, 2021.

[9] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[10] A. Ghosh, D. Das, S. Ghosh, and S. Sen, "Em sca & fi self-awareness and resilience with single on-chip loop & ml classifiers," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 592–595, 2022.

[11] B. Ning and Q. Liu, "Modeling and efficiency analysis of clock glitch fault injection attack," in *2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pp. 13–18, 2018.

[12] J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Practical fault attack on deep neural networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2204–2206, 2018.

[13] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[14] F. Yao, "Deepvenom: Persistent dnn backdoors exploiting transient weight perturbations in memories," in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 244–244, IEEE Computer Society, 2024.

[15] F. Yao, A. S. Rakin, and D. Fan, "{DeepHammer}: Depleting the intelligence of deep neural networks through targeted chain of bit flips," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1463–1480, 2020.

[16] X. Zhang, A. A. Ding, and Y. Fei, "Deep-learning model extraction through software-based power side-channel," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2023.

[17] R. Sun, P. Qiu, Y. Lyu, J. Dong, H. Wang, D. Wang, and G. Qu, "Lightning: Leveraging dvfs-induced transient fault injection to attack deep learning accelerator of gpus," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 1, pp. 1–22, 2023.

[18] W. Liu, C.-H. Chang, F. Zhang, and X. Lou, "Imperceptible misclassification attack on deep learning accelerator by glitch injection," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[19] N. Khoshavi, C. Broyles, Y. Bi, and A. Roohi, "Fiji-fin: A fault injection framework on quantized neural network inference accelerator," in *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 1139–1144, IEEE, 2020.

[20] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraș, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 497–514, 2019.

[21] Y. Liu, L. Wei, B. Luo, and Q. Xu, "Fault injection attack on deep neural network," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 131–138, IEEE, 2017.

[22] F. Khalid, M. A. Hanif, and M. Shafique, "Exploiting vulnerabilities in deep neural networks: Adversarial and fault-injection attacks," *arXiv preprint arXiv:2105.03251*, 2021.

[23] H. Li, M. Ninan, B. Wang, and J. M. Emmert, "Tinypower: Side-channel attacks with tiny neural networks," in *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 320–331, IEEE, 2024.

[24] D. Genkin, N. Nissan, R. Schuster, and E. Tromer, "Lend me your ear: Passive remote physical side channels on {PCs}," in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 4437–4454, 2022.

[25] S. Potluri and A. Aysu, "Stealing neural network models through the scan chain: A new threat for ml hardware," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–8, IEEE, 2021.

[26] A. Dubey and A. Aysu, "A full-stack approach for side-channel secure ml hardware," in *2023 IEEE International Test Conference (ITC)*, pp. 186–195, IEEE, 2023.

[27] Y. Man, R. Muller, M. Li, Z. B. Celik, and R. Gerdes, "That person moves like a car: Misclassification attack detection for autonomous systems using spatiotemporal consistency," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 6929–6946, 2023.

[28] M. Blum, M. Luby, and R. Rubinfeld, "Self-testing/correcting with applications to numerical problems," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pp. 73–83, 1990.

[29] R. Rubinfeld, "Robust functional equations with applications to self-testing/correcting," tech. rep., Cornell University, 1994.

[30] F. Erata, T. Chiu, A. Etim, S. Nampally, T. Raju, R. Ramu, R. Piskac, T. Antonopoulos, W. Xiong, and J. Szefer, "Systematic use of random self-reducibility in cryptographic code against physical attacks," 2024.

[31] F. Erata, O. Paradise, T. Typaldos, T. Antonopoulos, T. Nguyen, S. Goldwasser, and R. Piskac, "Learning randomized reductions," 2026.

[32] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma, "Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," *Advances in neural information processing systems*, vol. 31, 2018.

[33] M. Research, "Fastgrnn." https://github.com/microsoft/EdgeML/tree/master.

[34] K. Thommas, "Tinyml: Convolutional neural networks (cnn)," 2022.

[35] B. Artley, "Mnist keras simple cnn 99.6%," 2024.

[36] TensorFlow, "Post-training quantization," 2024.

[37] ARM-software, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," 2024.

[38] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*. John Wiley & Sons, 2013.

[39] P. Mishra, "Wakeworddetection," 2024.

[40] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," 2019.