# Enhancing DNN Accelerator Integrity via Selective and Permuted Recomputation

Jhon Ordoñez and Chengmo Yang
Electrical and Computer Engineering, University of Delaware
{jordonez,chengmo}@udel.edu

## ABSTRACT

Hardware accelerators have been widely deployed in many machine learning applications due to their superior performance and energy efficiency. However, these accelerators are vulnerable to fault injection attacks, compromising their integrity and reliability. In particular, recent studies have revealed a targeted attack on black-box DNN models, which, through glitching the execution of the fully connected (FC) layer, is capable of derailing the DNN outputs to arbitrary classes. To defend DNN accelerators against this severe attack, this paper proposes a selective and permuted recomputation scheme. Instead of adopting dual or triple modular redundancy, which incurs high overhead, the proposed scheme selects a subset of critical FC outputs for recomputation. Meanwhile, it permutes the computation of the FC layer to prevent an adversary from pinpointing the exact time of executing the target class. The proposed defense is evaluated on three popular DNN models, namely, ResNet-50, InceptionV3, and MobileNetV3. Results show that under fault injection attacks, it can successfully recover 90-95% of the models' original accuracy, achieved with less than 1.61% runtime overhead and no storage overhead.

## KEYWORDS

DNN accelerator integrity, Clock glitching, Fault-tolerant execution

**ACM Reference Format:**
Jhon Ordoñez and Chengmo Yang. 2024. Enhancing DNN Accelerator Integrity via Selective and Permuted Recomputation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3676536.3676842

## 1 INTRODUCTION

Deep Neural Networks (DNNs) have been widely used in many application domains, such as computer vision, natural language processing, and autonomous vehicles, among others. To accommodate DNNs of ever-increasing sizes on resource-constrained devices, hardware accelerators have played an important role since they offer high performance and energy efficiency. Unfortunately, the advances in DNN deployment also expose them to various security threats, especially fault injection attacks which disturb the DNN functionality to produce malicious outputs, compromising the reliability and integrity of safety-critical systems.

Malicious faults can be injected into a DNN accelerator through a variety of means such as Rowhammer [13], clock glitching [12], voltage glitching [14], or Dynamic Voltage and Frequency Scaling (DVFS) [23]. Depending on their impact on the model outputs, fault injection attacks can be classified as either *random*, wherein the adversary cannot control the output class to which model inputs are misclassified; or *targeted*, wherein the adversary intentionally misleads inputs to a target output class. Most targeted attacks are hard to perform as they require a certain amount of model knowledge and a complex attack scheme. For example, the targeted bit-flip adversarial weight attack (T-BFA) [19], which intentionally misclassifies inputs from one class to another, requires complete knowledge of the model and part of the training data to perform a complex weight-bit search algorithm. However, a recent work has demonstrated a targeted fault injection attack requiring no prior knowledge of the DNN model [18]. Demonstrated on a Deep-Leaning Processing Unit (DPU) with faults injected via clock glitching, this attack discovers that the DPU executes the fully connected (FC) layer in order, which allows a linear relationship to be established between the offset of clock glitching and the affected classes.

To defend DNN accelerators against this targeted fault injection attack, in this work we propose a permuted and selective recomputation scheme. By permuting the execution of the FC layer in the DPU, our scheme prevents an adversary from pinpointing the exact time when the target class is computed, making the attack random. Meanwhile, we propose a frugal scheme to detect and recover the injected faults via selectively recomputing only a small group of FC outputs, namely, those who may affect the final prediction outcome. This permuted and selective recomputation framework is deployed on an FPGA, configured to implement a DPU to accelerate three popular DNN models, namely, ResNet-50 [4], InceptionV3 [22], and MobileNetV3 [8], all trained on the ImageNet dataset [2]. Overall, the main contributions of this work include:

- The proposal of a frugal and effective scheme to protect the most vulnerable layer in DNN models against targeted fault injection attacks.
- The reverse engineering of the DPU to implement the proposed defense and increase its controllability.
- The thorough evaluation of the proposed defense in terms of its fault detection/recovery capability and overhead.

The rest of this paper is structured as follows: Section 2 briefly reviews fault injection attacks on DNN models and existing countermeasures. Section 3 characterizes the impact of faults on the outputs of the FC layer and DNN final output, which provides a theoretical foundation for the proposed defense scheme. The defense scheme and its implementation are introduced in Section 4,


*ICCAD '24, October 27–31, 2024, New York, NY,USA*

ACM ISBN 979-8-4007-1077-3/24/10
https://doi.org/10.1145/3676536.3676842

while Section 5 presents the experimental results. Finally, Section 6 concludes this work.

# 2 BACKGROUND

## 2.1 Fault Injection Attacks

Fault injection attacks exploit electrical-level security risks [15] to alter the intended behavior of a device, so as to bypass its security or compromise its integrity. Conventional fault injection attacks target crypto engines, while recent work has started examining vulnerabilities of DNN hardware accelerators.

One set of attacks inject faults into DNN weight memory. For instance, the attack in [3] undervoltes FPGA on-chip memories to cause 30% bit-flip errors in DNN memory. Yet the model accuracy is reduced only by 4.92% because faults are injected randomly, considering the DNN model as a black-box. In comparison, the bit-flip attack (BFA) in [24] leverages Rowhammer [13] to alter a small set of targeted bits (3 to 24 bits) in model weights so as to degrade the prediction accuracy to random guesses (i.e. untargeted attack). The set of bits to flip is precisely selected via a gradient-based search algorithm. A later work called T-BFA modifies the search algorithm to implement a targeted attack that intentionally misleads DNN inputs to a target output class [19]. Both BFA and T-BFA require complete knowledge of the DNN model and part of the training data (i.e., white-box attacks).

Another set of attacks inject faults into the intermediate results of DNN computation via clock glitching [12], voltage glitching [14], or Dynamic Voltage and Frequency Scaling (DVFS) [23]. The attack in [12] glitches the DSP clock on an FPGA that runs a DNN model during a high percentage of the total inference time. The work in [14] proposes a power striker that aggressively overloads the shared Power Distribution Network (PDN) to incur voltage glitches. Likewise, the attack in [21] sets the working voltage of Nvidia GPUs to a deficient value. All of these attacks are *untargeted*, that is, inputs are misclassified into random output classes.

Overall, regardless of the fault injection approach or the accelerator platform, most of the previous fault injection attacks are untargeted. Targeted attacks, such as T-BFA, require complete knowledge of the DNN model and part of the training data to perform.

## 2.2 Derailed Attack

A recent work [18] has revealed a targeted fault injection attack on black-box DNN models, capable of derailing inputs to a targeted range of classes. Here we briefly review this attack, which is the vulnerability that this work aims to mitigate.

The derailed attack [18] adopts clock glitching to cause timing violations and, consequently, computation errors in the DPU. The attack uses two parameters: **offset** and **width**, to control the location and duration of the glitch. The authors discovered two important facts: (1) the FC layer is most vulnerable to fault injection attacks. Faults injected in a convolutional layer have a significant chance of being masked by the activation function or the max pooling layer, whereas faults injected in the FC layer directly affect DNN outputs; and (2) the DPU executes the FC layer in order, with multiple classes computed at a time. Based on these facts, the attack establishes a linear relationship between the glitch offset and derailed predictions in a certain range of classes, as shown in Eq. (1):

$$\text{offset} = T_{FCstart} + \frac{C_{target}}{C_{total}} \times (T_{FCend} - T_{FCstart}) \quad (1)$$

where $C_{target}$ is the index of the target class, $C_{total}$ is the total number of classes, and $T_{FCstart}$ and $T_{FCend}$ are the start and end time of computing the FC layer in the DPU, respectively. The derailed attack adopts a two-step search process: a coarse-grained search that aims to locate the start and end of the FC layer execution ( $T_{FCstart}, T_{FCend}$), followed by a fine-grained search that examines the impact of glitch offset on $C_{target}$, verifying Eq. (1). This attack is implemented in the FPGA and evaluated on three DNN models. A short glitch of 10 clock cycles is able to derail 80-90% of inputs into the range of target classes, as reported in [18].

## 2.3 Existing Countermeasures

Different defense mechanisms have been developed for DNN accelerators to detect and/or mitigate fault injection attacks.

A large portion of prior work was focused on attacks that modify the weights of DNNs. One set of work involves retraining the model. For instance, Lee *et al.* [9] propose converting a regular FC layer to a Bipolar FC layer to eliminate those with highly sensitive bits. Likewise, RA-BNN [20] is a binary neural network derived from an 8-bit quantized model. While it improves resilience against BFA, the accuracy of BNN is lower. Another set of works performs online detection of bit-blips leveraging coding techniques. In [10], a value-aware parity insertion Error Correction Code (ECC) is introduced, which inserts parity bits to high-order bits in weight values as they contribute more to accuracy degradation. Likewise, [6] protects the integrity of the high-order bits by establishing a property in the sum of a group of weights in each layer, which achieves a 100% detection rate of bit-flip chains. The third set of work mitigates faults via duplicating important neurons or weights. Li *et al.* [11] quantify the sensitivity of each neuron to determine the set of duplications, whereas Baek *et al.* [1] duplicate 50% of the weights by ranking them according to the sum of weights in a layer and their second-order derivatives. These duplication strategies incur high computation and storage overhead. In comparison, the technique in [7] tolerates permanent faults in weight memory without any duplication, via setting the fault-free bits in weight memory to effectively approximate weight values.

A few previous works have attempted to tolerate computation errors on activations or weights. The work in [5] replaces the unbounded activation functions with a clipped version. It uses a subset of the validation dataset to fine-tune the clipping thresholds. Likewise, Zhan *et al.* [25] design a boundary-aware ReLU (BReLU), ensuring that a deviation between the boundary and original outputs is not large enough to affect the final DNN output. The boundaries are determined based on a gradient-ascent algorithm. The online testing approach in [16] monitors the accuracy drop with a small set of test images to identify the corresponding fault type and predict the severity of faults. An extended analysis is presented in [17], exploiting test image selection based on output probability distribution, gradient sensitivity, or neuron coverage.

Overall, most of the previous countermeasures aim to tolerate bit-flip errors in model parameters rather than computation errors caused by timing violations. Countermeasures that tune the

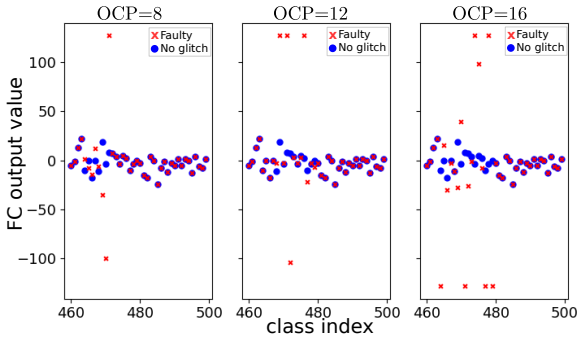


**Figure 1: Impact of OCP on the number of FC outputs affected by a single clock glitch, tested for ResNet-50 with glitch offsets targeting classes around 464 and glitch width=20*ns*.**

activation function are effective for tolerating faults injected in convolutional layers rather than the FC layer. In contrast, this work aims to develop a countermeasure for the *derailed* attack, which injects faults into the FC layer. The proposed countermeasure requires no retraining or prior knowledge of model parameters. The memory overhead is zero, while the runtime overhead is negligible.

# 3 TECHNICAL MOTIVATION

## 3.1 Detailed Attack Analysis

To develop an effective countermeasure against the Derailed Attack, we first conduct a thorough analysis on it. For this purpose, we adopt the same fault injection setup that implements the DPU and clock glitching circuit on an AMD KRIA KV260 board. The glitch circuit features a multiplexer that switches between two clock sources. The width and offset of the glitch are set through the Advanced eXtensible Interface (AXI). More details can be found in [18].

One critical observation made in [18] is that when computing the FC layer, the DPU executes multiple classes at a time, and this parameter equals the Output Channel Parallelism (OCP), which can be set to 8, 12, or 16. This implies that *a single injected fault affects multiple classes at the same time*. To verify this, we conduct an experiment to inject a clock glitch into ResNet-50 at an offset targeting to affect classes around 464 according to Eq. (1). The model is trained on ImageNet (containing 1000 classes) and quantized to 8-bit, which means weights are in the range of [-128, 127]. Fig. 1 plots the raw FC output values before applying softmax. As can be seen, as the OCP increases, the injected fault causes deviations in more classes. Moreover, the fault affects the computation in both positive and negative directions, with some faulty values very close to the extremes (127 or -128). However, their impact is asymmetric: *high positive faulty values typically lead to misclassification into the target classes*, whereas low positive or negative faulty values do not alter prediction outcomes in most cases.

Based on these observations, we propose a defense scheme that *permutes the execution of the FC layer*, so as to prevent an adversary from pinpointing the exact time when the target class is computed. Meanwhile, to efficiently detect a fault, we propose a selective recomputation scheme that *recomputes a group of OCP classes only if the group contains large positive values.*

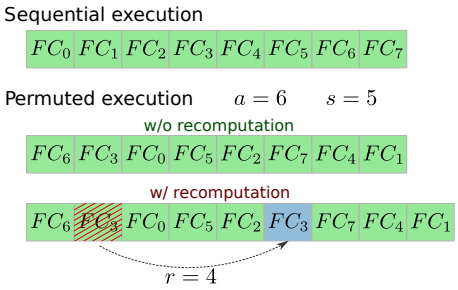


**Figure 2: Permuted sequence of execution for 8 groups, assuming an initial value $a = 6$, a stride $s = 5$, and recomputation at $r = 4$.**

## 3.2 Threat Model

We assume that the adversary gains access to the hardware accelerator through an interface or shares hardware resources. The adversary can control and configure fault injection by setting different parameters via software (e.g., glitch offset and width). The adversary performs a fault injection attack following the method in [18]. The adversary has no prior knowledge of the DNN parameters, but can inspect the DNN outputs to reverse-engineer the model and cause misclassifications into a specific range of classes. The attack is targeted, meaning that the adversary would not glitch the execution of the entire FC layer.

Regarding the defender's capability, we assume that the defender can modify the hardware accelerator to implement the proposed scheme. The defense requires no retraining or prior knowledge of the DNN model parameters.

# 4 PROPOSED DEFENSE

This section explains the algorithms for permuting the computation of the FC layer and selectively recomputing a group of classes, as well as the methods for implementing the algorithms into the DPU.

## 4.1 Permuted Computation

To prevent an adversary from pinpointing the target classes to attack with Eq. (1), one idea is to introduce randomness to the execution of the FC layer. However, as the DPU computes a consecutive group of classes in the FC layer at a time, *the execution must still be performed in groups of OCP classes to preserve intra-group locality*. For this reason, we design an algorithm that rearranges the order of computation with two parameters, namely, a stride $s$ and a random initial value $a$. Assume $N$ is the total number of classes in the DNN model, the FC layer is split into $g = \lceil N/OCP \rceil$ groups: $FC_0, FC_1, \ldots, FC_{g-1}$. The index of the $i$-th group to execute can be computed with the following equation:

$$idx = (a + i \times s)\%g, \text{ for } i = 0, 1, 2.., g-1, 0 \leq a < g \qquad (2)$$

To ensure all $g$ groups are calculated, the greatest common divisor of $s$ and $g$ needs to be one: $gcd(s, g) = 1$. As a concrete example, Fig. 2 presents 8 groups, $FC_0$, $FC_1$, ..., $FC_7$, calculated in order initially. Permuted execution determines the order of computation with Eq. (2) assuming $a = 6$ and $s = 5$. Note that this permutation does not disrupt intra-group locality and requires no migration of data in memory, thus can be easily incorporated into the DPU.

In case of a suspicious group that contains high output values, a recomputation of that specific group needs to be scheduled. To defend against an advanced attacker who may have knowledge of the defense scheme, the recomputation cannot be consistently scheduled at the end of the FC layer computation, otherwise the attacker could inject a fault at that location to corrupt the recomputation. Likewise, the recomputation cannot be scheduled right after the original computation of that group, otherwise the attacker could inject a longer glitch to corrupt both runs. Given these considerations, we propose to use a random parameter $r$ to schedule the recomputation $r$ steps after the first computation ($r < g - idx$). This is also illustrated in Fig. 2. If the execution of $FC_3$ is potentially faulty, its recomputation is scheduled after $r = 4$ FC groups.

## 4.2 Selective Recomputation

*4.2.1 Fault Detection.* With permuted FC computation, it is almost impossible for an adversary to glitch the same group twice. As a result, faults in an FC group can be detected via recomputing the group and comparing the outputs of the two runs. One straightforward approach is dual modular redundancy (DMR), which would recompute the entire FC layer. While it effectively detects all faulty FC groups, the disadvantage is the high recomputation overhead, which is consistently imposed even when the accelerator is not under fault injection attack.

Instead of adopting DMR, we develop a frugal selective recomputation scheme, aiming to capture not all faulty groups, but only those containing high positive faulty values that may lead to misclassification. The specific strategy is given below:

**Selective recomputation:** *recompute an FC group if and only if it contains $k$ values that are greater than or equal to a threshold $T$.*

The two parameters $k$ and $T$ can be adjusted to balance detection sensitivity and specificity. Setting $k = 1$ will select a number of benign groups for recomputation because even in the fault-free case, some classes may have large positive values. On the other hand, as a clock glitch affects a group of OCP classes at a time, it is likely for more than one class to have large positive values, as illustrated in Fig. 2. Therefore, setting $k = 2$ will reduce the number of false positives (FP) but may cause more false negatives (FN). Likewise, a larger $T$ will select fewer groups for recomputation, thus reducing FP but increasing FN. The best combination of $k$ and $T$ should minimize the sum of FP and FN while maximizing true positives (TP). We propose the following metric to select these model-specific parameters, wherein $p$ is a parameter reflecting the relative importance of FN vs FP:

$$\text{metric} = \frac{FP + p \times FN}{(1+p) \times TP} \tag{3}$$

*4.2.2 Fault Recovery.* We propose and evaluate two fault recovery methods, namely, an *approximate recovery* that requires no extra recomputation of a faulty group, and a *precise recovery* that computes a faulty group for a third time.

While computing an FC group twice can effectively detect mismatches, it is insufficient to precisely recover the detected fault. Instead, our *approximate recovery* leverages the fact that if two values mismatch, it is more likely for the one with a larger absolute value to be faulty. Accordingly, this method takes the value closer to zero as the "clean" value. The benefit is that this method adds no extra runtime overhead to the DPU, making it more suitable for applications that require minimum overhead and high time predictability.

In the case of *precise recovery*, if two runs of one FP group have mismatching results, another recomputation of the corresponding FC group will be scheduled, and the median of the three values will be used as "clean" value. Note that this strategy requires augmenting the permuted computation procedure to conditionally schedule a third run, save the extra output vector, and compute the final output in a more complex manner. Therefore, it is preferred only if the application is less strict in terms of overhead and time predictability.

## 4.3 Implementation

Algorithm 1 presents the implementation of permuted and selective FC recomputation, which is executed during inference upon completing all the layers prior to the FC layer. At the beginning, the stride $s$ and the initial value $a$ are picked, which are unknown to the attacker. Then, the main execution loop computes the $g$ FC groups in an order determined by Eq. (2). The execution of every FC group ($FC_{idx}$) generates multiple outputs (= OCP) stored in a vector $y_{idx}$. If the inspection of $y_{idx}$ indicates a potential fault, the random value $r$ is chosen and used to calculate the recomputation location $l$, while $f$ stores the current group index. During subsequent computation, once the index $l$ is reached, the recomputation of $FC_f$ is performed. This generates a new output vector $y'$, which is analyzed together with $y_f$ by the recovery procedure for fault detection and recovery.

The selective recomputation scheme is implemented as the "Inspect" procedure. It takes two inputs, $T$ and $k$, and inspects $y_{idx}$, a specific group of OCP outputs. The group is selected for recomputation if it has $k$ or more elements no smaller than $T$. Likewise, the "Recovery" procedure implements the *approximate recovery* method, which is selected for implementation given its low complexity and high effectiveness (to be shown in Section 5). This procedure takes two vectors of OCP elements as inputs, namely, $y_f$ (first run) and $y'$ (second run). If the two results of a class $i$ do not match, the value closer to zero is taken as the recovered value.

In terms of hardware implementation, one advantage of the proposed defense mechanism is that it only requires basic arithmetic operations and control logic, which can be easily implemented inside the programmable logic (PL) along with the DPU. Fig. 3 presents a structural diagram of the proposed implementation, which only requires minimum modification to the DPU. Specifically, the DPU only needs one extra input, namely, the FC group index $FC_{idx}$, and produces one extra output vector $y_{idx}$. The *Permuted computation* module implements a state machine to generate $FC_{idx}$ for the DPU. The parameters $(a, s, g)$ that define the permuted execution sequence can be received through any interface from software implementation, such as the Advanced eXtensible Interface (AXI). Meanwhile, this module interacts with the *Selective recomputation* module by receiving a 'faulty' signal, using it to schedule recomputation, and generating a 'recover' signal to trigger the recovery procedure. The *Selective recomputation* module has two blocks: Detection and Recovery. The detection block implements a counter and multiple comparators, taking thresholding parameters $k$ and $T$ as inputs and

**Algorithm 1** Permuted and Selective Recomputation of FC layer

$g \leftarrow \lceil N/OCP \rceil$
Choose $s \mid gcd(s, g) = 1$
Choose a random $a \mid 0 \leq a < g$
$FC$ groups $\rightarrow FC_0, FC_1, FC_2, \ldots, FC_{g-1}$
Output array $y \rightarrow y_0, y_1, y_2, \ldots, y_{g-1}$
**for** $i = 0, 1, 2, \ldots, g-1$ **do**
  $idx \leftarrow (a + i \times s)\%g$
  $y_{idx} \leftarrow run(FC_{idx})$
  $recompute \leftarrow$ INSPECT$(y_{idx})$
  **if** *recompute* is true **then**
    Choose a random $r \mid 1 \leq r < g - i$
    Recomputation location $l \leftarrow i + r$
    $f \leftarrow idx$
  **end if**
  **if** $l = i$ **then**
    $y' \leftarrow run(FC_f)$
    $y_f \leftarrow$ RECOVERY$(y_f, y')$
  **end if**
**end for**

---

**procedure** INSPECT$(y_{idx})$
  $T$: value threshold, $k$: count threshold
  Count $c \leftarrow 0$
  **for** $i = 0, 1, 2, \ldots, OCP - 1$ **do**
    **if** $y^i_{idx} \geq T$ **then**
      $c \leftarrow c + 1$
    **end if**
  **end for**
  **if** $c \geq k$ **then return** true
  **else return** false
  **end if**
**end procedure**

---

**procedure** RECOVERY$(y_f, y')$
  **for** $i = 0, 1, 2, \ldots, OCP - 1$ **do**
    $y^i_f \leftarrow closest\_to\_zero(y^i_f, y'^i)$
  **end for**
**end procedure**

generating the 'faulty' signal. It also saves the vector $y_{idx}$ to internal memory if the recomputation condition is met. Likewise, the recovery block has multiple comparators to compare two vectors: the $y_{idx}$ sent by the DPU and the one saved in its internal memory. It selects the values closer to zero to form the recovered output vector $FC_{output}$.

# 5 EXPERIMENTAL RESULTS

## 5.1 Experimental Setup

Fig. 4 presents our experimental setup for evaluating the proposed defense scheme against targeted fault injection attacks. The DPU, the clock glitching circuit, and the proposed defense are all implemented on an AMD KRIA KV260 FPGA board that combines a Processing System (PS) for software implementations and Programmable Logic (PL) for hardware implementations. To control

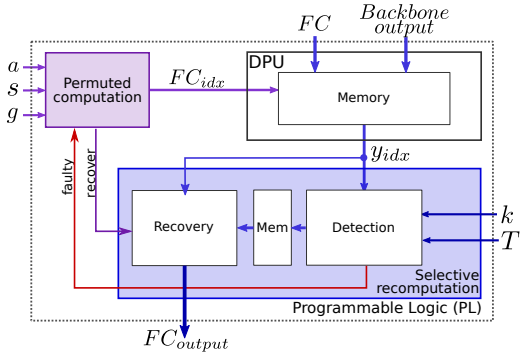


**Figure 3: Proposed hardware implementation of the defense mechanism along with the DPU.**

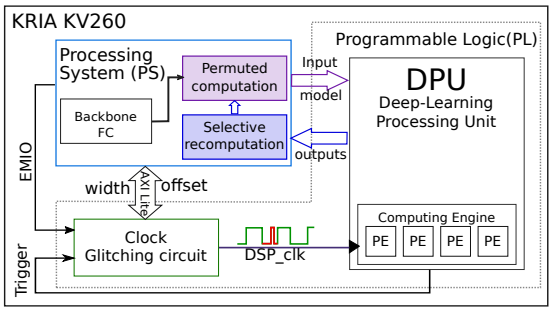


**Figure 4: FPGA setup that implements the DPU, the clock glitching circuit, and the proposed defense.**

fault injection, the PS communicates with PL through the Advanced eXtensible Interface (AXI) to set the **width** and **offset** for the clock glitching circuit to generate a faulty *DSP_clk* for DPU's computing engine. The proposed defense is implemented as two software modules, *permuted computation* and *selective recomputation*, within the PS. The interaction between these modules and the DPU follows the structure presented in Fig. 3.

To evaluate the effectiveness of the proposed defense, we have conducted fault injection experiments on three popular DNN models: InceptionV3 [22], ResNet-50 [4], and MobileNetV3 [8], all trained on the ImageNet [2] dataset and quantized to 8 bits using the Xilinx Vitis AI framework. The experiments use about 4000 images, randomly selected from the dataset, to evaluate model accuracy under the influence of fault injection and the mitigation mechanism.

## 5.2 Evaluation of Permuted Computation

As explained in Section 4.1, permuted computation executes the FC layer in an out-of-order sequence whose initial parameters are unknown to the adversary. To evaluate the effectiveness of this defense in preventing an adversary from attacking a specific range of classes, we randomly selected three images to perform three fault injection runs to the ResNet-50 model, all sharing the same glitch offset of $8.49ms$ and the same width of $20ns$. Note that only permuted computation was applied, whereas selective recomputation was disabled in these experiments. Fig. 6 plots the raw FC outputs of these runs. It can be seen that although the same glitch offset was used, each fault produced spikes on a different range of classes, preventing a targeted attack.

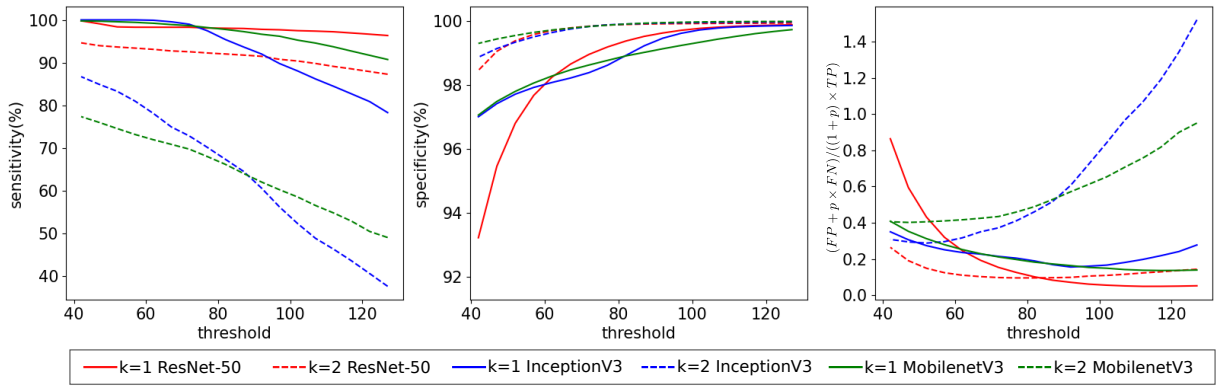


**Figure 5: Exploration of parameters $k$ and $T$ on their impact on sensitivity (=$TP/(TP+FN)$), specificity (=$TN/(TN+FP)$), and the metric defined in Eq. (3).**

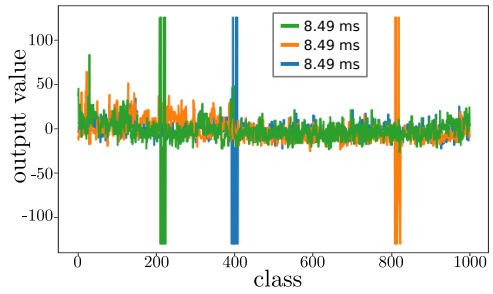


**Figure 6: Impact of permuted computation on shifting the classes under attack. All three runs have a clock glitch at the offset 8.49$ms$. Data collected on ResNet-50.**

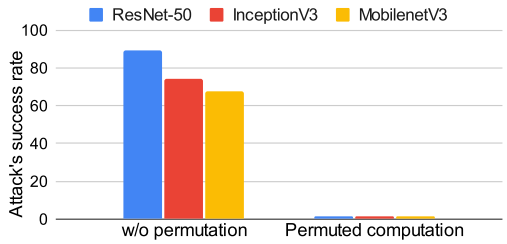


**Figure 7: Impact on attack success rate**

Moreover, we also evaluated the impact of permuted computation on the success rate of the attacks, defined as the percentage of images predicted as a class in $R_{target}$ [18]. We randomly selected ~300 images and injected faults at different offsets, with a glitch width of 20$ns$. As shown in Fig. 7, without applying the proposed defense, the success rate of the attack is 67-90%. In comparison, permuted computation significantly decreases the success rate to 1.33% for Resnet-50, 1.13% for InceptionV3, and 1.26% for MobileNetV3. A few images could still be successfully derailed since the permuted $idx$ may equal the target $C_{target}$ with a chance of OCP/1000 = 1.6%. Overall, these experiments confirm that permuted computation effectively mitigates a targeted fault injection attack.

## 5.3 Evaluation of Selective Recomputation

*5.3.1 Fault detection design space exploration.* As explained in Section 4.2, the detection algorithm flags an FC group if $k$ of its results are greater than or equal to a threshold $T$. Our first set of experiments explores these two parameters, aiming to identify the best combination that reduces the False Positives (FP) and False Negatives (FN) while increasing the True Positives (TP). These cases are defined as follows:

- True Positives (TP): A faulty FC group flagged by the detection algorithm;
- False Positives (FP): A fault-free FC group (that contains high positive values) flagged by the detection algorithm;
- False Negatives (FN): A faulty FC group that alters the final prediction outcome but is not flagged;
- True Negatives (TN): An unflagged FC group that is either clean or contains benign faults that do not alter prediction.

The recomputation overhead, which is critical to fault-free executions, is largely affected by the FP count. In contrast, the detection rate of critical faults is largely affected by the FN count. Considering both factors, we conducted 8000+ experiments combining 40% of no-glitch runs and 60% of glitched runs. Glitches were injected at different locations during the inference. Since around 80% of the glitches cause a misprediction, these experiments effectively balance the impact of fault-free and faulty cases.

Fig. 5 presents the impact of $k$ and $T$ on three metrics: *sensitivity*, *specificity*, and the metric defined in Eq. (3). The factor $p$ is used to quantify the relative importance of FN vs FP in the target application. Without the loss of generality, we use $p = 10$.

If $k = 2$, i.e., the detection algorithm flags an FC group if at least two class outputs reach to $T$, the detection specificity can be largely enhanced. These cases are shown with the dashed lines in the middle figure of Fig. 5. However, this also largely degrades model sensitivity, as shown by the dashed lines in the leftmost figure. As detection sensitivity is more crucial, we recommend using $k = 1$.

In terms of the threshold $T$, as expected, a larger $T$ will reduce sensitivity but increase specificity. Selection of the best $T$ value for each model can be done by identifying the minimum value of the metric presented in the rightmost figure. Specifically, for ResNet-50 (solid red line), setting $T$ to a value around 110 reaches the lowest value of the metric. Likewise, MobileNetV3 (solid green line) favors a threshold $T$ around 120, while InceptionV3 (solid blue lines) favors a $T$ around 100. To reduce the hardware resources

**Table 1: Comparison of proposed defense against TMR in terms of recovered accuracy (%) and overhead (%)**

| | Original accu. | Faulty accu. | Recovered accu. - 1 fault | | | Recovered accu. - 2 faults | | | Overhead | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 2 runs | 3 runs | TMR | 2 runs | 3 runs | TMR | 2 runs | 3 runs | TMR |
| ResNet-50 | 77.675 | 11.196 | 74.097 | 74.197 | 77.675 | 73.071 | 73.369 | 77.642 | 1.511 | 2.966 | 200 |
| InceptionV3 | 92.014 | 19.072 | 85.813 | 86.062 | 92.014 | 85.243 | 85.764 | 92.008 | 1.610 | 3.112 | 200 |
| MobilenetV3 | 78.634 | 30.354 | 71.152 | 72.86 | 78.634 | 70.700 | 72.784 | 78.534 | 1.447 | 2.615 | 200 |

required for implementing the comparators, a $T$ value whose binary representation has consecutive 1's in the high-order bits is recommended. Given this consideration, the threshold value for Resnet-50 is **112**, which is 0**1110**000 in binary. Likewise, the threshold for InceptionV3 is $T$=**96** (0b0**11**00000) and MobileNetV3 can take $T$=**120** (0b0**1111**000).

*5.3.2 Comparison against DMR.* Our second set of experiments compares selective recomputation against dual modular redundancy (DMR) in terms of its fault detection capability and runtime overhead. In particular, the detection capability is evaluated with two metrics: the percentage of faulty groups (with mismatches) flagged, and more importantly, the percentage of critical faults (that alter the prediction) flagged. The runtime overhead is evaluated as the percentage of FC groups recomputed. These experiments use the model-specific $k$ and $T$ values selected before for fault detection. Each experiment run injects a single glitch during FC execution.

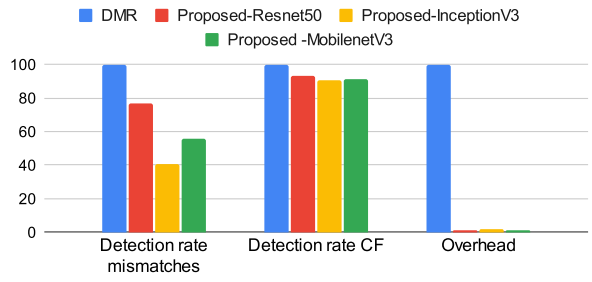


**Figure 8: Detection rate and overhead comparison between DMR and the proposed scheme.**

Fig. 8 presents the comparison results. Note that the DMR results are not replicated for different models since it re-executes the entire FC layer and thus consistently detects 100% mismatches in FC group outputs and 100% critical faults, while imposing 100% recomputation overhead. In comparison, our selective recomputation only imposes negligible overhead, specifically, 1.51% for ResNet-50, 1.61% for InceptionV3, and 1.45% for MobilenetV3. Meanwhile, it is able to detect 41-77% of mismatches in FC group outputs. This is expected as our scheme is designed to capture not all the faults, but only those high positive faulty values that tend to cause misprediction. For critical faults, our scheme offers much higher detection rates at 93.31%, 90.27%, and 91.26% for ResNet-50, InceptionV3, and MobilenetV3, respectively.

Overall, the comparison against DMR confirms that our selective recomputation is a frugal (62X overhead reduction) and highly effective fault detection scheme (90%+ coverage of critical faults).

*5.3.3 Comparison against TMR.* Our third set of experiments compares selective recomputation against triple modular redundancy (TMR) in terms of its fault recovery capability and runtime overhead. Specifically, both the approximate and precise recovery methods in Section 4.2 have been implemented. Furthermore, we established two scenarios where the adversary injects one or two faults in a run. A double-fault run is expected to be more difficult to recover.

Table 1 presents the recovered model accuracy achieved by the two variants of the proposed defense and TMR. Without fault recovery, one can observe that the injected faults largely degraded model accuracy to 11-30%. By executing the entire FC layer three times, TMR was able to recover the model back to its original accuracy at the cost of 200% overhead. In comparison, our recovery scheme is much more efficient and still highly effective. In the scenario of a single injected fault, our approaches (2 and 3 runs) recovered the model accuracy close to the original, precisely at 95.4%, 93.3%, and 90.5% for ResNet-50, InceptionV3 and MobileNetV3, respectively. When two faults were injected, the recovered accuracy was only slightly lowered. This is because with permuted computation, the chance for two faults to affect the same FC group is extremely low. Comparing the two recovery approaches, it can be seen that performing an extra recomputation (3 runs) only gives a tiny improvement in recovered accuracy. This reveals two facts: (1) the near-to-zero heuristic is able to recover most of the detected faults, and (2) the faults that cannot be recovered are the ones missed by the detection algorithm. Given the fact that 3-run recovery almost doubles the recomputation overhead and requires much complex hardware implementation, we recommend adopting 2-run recovery.

Overall, the results demonstrate the high effectiveness of our defense scheme in recovering single and double errors, at a cost that is 124X lower than TMR.

## 6 CONCLUSIONS

This work developed a defense mechanism against a targeted fault injection attack to improve the integrity of DNN hardware accelerators. Specifically, a permuted and selective recomputation scheme was proposed to harden the execution of the FC layer, which is the target of the fault injection attack. The defense scheme is frugal and can be easily implemented in hardware alongside the DNN accelerator. The experiments demonstrated the high effectiveness of the defense scheme, which achieved more than 90% detection rate of critical faults and recovered the model accuracy 90.5%+ to its original. The incurred runtime overhead is less than 1.61%, which is 62X lower than DMR and 124X lower than TMR. This frugal and effective defense scheme is well suited to mission-critical DNN applications with strict overhead and time predictability requirements.

## 7 ACKNOWLEDGMENTS

The work is partially supported by National Science Foundation under grant #1909854.

# REFERENCES


[1] Iljoo Baek, Wei Chen, Zhihao Zhu, Soheil Samii, and Ragunathan Raj Rajkumar. 2022. FT-DeepNets: Fault-Tolerant Convolutional Neural Networks with Kernel-based Duplication. In *2022 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*.

[2] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[3] Kamyar Givaki, Behzad Salami, Reza Hojabr, S. M. Reza Tayaranian, Ahmad Khonsari, Dara Rahmati, Saeid Gorgin, Adrian Cristal, and Osman S. Unsal. 2020. On the Resilience of Deep Learning for Reduced-voltage FPGAs. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[5] Le-Ha Hoang, Muhammad Abdullah Hanif, and Muhammad Shafique. 2020. FT-ClipAct: Resilience Analysis of Deep Neural Networks and Improving their Fault Tolerance using Clipped Activation. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[6] Fateme S. Hosseini, Qi Liu, Fanruo Meng, Chengmo Yang, and Wujie Wen. 2021. Safeguarding the Intelligence of Neural Networks with Built-in Light-weight Integrity MArks (LIMA). In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*.

[7] Fateme S. Hosseini, Fanruo Meng, Chengmo Yang, Wujie Wen, and Rosario Cammarota. 2021. Tolerating Defects in Low-Power Neural Network Accelerators Via Retraining-Free Weight Approximation. *ACM Trans. Embed. Comput. Syst.* (2021).

[8] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. 2019. Searching for MobileNetV3. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*.

[9] Suyong Lee, Insu Choi, and Joon-Sung Yang. 2022. Bipolar vector classifier for fault-tolerant deep neural networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*.

[10] Seo-Seok Lee and Joon-Sung Yang. 2022. Value-aware Parity Insertion ECC for Fault-tolerant Deep Neural Network. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[11] Yu Li, Yannan Liu, Min Li, Ye Tian, Bo Luo, and Qiang Xu. 2019. D2NN: a fine-grained dual modular redundancy framework for deep neural networks. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*.

[12] 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)*.

[13] Xiaoxuan Lou, Fan Zhang, Zheng Leong Chua, Zhenkai Liang, Yueqiang Cheng, and Yajin Zhou. 2019. Understanding Rowhammer Attacks through the Lens of a Unified Reference Framework. *ArXiv* (2019).

[14] Yukui Luo, Cheng Gongye, Yunsi Fei, and Xiaolin Xu. 2021. DeepStrike: Remotely-Guided Fault Injection Attacks on DNN Accelerator in Cloud-FPGA. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*.

[15] Dina G. Mahmoud, Vincent Lenders, and Mirjana Stojilović. 2023. Electrical-Level Attacks on CPUs, FPGAs, and GPUs: Survey and Implications in the Heterogeneous Era. *Comput. Surveys* (2023).

[16] Fanruo Meng, Fateme S. Hosseini, and Chengmo Yang. 2021. A Self-Test Framework for Detecting Fault-induced Accuracy Drop in Neural Network Accelerators. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[17] Fanruo Meng and Chengmo Yang. 2022. Exploring Image Selection for Self-Testing in Neural Network Accelerators. In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*.

[18] Jhon Ordoñez and Chengmo Yang. 2024. Derailed: Arbitrarily Controlling DNN Outputs with Targeted Fault Injection Attacks. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[19] Adnan Siraj Rakin, Zhezhi He, Jingtao Li, Fan Yao, Chaitali Chakrabarti, and Deliang Fan. 2022. T-BFA: Targeted Bit-Flip Adversarial Weight Attack. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022).

[20] Adnan Siraj Rakin, Li Yang, Jingtao Li, Fan Yao, Chaitali Chakrabarti, Yu Cao, Jae-sun Seo, and Deliang Fan. 2021. RA-BNN: Constructing Robust & Accurate Binary Neural Network to Simultaneously Defend Adversarial Bit-Flip Attack and Improve Accuracy. *ArXiv* (2021).

[21] Rihui Sun, Pefei Qiu, Yongqiang Lyu, Donsheng Wang, Jiang Dong, and Gang Qu. 2023. Lightning: Striking the Secure Isolation on GPU Clouds with Transient Hardware Faults. *ACM Trans. Des. Autom. Electron. Syst.* (2023).

[22] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[23] 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 (DAC)*.

[24] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. 2020. DeepHammer: Depleting the Intelligence of Deep Neural Networks through Targeted Chain of Bit Flips. In *29th USENIX Security Symposium (USENIX Security 20)*.

[25] Jinyu Zhan, Ruoxu Sun, Wei Jiang, Yucheng Jiang, Xunzhao Yin, and Cheng Zhuo. 2022. Improving Fault Tolerance for Reliable DNN Using Boundary-Aware Activation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41 (2022).