DeepShuffle: A Lightweight Defense Framework against Adversarial Fault Injection Attacks on Deep Neural Networks in Multi-Tenant Cloud-FPGA

Yukui Luo
Northeastern University
luo.yuk@northeastern.edu

Adnan Siraj Rakin
Binghamton University
arakin@binghamton.edu

Deliang Fan
Johns Hopkins University
dfan10@jhu.edu

Xiaolin Xu
Northeastern University
x.xu@northeastern.edu

***Abstract*—FPGA virtualization has garnered significant industry and academic interests as it aims to enable multi-tenant cloud systems that can accommodate multiple users' circuits on a single FPGA. Although this approach greatly enhances the efficiency of hardware resource utilization, it also introduces new security concerns. As a representative study, one state-of-the-art (SOTA) adversarial fault injection attack, named *Deep-Dup* [1], exemplifies the vulnerabilities of off-chip data communication within the multi-tenant cloud-FPGA system. Deep-Dup attacks successfully demonstrate the complete failure of a wide range of Deep Neural Networks (DNNs) in a black-box setup, by only injecting fault to extremely small amounts of sensitive weight data transmissions, which are identified through a powerful differential evolution searching algorithm. Such emerging adversarial fault injection attack reveals the urgency of effective defense methodology to protect DNN applications on the multi-tenant cloud-FPGA system.**

This paper, for the first time, presents a novel *moving-target-defense* (MTD) oriented defense framework *DeepShuffle*, which could effectively protect DNNs on multi-tenant cloud-FPGA against the SOTA Deep-Dup attack, through a novel lightweight model parameter shuffling methodology. DeepShuffle effectively counters the Deep-Dup attack by altering the weight transmission sequence, which effectively prevents adversaries from identifying security-critical model parameters from the repeatability of weight transmission during each inference round. Importantly, DeepShuffle represents a *training-free* DNN defense methodology, which makes constructive use of the typologies of DNN architectures to achieve being lightweight. Moreover, the deployment of DeepShuffle neither requires any hardware modification nor suffers from any performance degradation. We evaluate DeepShuffle on the SOTA open-source FPGA-DNN accelerator, Vertical Tensor Accelerator (VTA), which represents the practice of real-world FPGA-DNN system developers. We then evaluate the performance overhead of DeepShuffle and find it only consumes an additional $\sim$3% of the inference time compared to the unprotected baseline. DeepShuffle improves the robustness of various SOTA DNN architectures like VGG, ResNet, etc. against Deep-Dup by orders. It effectively reduces the efficacy of evolution searching-based adversarial fault injection attack close to random fault injection attack, e.g., on VGG-11, even after increasing the attacker's effort by 2.3$\times$, our defense shows a $\sim$93% improvement in accuracy, compared to the unprotected baseline.

***Index Terms*—Deep Neural Network, Security, Defense, Multi-tenant Cloud-FPGA**

1. Introduction

Public leasable cloud computing infrastructures have witnessed significant growth in recent years and become the main workhorse for accelerating diverse computation-intensive applications, such as Deep Neural Network (DNN) based AI. One of the main contributors to these high-performance cloud infrastructures is the underlying hardware accelerators, e.g., the graphic computing unit (GPU). These emerging cloud infrastructures can be configured to offer two types of services, i.e., platform as a service (PaaS) [2] and infrastructure as a service (IaaS) [3], which not only store and manage data but also enable the deployment of systems, computation tasks, and data processing in the cloud.

To fully exploit the benefits of cloud infrastructures, the leading cloud service vendors, like Amazon AWS [4] and Microsoft Azure [5], have adopted techniques in the following two directions: hardware virtualization and integration of high-efficiency hardware accelerators. Currently, these vendors have achieved virtualization of traditional hardware components, such as CPUs, GPUs, and DDR memories, by executing a virtual machine monitor (VMM), yet the 'Hypervisor' [6]. More recently, these cloud service vendors have started integrating new hardware accelerators, like field programmable gate array (FPGA), into their cloud servers, for the following reasons. Unlike the conventional hardware accelerators like CPUs and GPUs that optimize instruction streams for acceleration, FPGAs allow users to directly reconfigure the underlying gate-level hardware resources. Such flexible and direct control flows combined with customized hardware lead to significant performance and energy efficiency improvement. For example, AMD Xilinx's high-end AI Adaptive Compute Acceleration Platform (ACAP) card VCK5000 [7] achieves $\sim$1.8$\times$ frames per second per watt (FPS/W) compared to the Nvidia Ampere flagship GPU: A100 SXM [8] in a standard MLPerf [9] benchmark (Inference ImageNet with ResNet-50) [10].

Compared to the mature hardware virtualization technologies for CPU and GPU, FPGA virtualization in the cloud is an emerging topic and has received great attention in recent years, but is still in its infant stage. Most existing works in this domain have focused on enabling resource sharing on cloud-FPGA, i.e., for multi-tenancy. For example, in [11], Zha *et al.* proposed solutions to relax the tight coupling between FPGA applications compilation and resource allocations. Meanwhile, several other works target building end-to-end frameworks to enable multi-tenant cloud-FPGA, including AmorphOS [12] built on the AWS EC2 F1, Coyote [13] built on AMD-Xilinx Heterogeneous Accelerated Compute Clusters (HACC), and OPTIMUS [14] built with Intel Hardware Accelerator Research Program (HARP) platform.

Although encouraging, the current development of such multi-tenant cloud-FPGA has mainly focused on performance, leaving the security perspective largely under-explored. Specifically, the sharing of critical hardware resources, e.g., the power supply, enables "indirect" interaction between the circuit applications of different users. As a result, a number of recent works have demonstrated the vulnerabilities of multi-tenant cloud-FPGA associated with diverse applications.

In a recent USENIX Security'21 work [1], Rakin *et al.* presented an adversarial fault injection attack framework, *Deep-Dup*, which leverages the supply voltage fluctuations and a side-channel-guided attacking controller to inject well-timed faults at run-time to degrade the performance of DNN models executing in a cloud-FPGA context. The Deep-Dup attack successfully demonstrates complete fail of wide ranges of neural networks in a black-box setup (assuming no availability of neural network architecture, weight parameter values, training/test data, etc.), with only injecting fault to an extremely small amount (tens out of millions) of sensitive weight data transmission packages from external memory to on-chip buffer. Its high attack efficiency and efficacy are achieved through a powerful differential evolution searching algorithm that could leverage the repeatability property (i.e., fixed pattern) of weight parameter transmission during each inference round. Although compelling, the Deep-Dup work also has several critical limitations and future works:

(i) The victim DNN models used in [1] are primarily handcrafted at the register-transfer level (RTL), which do not accurately reflect the SOTA design practice, i.e., generating high-performance DNN models with contemporary FPGA-DNN compilation frameworks. In this paper, we opt for Versatile Tensor Accelerator (VTA) [15], an open-source, instruction-flow-driven DNN accelerator that is more suitably utilized in the cloud environment. VTA not only enables customization for IaaS but is also well-suited for Deep Learning as a Service (DLaaS), as this accelerator is highly optimized and supported by a mature software compiler: Apache TVM [16].

(ii) The underlying attacking principle and its impact on the DNN model behaviors is still unclear. For example, how do the injected faults propagate through the DNN model layer by layer and finally affect the final inference results? What are the impacts on the DNN feature space with adversarial fault injection? Investigating such questions is important in understanding the attacking principle, thus developing effective defense strategies.

(iii) Most importantly, to the best of our knowledge, there is still no effective systematic defense framework against such Deep-Dup like adversarial fault injection.

The followings are the main contributions of this work:

- We prototype a multi-tenant cloud-FPGA using the SOTA open-source acceleration framework and accelerator: Apache TVM [16] and VTA [15], to generate DNN model implementations, towards formulating generic study and evaluation platform. Using such SOTA experimental setup, we re-implement the end-to-end attack flow of Deep-Dup to re-calibrate its applicability on the VTA-generated DNN model deployment.
- We conduct thorough and detailed analyses of neural network model behavior under Deep-Dup generated adversarial fault injection. In particular, we focus on the input feature maps for both the clean model and the post-attack model. We visualize the layer-wise feature map changes to conduct fine-grained analysis. Through such studies, we discover that adversarial fault injection mainly introduces three types of changes to neural network models: brightness fluctuation, erasing the existing feature, and creating void features that do not exist.
- Gaining knowledge of the principle of Deep-Dup like adversarial fault injection on DNN model execution, we develop DeepShuffle, a *moving-target-defense* (MTD) oriented lightweight defense strategy. Specifically, DeepShuffle constructively utilizes the layer-wise data-dependency and convolution channel-wise independency to shuffle DNN model parameters at run-time, so as to break the foundation of these searching-based adversarial fault injection attacks, which mainly leverage the repeatability property (i.e., fixed pattern) of weight parameter transmission during each inference round to identify the most sensitive weight parameters. It thus significantly improves the robustness of a DNN model against adversarial fault injection attacks.
- We embed our proposed DeepShuffle framework in an end-to-end multi-tenant cloud-FPGA and conduct a thorough analysis of its working principles. We test the effectiveness of DeepShuffle against adversarial fault injection attacks on different vision datasets like CIFAR-10, ImageNet, and speech command recognition dataset, as well as with various DNN architectures (e.g., VGG, ResNet, and MobileNet), where Deep-Dup has already been successfully demonstrated. Our experimental results demonstrate that DeepShuffle can enhance the robustness of a wide range of DNNs. For example, for the VGG-11 implementation on VTA, even after increasing the attacker's effort by 2.3×, our defense shows a $\sim$60% (un-targeted attack) & $\sim$93% (targeted attack) improvement compared to the baseline in accuracy after an attack. To achieve this strong defense performance, DeepShuffle does not require any training stage overhead and also maintains the model performance (e.g., accuracy) for benign (i.e., no attack) operation.

- We validate the effectiveness of DeepShuffle in very extreme scenarios like black-box setup. Furthermore, we envision a strong attacker who even has the knowledge of the shuffling principles of DeepShuffle and strives to bypass its protection. Our experiments show that DeepShuffle can still provide sufficient defense capability against such extremely strong attacks.

The remainder of this paper is structured as follows. Sec. 2 reviews the background, related works, and the most commonly used FPGA-DNN acceleration framework, VTA, along with an explanation of the mechanism behind the Deep-Dup attack. Sec. 3 presents the threat model and recalibrates the performance of Deep-Dup concerning DNN applications executed on VTA. Sec. 4 delves into the principles underpinning DeepShuffle. This section provides examples of implementing channel shuffling in modern DNN architectures and demonstrates both the Deep-Dup attack and the protective effects of DeepShuffle within DNN layers. Sec. 5 evaluates the performance of DeepShuffle across various datasets and DNN architectures under both untargeted and targeted Deep-Dup attacks. Sec. 6 and Sec. 7 discuss the adaptive attack and defense set, future work, and present the conclusion.

2. Background and Related Works

2.1. Attacking DNNs in CPU and GPU

The security of DNNs has been challenged by many attacks, such as the adversarial example attacks that add imperceptible noise to the input data to manipulate the inference of DNN models [17], [18], [19]. As mitigation, a number of defense strategies [19], [20], [21] have been developed against such adversarial input attacks. These attacks and defenses are mostly developed from the algorithmic perspective, leaving the vulnerabilities associated with the hardware acceleration system under-explored.

Recently, a number of hardware-induced attacks have been proposed against the security of DNN models. Yao *et al.* [22] propose to inject faults on DNN models executed on a CPU, using the so-called Rowhammer attack [23] to flip the critical model parameters (e.g., the quantized weight values) stored in DRAM, so as to degrade the inference accuracy of DNN models. In [24], Yan *et al.* demonstrate a successful DNN model architecture extraction attack using the cache side-channels. Similarly, Zhu *et al.* [25] shows an attack that recovers a whole model by monitoring the plaintext packets transmitted over the PCIe bus of a GPU. Correspondingly, several defense mechanisms have been developed against the hardware-induced attacks [26], [27], [28], [29], [30]. For example, reducing the bit-width (e.g., binary [30]) of the weights of a DNN model is highly effective in mitigating bit error propagation across layers. In addition, [28], [29] have developed random DRAM row-swapping mechanisms to break the correlation between attacker and victim rows.

2.2. FPGA-DNN Acceleration Framework

Various acceleration frameworks have been proposed to enable the hardware-software co-design of DNN models to achieve high performance. Apache TVM [16] is a representative DNN acceleration framework, which supports diverse hardware platforms, such as CPU, GPU, and FPGA. Specifically, the Apache TVM framework offers a Versatile Tensor Accelerator (VTA) [31], an open-source framework to accelerate DNN models on FPGAs. To bridge the gap between popular front-end software programs like PyTorch and Tensor flow, the Apache TVM framework serves as a compiler to convert the DNN model into an instruction stream executed by VTA on FPGAs. It offers different quantization schemes from 32-bit floating point to 8-bit integers, graph optimization, and tensor optimization to generate VTA-executable instruction streams. These instruction streams can be optimized for different FPGA devices, and Apache TVM also provides corresponding optimization and auto-scheduling tools called Ansor [32] to explore the best model candidate for a specific FPGA device.

2.3. Multi-tenant Cloud-FPGA

The software-programmable devices like CPU or GPU can be easily managed by the hypervisor. In contrast, FPGAs enable users to directly reconfigure the underlying hardware resources, which presents a unique challenge for virtualization. Enabling multi-tenancy in the cloud-FPGA is an emerging topic that receives great attention from both industrial and research communities. In addition to the recent FPGA virtualization technologies presented in Sec. 1, there are also many industrial advancements, such as the Stacked Silicon Interconnect (SSI) technology from Xilinx [33], which integrates multiple dies into a single FPGA chip to provide ultra-high interconnect bandwidth, lower power consumption and $\frac{1}{5}$ the latency of standard I/Os between dies. Dynamic Function eXchange (DFX) technology [34] is another recent technology that facilitates the development of multi-tenant cloud-FPGA. It has been developed and integrated with the SSI technology in the high-end FPGAs like Xilinx Alveo U280 [35].

2.4. Attacking DNNs in Multi-tenant Cloud-FPGA

A malicious cloud-FPGA user can use power-hungry circuits [36], [37], e.g., a number of ring-oscillators [38], to suddenly increase the power demand and fluctuate the voltage supply. As a result, the circuit applications (of other users) will be possibly injected with faults. Leveraging such attacks, Krautter *et al.* [38] successfully extracted the key of advanced encryption standard (AES). In [39], Tian *et al.* demonstrate how to remotely extract the DNN architecture on a multi-tenant cloud-FPGA.

The main adversarial fault attack case study used in this paper, i.e., Deep-Dup [1], adopts similar attack schemes to inject faults on DNN models running on a multi-tenant cloud-FPGA. Deep-Dup takes full advantage of practical

layer-wise DNN model execution on an FPGA, i.e., there exist frequent data (model parameters) transmission between the on-chip and off-chip memories. Launching attacks during this transmission process causes the on-chip weight buffer to acquire and retain erroneous data, effectively implementing adversarial weight injection at the hardware level. Deep-Dup implements a complete black-box attack process. It applies an on-chip sensor like the Time-to-Digital Converter (TDC) to identify the computation and data transfer of the victim DNN inference application. Combined with the progressive differentiate evolution search (P-DES) algorithm, the Deep-Dup attack can profile and launch appropriate fault injections during each data transmission.

3. Threat Model and Baseline in this Work

3.1. Threat Model

Without loss of generality, we adopt the same threat model for multi-tenant cloud-FPGA used in related hardware security works [1], [36], [38], [39], as well as the FPGA virtualization works [11], [12], [13], [14]. The threat model has the following characteristics. There are multiple users who can run their circuit applications simultaneously on the same cloud-FPGA, with the assumption that the hypervisor of the cloud service is trustworthy. For security concerns, the circuits of different users are independent, i.e., their circuit applications are physically isolated. Each FPGA tenant has the ability to program their application as long as there is sufficient hardware resource. For example, the attacker can implement power-hungry circuits to inject faults into the victim DNN models. Following [1], we also assume that the attacker knows the type of data being transmitted between the on-chip and off-chip memories, e.g., either a DNN model or input data. Note this can be achieved using the TDC-based side-channel [1], [38], [39].

3.2. General Experimental Setup in This Work

FPGA-DNN Prototype: We build a multi-tenant FPGA prototype using a ZCU104 FPGA [40], for the following two reasons: (i) The used FPGA device ZCU104 has similar hardware configurations (e.g., ARM core and on/off-chip memories) to these high-end FPGAs in the commercial cloud like AWS F1 [41] and Alibaba Cloud F3 [42]; (ii) With a local-accessible prototype, we will be able to conduct fine-grained analysis, especially to control the timing of fault injections. In contrast to the existing works that mostly use a handcrafted DNN accelerator as the victim implementation [1], [43], we resort to the SOTA open-source VTA framework as the DNN accelerator. The most notable difference between these two setups is that VTA is a highly optimized FPGA-based accelerator offering several advanced features like parallel computing, enabling us to mimic real-world design practice.

Victim: In our setup, the victim utilizes the VTA as the FPGA-DNN acceleration framework. The victim may represent a user who deploys his/her own accelerators on the cloud-FPGA for inference acceleration, or a service provider offering deep learning as a service (DLaaS). As an instruction flow-driven DNN accelerator, VTA has specific weight-loading instructions to transfer the DNN model parameters from the onboard DRAM to the on-chip memory (BRAM), which creates *attacking opportunities* for a Deep-Dup attack.

Attacker: We assume the attacker is a malicious cloud-FPGA user whose circuit application is co-located with the victim user's, i.e., the VTA-based FPGA-DNN accelerator. Specifically, the attacker can use malicious power-hungry circuit to corrupt the inference flow of the DNN accelerator.

Un-targeted and Targeted Attacks: We define two attacking objectives following [1]: (i) Un-targeted attack that aims at degrading the overall inference accuracy of a DNN model, i.e., for all input classes; (2) Targeted attack that only causes misclassification of a specific (target) input class.

3.3. Baseline in This Work

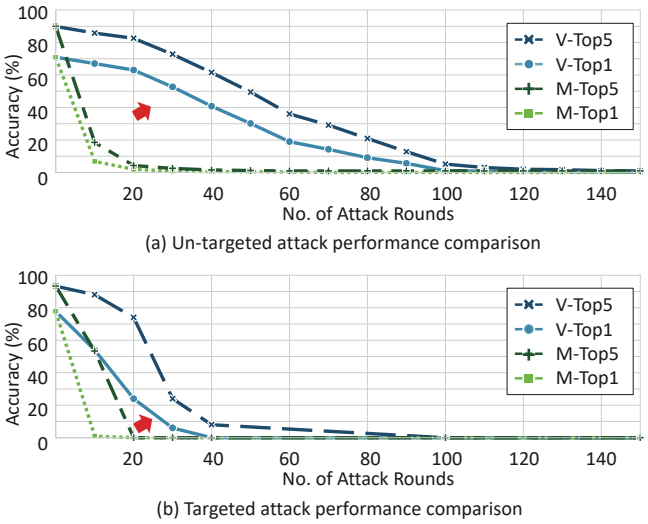


Figure 1. Deep-Dup attack on different accelerators. (a) Test accuracy degradation due to Un-targeted attack. (b) Targeted attack on class-269.

Since VTA has a highly optimized architecture, e.g., a GEMM core offering multiple input and output channels, which is significantly different from the previous handcrafted FPGA-DNN accelerators. Therefore, we first rebuild the baseline in this work, using VTA as the victim accelerator. To get comprehensive conclusions, we conducted both un-targeted and targeted attacks that uses the "grey wolf" of class-269 as a case study. Furthermore, we manually made DNN models for comparison. The attacking results for both DNN accelerators are illustrated in Fig. 1(a) and (b), respectively, in which we show the Top-5 and Top-1 inference performance for the manually ("M") generated DNN accelerator and VTA ("V"), under different attacking conditions. As shown in Fig. 1, VTA is more robust than the manually made DNN accelerators against adversarial fault injection attacks. The main reason is VTA executes the

1^{st} Conv2D layer and the last fully connected (FC) layer outside the FPGA hardware, i.e., on the ARM processor. In the remainder of this paper, we use the VTA attacking results as the baseline. Additionally, in Sec. 5.6, we conduct a fine-grained analysis to investigate the layer-wise impact of adversarial fault injections on VTA.

4. Our Proposed Defense: DeepShuffle

4.1. Revisiting Hardware-induced Adversarial Fault Injection Attacks on DNNs

Although most hardware-induced fault injection attacks on DNNs can be conducted in a random manner, i.e., do not follow certain rules, none of these SOTA advanced attacks adopts this strategy. The reason is simple, the fault injection attacks on DNNs mainly target manipulating the model parameters like weight values. Due to the ever-increasing size of modern DNN models, there exist a large number of such candidates, making random fault injection less efficient. For example, injecting faults to randomly selected model parameters may only introduce trivial performance loss, as demonstrated in [1], [22]. As a result, most SOTA fault injection attacks are conducted in an *adversarial* manner. Taking the DeepHammer attack [22] and Deep-Dup [1] as examples, they both employ searching framework to identify the most critical attacking opportunities. Similarly, other hardware-induced fault injection attacks, e.g., using laser beams [44], also require the attacker to focus on a specific area of the victim chip.

4.2. DeepShuffle: A Channel-Wise Shuffling Defense for DNN Models at Run-time

Gaining knowledge of these SOTA hardware-induced attacks on DNNs, we conclude that accurate profiling of the attacking opportunities associated with specific model parameters (e.g., layer or weights) is critical to make these attacks efficient and stealthy. Therefore, an efficient defense method is to invalidate such profiled attacking opportunities. To achieve this goal, we propose a lightweight defense framework DeepShuffle that adopts the so-called *moving target defense* (MTD) philosophy.

Although this defense philosophy is straightforward, it is nontrivial to deploy such a strategy in practice due to the strong data dependency between different DNN model layers, since any improper change in the model structure or parameters will render the inference result to be incorrect.

Key observation: We analyze the representative DNN model architectures shown in Fig. 2(a), and find that the 2D convolution (Conv2D) layer is the most fundamental building block. Typically, the inputs to such a layer consist of three parts:

(1) A 4-dimensional input feature maps (IN) with size (N, IC, H_i, W_i), where N denotes the batch size, represented by $n \in [0, N-1]$ for the n^{th} batch, IC is the input channel size represented by $ic \in [0, IC-1]$ for the ic^{th} input channel, and H_i and W_i stand for the height and width of the input feature map, respectively.

(2) A 4-dimensional kernel (W) with size (IC, OC, K_h, K_w), where each 2-dimensional plane with height (K_h) and width (K_w) is referred to as a filter. In other words, a kernel contains $IC \times OC$ filters, w OC is the output channel size, represented by $oc \in [0, OC-1]$ for the oc^{th} output channel.

(3) A 1-dimensional bias (b) with size OC. The output (O) of a Conv2D layer can be expressed as a 4-dimensional tensor with size (N, OC, H_o, W_o), where H_o and W_o depend on the padding method, K_h, K_w, and the stride size for the filter sliding during the execution of the cross-correlation operator ($\star$) [45], see Eq. 1.

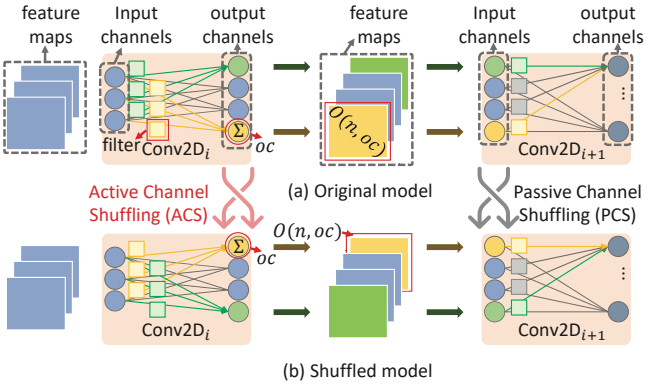


Figure 2. Illustration for Active and Passive Channel Shuffling. The green and yellow channels are exchanged from (a) to (b).

We can see from the original model in Fig. 2 (a), although different model layers have strong data dependency, such dependency mainly exists along the channel-wise. Specifically, the convolution result for output channel oc ($O(n, oc)$) in the Con2D layer i is the summation of cross-correlation operator results derived from the corresponding filters and input channel feature maps, as expressed in Eq. 1.

$$O(n, oc) = b(oc) + \sum_{ic=0}^{IC-1} W(oc, ic) \star IN(n, ic) \quad (1)$$

We highlight such a channel-wise data dependency with different colors in Fig. 2. For example, there are three filters connected to the masked output channel (oc) in yellow, each of them is cross-correlated with its corresponding input feature map and added together to produce the output $O(n, oc)$. From the model inference flow, $O(n, oc)$ bridges its two neighbour layers, by serving as the output of the i^{th} Conv2D layer and the input of the $(i+1)^{th}$ Conv2D layer.

Inspired by this observation, we propose a **training-free** defense strategy, *DeepShuffle*, which applies channel-wise shuffling to change the underlying DNN model topology while still maintaining the original layer-wise inference flow. Taking the illustrated DNN model in Fig. 2 (b) as an example, if we change the position of the oc channel within the 4-dimensional space, e.g., from Fig. 2 (a) to (b), the position of the output $O(n, oc)$ in the 4-dimensional output feature map will also change. Following these operations, we

can simply shuffle the input channels of the connected layer accordingly to ensure computational correctness, without introducing extra operators.

To clearly describe these channel-wise topology changes, we define two types of model shuffling: (1) Active channel shuffling (ACS) that is applied to an earlier (e.g., the i^{th}) Conv2D layer, which changes the connections between the input and output channels in that layer; and (2) Passive channel shuffling, which shuffles the $(i+1)^{th}$ Conv2D layer, to follow the channel-wise topology change in the i^{th} layer and ensure the inference correctness.

We summarize these basic operations into the following two model shuffling principles to guide the implementation of DeepShuffle:

- **Principle 1:** For a Conv2D layer i, if we shuffle its weight parameters along the output channel, the output feature maps of this layer will be shuffled accordingly.
- **Principle 2:** If we apply active channel shuffling (ACS) on a Conv2D layer i following a specific shuffling rule (SR_i), the same rule should also be applied to its connected layers in order to maintain computational correctness in the DNN inference without using any additional operators.

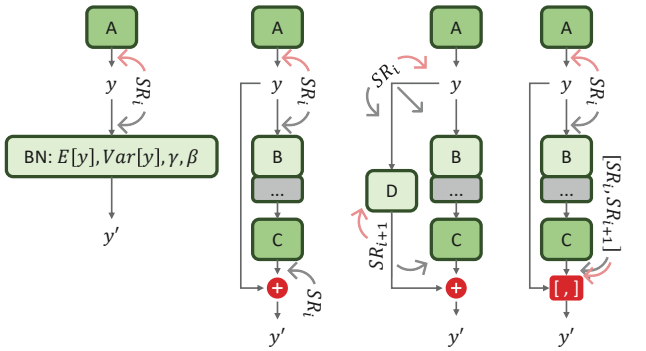
4.3. DeepShuffle: Adaption and Integration with Different DNN Model Blocks and Layers

How to apply DeepShuffle: As discussed in Sec. 4.1, a strong attacker can launch adversarial fault injections on the FPGA-DNN accelerator in a black-box setup, i.e., without knowing the detailed architecture and weight parameters of the victim model. Therefore, applying a static or constant model shuffling rule (SR) cannot fully defend against such attacks, since it only changes the victim model to another topology. Instead, we should create a large search for the SRs and keep updating them, so as to invalidate the profiled attacking opportunities by the adversary, in accordance with the moving target defense (MTD) philosophy.

Deploy DeepShuffle in Practice: We propose the following method to integrate with the VTA acceleration framework. Our proposed method makes constructive use of the data structure of this accelerator, without making any modification to its hardware implementations. Specifically, during the model loading process, VTA stores the weights in an array following the (IC, OC, K_h, K_w) format. We use DeepShuffle to read the lengths of IC and OC and generate a shuffling rule (SR), following which we perform *reordering* on the weights, along the IC and OC dimensions. Note that this reorder operation does not reload the weight array, but merely changes the order of the program pointers along the corresponding dimensions. Therefore, it is very lightweight, see Sec. 4.4.

Where to apply DeepShuffle: While the basic principles of DeepShuffle are straightforward, it is nontrivial to integrate them with different DNN model operators and blocks. We summarize the following SRs:

(1) *The input channel of the first layer in a DNN model should not be shuffled*, for the following two reasons: (a) Low efficiency. Taking a DNN model for image classification as an example, shuffling the input channels of its first layer will change the structure of the input feature map. As a result, all inputs must be pre-processed accordingly to fit the SR. (b) The input channel size of the first Conv2D layer is very limited. We scan the DNN models in the Model Zoo [46] and find they all have very small input channels, e.g., the input channel sizes for Gray, RGB, and CMYK images are simply 1, 3, and 4, respectively. As a result, the maximum applicable channel SRs is no more than 4!. Instead, shuffling the output channel of the first layer will significantly increase the number of applicable SRs. Taking a ResNet-18 DNN model trained for ImageNet as an example, its first Conv2D layer has 64 output channels. Therefore, if the shuffling is only applied on the output channel of this layer, we can have available SRs of 64!.



(a) Batch Norm (BN) (b) Residual block (c) Residual block w/ down sampling (d) Dense block

Figure 3. Shuffling examples in various convolutional-based architecture blocks: (a) Shuffling rule implemented on the Batch Normalization layer. (b) Shuffling rule distributed in the residual block. (c) Shuffling rule distribution in the residual block with a down-sampling convolution layer **D**. (d) Shuffling rule distribution within the dense block.

(2) *The output channels of the last layer (i.e., the one that generates logits output) should not be shuffled*. The number of output channels in the last layer is usually equal to the number of input classes, which are regulated in a fixed order. Therefore, if any SRs are applied, the user must also update the class labels (e.g., a lookup table), which is unnecessary since we can create a large search space by only shuffling the input channel of the last layer. Again we take a ResNet-18 model trained with the ImageNet as an example, the input features to its last fully-connection (FC) layer are generated by the output channel of the previous Conv2D layer of size 512, which can generate a huge enough SR search space of 512!.

Adapting DeepShuffle to modern DNNs: Modern DNN models have various specialized layers and complex connectivity patterns in addition to Conv, which requires special adaptions. We illustrate four representative layers in Fig. 3 and introduce how to adapt DeepShuffle to them each.

- **Batch normalization (BN) layer**. As illustrated in Fig. 3(a), a BN layer has a different set of four parameters defining its channels: running mean ($E[\boldsymbol{y}]$), running

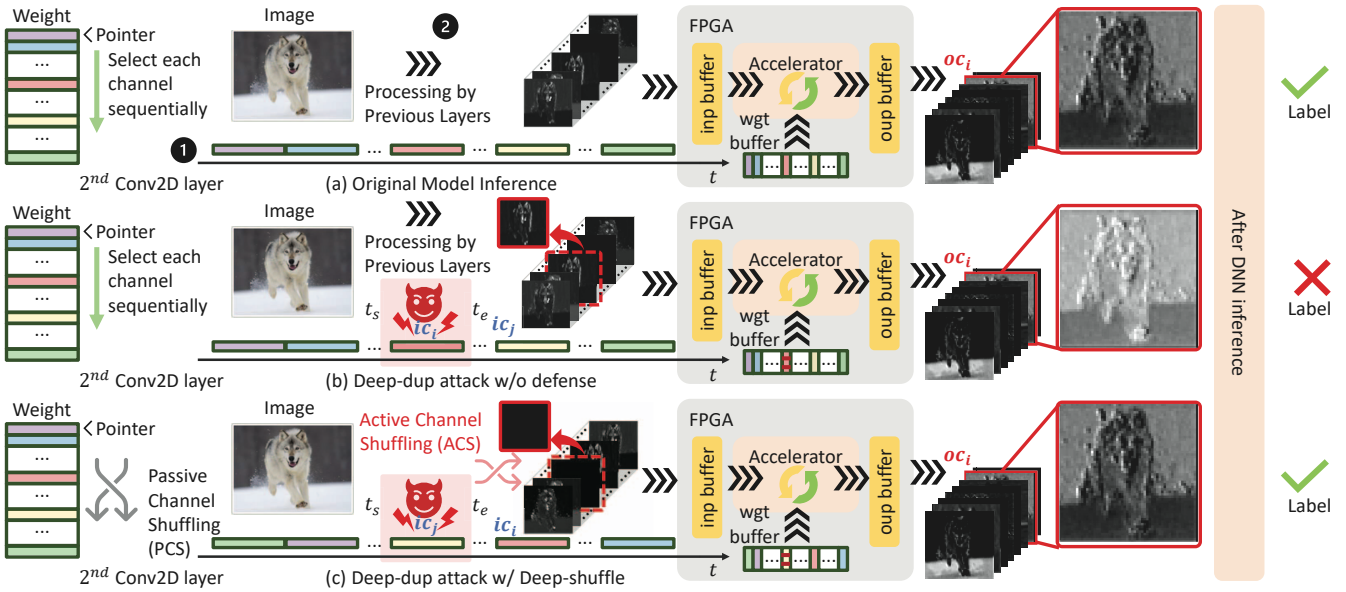


Figure 4. Visualization demonstrations of the 2^{nd} Conv2D layer in ResNet-18 on ImageNet, showcasing various inference environment scenarios. The layer's input and output comprise 64 channels each, and we examine these channels across all scenarios. (a) Normal inference scenarios. (b) Attacker targeting the weight transmission process, intelligently searching for attacking moments. (c) Application of DeepShuffle as a defense mechanism against the attack in (b).

variance ($Var[\boldsymbol{y}]$), weight (γ), and bias (β). Therefore, while applying DeepShuffle, we must shuffle all these four parameters accordingly, to fit the shuffling rule (SR) inherited from the layer A's output channels.

- **Residual block**. We consider two types of residual blocks used in *ResNet*. Specifically, one of them has a down-sampling convolution layer D on the cross-layer connection (Fig. 3(c)), while the other does not have (Fig. 3(b)). These two structures share a similarity, they both add the cross-layer connection to the output, i.e., layer C's output. Differently, the structure in Fig.3(b) applies the same shuffling rule (SR_i) to the output channels of layers A and C, and the input channel of layer B, while the structure shown in Fig. 3(c) applies SR_i to layer A's output channel and layers B and D's input channels. The SR_{i+1} is applied simultaneously to the output channels of layer D and layer C.
- **Dense block.** Fig. 3(d) illustrates the unique cross-layer connection of a DenseNet, which employs concatenation at the junction. In this structure, SR_i and SR_{i+1} must be combined. During concatenation, layer A's output comes first, followed by layer C's output. Since SR_i and SR_{i+1} represent channel indices, the latter needs to be added to the former's total count during the combination. For example, if layer A has three output channels with $SR_i = [3, 1, 2]$ and layer C also has three output channels with $SR_{i+1} = [2, 3, 1]$, the concatenation will yield six output channels. In this case, SR_i remains unchanged, while each element in SR_{i+1} will be increased by three as $[5, 6, 4]$. The new combined shuffling rule then becomes $[3, 1, 2, 5, 6, 4]$.

Considering the complexities of modern DNN layers and their interconnections, we formulate the third design principle, which aims to make DeepShuffle more generic for applications across a broader range of DNN architectures.

- **Principle 3:** In a DNN layer with multiple channels, it is essential to adapt layers with trainable parameters meticulously. In contrast, layers without trainable parameters, such as pooling and activation layers, do not necessitate any specialized design considerations.

4.4. DeepShuffle: An End-to-End Demonstration

We demonstrate the end-to-end workflow of DeepShuffle using a ResNet-18 model-based image classifier trained with ImageNet. To formulate generic conclusions, we conduct all experiments in this section with a "wolf" image. Meanwhile, we conduct adversarial fault injections on the original model (i.e., the one w/o protection) and the model protected by DeepShuffle, to investigate the fine-grained impacts of the adversarial fault injection attacks and DeepShuffle.

Original model: We take the 2^{nd} Conv2D layer of the ResNet-18 as an example to illustrate the data transmission and processing. The configuration of this layer is [IC, OC, K_h, K_w, H_i, W_i, H_o, W_o] = [64, 64, 3, 3, 56, 56, 56, 56]. Fig. 4 (a) displays how the original model processes data using FPGA accelerators in the following steps:

❶ The pointer of the DNN weight loading program points to the starting address of the layer weights stored in external memory (DDR4), and then transmits them sequentially to the on-chip weight buffer (BRAM) of the FPGA.

❷ The DNN program loads the input feature maps of this layer from external memory to the on-chip input buffer (BRAM) of the FPGA for inference.

Typical input of a DNN layer has input feature maps and weights. For certain layers, the size of input feature maps is much larger than the weights. In accelerator design, the weights of a layer are usually fully loaded onto the on-chip weight buffer. However, the size of the on-chip input feature buffer generally cannot accommodate all input feature maps. Therefore, DNN accelerators usually perform block-wise computations for each layer, as detailed in VTA's blocking strategy [15]. Therefore, we can observe that the weight parameters do not change after being loaded onto the on-chip weight buffer until all computations in this layer are completed. This is also the root reason an attacker can apply black-box attacks in Deep-Dup [1].

Model under attack: We deploy the fault injection attack in [1] on the 2^{nd} layer, assuming the weights of channel ic_i are transmitted during the time interval from t_s to t_e. The results are shown in Fig. 4 (b), from which we observe a largely changed output feature map, compared to Fig. 4 (a). Note that such changes will spread and be amplified in all subsequent layers, ultimately leading to an incorrect result.

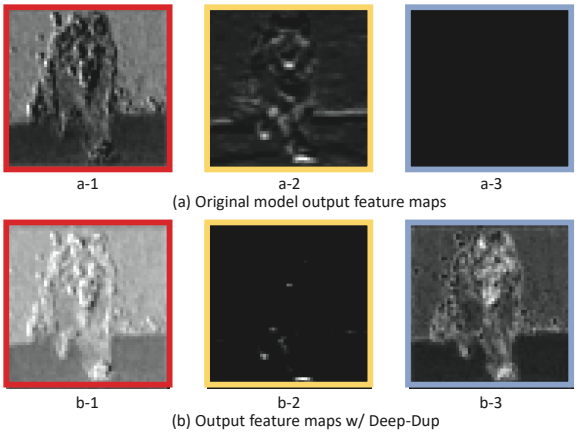


Figure 5. Affect examples.

We find three major changes in the feature map changes caused by fault injections, as illustrated in Fig. 5: (1) Brightness fluctuation (a-1 and b-1); (2) Erasing existing features (a-2 and b-2); (3) Creating void features that do not exist (a-3 and b-3). These three changes are fatal and will directly cause the inference of the input data to deviate from the preset path. We furthermore evaluate the post-attack impacts on the 2^{nd} Conv2D layer for all 64 output feature maps. The results are shown in Fig. 10 in Appendix A.

Model under DeepShuffle protection: We adopt the same attacking patterns from Fig. 4 (b) to evaluate DeepShuffle. Simply, we apply random channel shuffling to the model, which swaps the transmission order of input channels ic_i and ic_j. As a result, the input features and weights of both channels will be exchanged simultaneously in the transmission order, as illustrated in Fig. 4 (c). If the input feature of the ic_j channel contains little or no information, the attack at the original time period will be completely ineffective. The experimental results are shown in Fig. 4 (c), from which we observe that the oc_i output is the same as that of the original model in Fig. 4 (a).

4.5. DeepShuffle against Adversarial Fault Injection in Black-Box Setup

We note that trivial or static shuffling is not always sufficient for the following reasons.

(1) The adversary can always increase the number of fault injections, making any defense less efficient. However, such attacking practice will also render it less or not stealthy; thus we do not consider it in this work.

(2) As discussed in Sec. 4.1, advanced searching algorithms significantly help the fault injection attacks, which may even learn the shuffling rules of DeepShuffle and bypass its protection, especially for attacks like Deep-Dup that can be conducted in a black-box setup.

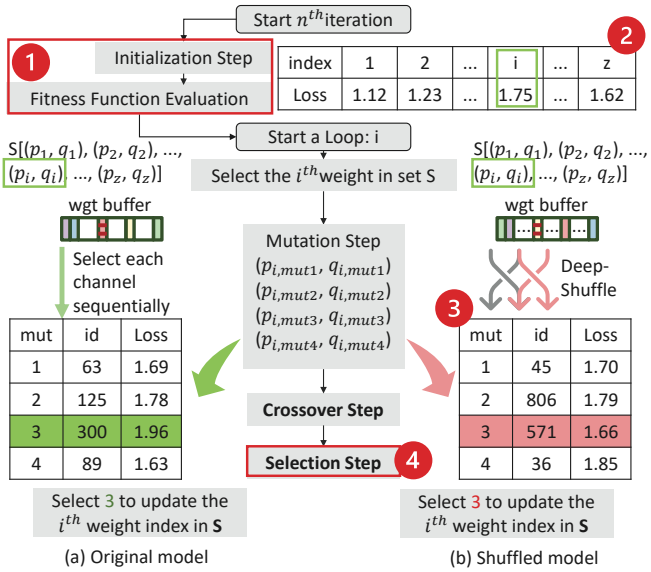


Figure 6. The impact of DeepShuffle on the P-DES.

We take the Progressive Differential Evolutionary Search (P-DES) used in Deep-Dup [1] as an example and analyze how the DeepShuffle defense could invalidate such advanced attacks in a black-box setting. We illustrate the mechanism of P-DES in Fig. 6 (a), which involves the following parameters: (1) Fitness function that evaluates the attack objective at a given attack iteration. (2) The number of evolution z, which is selected during the initialization step, as shown in Fig. 6. The more z values, the higher the probability of finding a more suitable weight candidate to be attacked. Deep-Dup used $z = 500$ in their experiments. However, we found that neither z nor the No. of Attack Rounds alone can adequately represent the difficulty of deploying a Deep-Dup on hardware. This is because before deploying Deep-Dup, the attacker needs to conduct profiling. Therefore, we define a new metric: "Deep-Dup profiling inferences" (see Eq. 2). (3) Mutation functions are used to generate new

attack candidates that can directly affect the performance of the search algorithm. For example, P-DES incorporated 4 mutation methods.

As illustrated in Fig. 6, DeepShuffle can potentially affect the winner selection of P-DES at two stages. The first step of the search algorithm is to generate a set of the initial population with z size and evaluate their loss function shown in Fig. 6 (a). As shown in (1), during the initialization step, P-DES generates z attack configurations, i.e., referred as **candidate**. Then, P-DES performs z attacks one at a time and calls inferences to evaluate the candidate, i.e., calculating the loss. During this process, DeepShuffle will randomly shuffle the sequences of these weights, and thus the loss profile (2) loses its reference index due to shuffling. For example, consider the above example in Fig. 6 (a), where the highest loss value can be achieved by the i^{th} index. Ideally, with no shuffle i^{th} index is the winning candidate among the current population. But after applying DeepShuffle, the winning candidate in the current iteration will no longer be at the i^{th} position, completely disrupting the attacker's reference point even before starting the evolutionary search. Note that in our evaluation, we apply DeepShuffle for every inference call making it even harder for the search algorithm.

In the next step, P-DES applies mutation to each candidate. As shown in Fig. 6 (b), four mutation strategies are adopted for each candidate i. P-DES evaluates these four strategies and decides whether to update the attack configuration corresponding to the candidate i. In Fig. 6 (b), we show that, without the influence of DeepShuffle, P-DES uses the 3^{rd} mutation strategy with an id of 300 to update the current i^{th} candidate in this mutation step. However, when DeepShuffle applies, the mutation indexes reference gets shuffled, then P-DES will be "fooled" to select the same reference mutation index, i.e., the 3^{rd} mutation but a completely different candidate index of 507, as in (3). It will result in selecting the wrong mutation strategy 3^{rd} one instead of the 4^{th} mutation, resulting in a weaker attack, i.e., a lower increment in loss value.

In summary, DeepShuffle injects randomness into the P-DES process when evaluating, mutating, and updating candidates. This ultimately affects the selection step (4) and results in selecting the wrong winner candidate. As a result, we expect the performance of Deep-Dup to deteriorate close to the level of random attacks.

5. Experimental Analysis

5.1. Experimental Setup

Dataset and DNN Models: We employ popular datasets covering a wide range of models, including CIFAR-10 [47] and ImageNet [48], which have different input image sizes and numbers of classes. CIFAR-10 consists of images with an input size of 32×32 and 10 classes, while ImageNet consists of images with an input size of 224×224 and 1,000 classes. We also exhibit the performance of DeepShuffle across a wide range of DNNs architectures to demonstrate its compatibility. For CIFAR-10, we conduct inference on ResNet-20 [49] and VGG-11 [50]. For ImageNet, we apply ResNet-18, ResNet-50 [49], and MobileNet-V2 [51]. Our evaluation model architecture follows the same setting as the prior Deep-Dup attack [1]. In addition, we have added a new evaluation dataset using Google's speech command dataset [52] in the Appendix.

VTA Hardware Configuration: We configure the VTA hardware as follows.

1) VTA uses 8-bit integer data types for both input and parameters. The parallel computation module has a 16×16 GEMM kernel, meaning that both input and output channels have a parallelism of 16.
2) The buffer size for input and output is 32 KiB, while the parameter buffer size is 256 KiB. The internal result buffer size reserved for the multiply-accumulate (MAC) operator is 128 KiB.
3) After testing, we found that the maximum executing frequency (F_{max}) of VTA on ZCU104 is 450MHz. We set the VTA executing frequency to 400MHz, to ensure an adequate frequency margin of $\sim$10%.

General Deep-Dup Configuration: We test the **un-targeted attack** on CIFAR-10 and ImageNet, as shown in Sec. 5.2, and perform **targeted attack** on CIFAR-10, shown in Sec. 5.3. We randomly selected 128 & 64 images from the test dataset for CIFAR-10 and ImageNet to implement un-targeted attacks, respectively. For the targeted attack, we choose a specific class, "ship" (class-8), and then randomly select 128 images from the targeted class for implementation. Therefore, for these attacks and if the batch size for each inference is 1, the attacker needs to perform 128 or 64 inference steps for CIFAR-10's un-targeted and targeted Deep-Dup, and ImageNet's un-targeted Deep-Dup, during the fitness function evaluation.

Evaluation Metric and Hyper-parameters: We utilize **Test Accuracy (TA)** as the primary performance metric. TA represents the percentage of samples accurately classified by the network. **Post-Attack TA** refers to the test accuracy measured after the attack. Moreover, we report the **No. of Attack Rounds** required to achieve the attacker's desired test accuracy degradation as an evaluation metric for attack efficacy. Since the increased number of attacks implies that the attacker requires more resources (e.g., time, memory resources) and requires changing a significant portion of the model, we consider increasing attack rounds as a metric to evaluate our defense success.

To compare the clean model with the Deep-Dup affected model, we measure the similarity of specific layer output feature maps using **Feature Similarity Index Method (FSIM)** [53]. FSIM provides a similarity score between 0 and 1, where 1 represents two parts in comparison are the same, and 0 represents they are completely different. We introduce **Deep-Dup profiling inferences (DPI)** in Eq. 2, a new metric to measure the complexity of deploying the Deep-Dup attack and its overhead. DPI is based on several factors, including the number of evolutions (z), mutation

TABLE 1. SUMMARY OF UN-TARGETED ATTACK ACROSS MULTIPLE DNN ARCHITECTURE AND DATASETS.

Dataset	Model	Top-1 TA (%)	Baseline (No Defense)				DeepShuffle (Proposed Defense)			
			Top-1 Post-Attack TA (%)	No. of Attack Rounds	DPI (Attack Overhead)	Weight Change (%)	Top- 1 Post-Attack TA (%)	No. of Attack Rounds	DPI (Attack Overhead)	Weight Change (%)
CIFAR-10	VGG-11	90.13	11.98	241	616,960	0.0002	72.8	1,000	2,560,000	0.0008
	ResNet-20	90.79	11.8	68	174,080	0.025	11.8	456	1,167,360	0.0168
ImageNet	ResNet-18	69.5	0.22	213	272,640	0.00194	6.84	1,000	1,280,000	0.0091
	ResNet-50	75.78	0.52	374	478,720	0.0016	60.7	1,000	1,280,000	0.0043
	MobileNet-V2	71.13	0.13	2	2,560	0.00009	1.88	20	25,600	0.0009

numbers (M), the number of attack rounds (AR), the number of samples (S) input for attack deployment, and the input batch size (B_s), which represents how many samples are in one batch, we adopt $B_s = 1$ for all experiments.

$$DPI = z \times M \times AR \times \frac{S}{B_s} \tag{2}$$

DPI reflects the amount of inference calls the attackers make to achieve the desired objective. Since an increasing number of inference calls is associated with increased attack time and hardware resources, it is an ideal metric to measure the impact of our defense on attack overhead. A higher DPI value indicates that the attacker requires more time, more hardware resources and sacrifices attack stealthiness.

Similar to un-targeted Deep-Dup [1], we consider reducing the Post-Attack TA of the target model to the level of random guess as a successful attack. Hence, as an evaluation metric, we report the number of attack rounds required to degrade the model accuracy to a random guess level. For example, for the CIFAR-10 dataset with 10 classes, Post-Attack TA of $\sim$10% is considered a random guess accuracy. However, for ImageNet, due to its large scale with 1000 classes, the Post-Attack accuracy below $\sim$1% oscillates even after increasing attack iteration. Hence, we relax the condition by reporting the attack iteration required to reach below $\sim$1% since, at this point, the model is entirely malfunctioning, and the attack practically succeeded.

5.2. Evaluation of DeepShuffle against Un-targeted Deep-Dup Attack

We evaluate our proposed defense performance against the un-targeted Deep-Dup attack across multiple DNN architectures and vision datasets, following the same approach as Deep-Dup paper [1] presented, summarized in Tab. 1.

Setup: Attack Evolution z = 5; Max No. of Attack Rounds: 1,000; Shuffle frequency: every inference.

CIFAR-10 Evaluation. As shown in Tab. 1, the baseline models are incredibly vulnerable to Deep-Dup attack [1]. For example, the VGG-11 and ResNet-20 require just 241 and 68 rounds of attack, respectively, to completely fail, i.e., close to 10 % test accuracy/random guess case for a 10 class classification. In contrast, once equipped with DeepShuffle, both model shows significantly improved robustness. In particular, for VGG-11, our proposed defense can overcome the impact of Deep-Dup attack. It shows that even after 1,000 rounds of attack, the attack fails to degrade the model accuracy below 72 %. For ResNet-20, again DeepShuffle increases its robustness by requiring $\sim$6.7$\times$ more attack rounds to achieve similar accuracy degradation.

The above observation concludes that the proposed DeepShuffle can better resist the Deep-Dup attack for dense models such as VGG-11. However, protecting compact models (e.g., ResNet-20) using DeepShuffle is more challenging. This conclusion is similar to prior works [1], [54] as they have also confirmed the vulnerability of compact models against weight perturbation. Nevertheless, our DeepShuffle can still significantly increase the attacker's overhead (i.e., high DPI) and resist the Deep-Dup attack even on compact model architecture.

ImageNet Evaluation. Our evaluation of the DeepShuffle defense on the ImageNet dataset is shown in Tab. 1. Again, our test case model architectures ResNet-18, ResNet-50 & MobileNet-v2 are extremely vulnerable to Deep-Dup attack. For the residual models, compact ResNet-18 is more vulnerable (i.e., 213 attack rounds) than the larger ResNet-50 (i.e., 374 attack rounds). However, once our proposed defense was incorporated into the inference stage, it can defend the attack successfully. We observe that even after 1,000 rounds of attack for both models, the attack fails to reach the same level of accuracy degradation as the baseline model (i.e., no defense) when DeepShuffle protects the model.

For the compact model, MobileNet-v2 shows lower resistance to Deep-Dup and only needs two rounds of attack, to reduce its Post-Attack Accuracy to a random guess level. However, after applying DeepShuffle, the model's robustness is improved by 10$\times$. While DeepShuffle has increased robustness against the Deep-Dup attack on MobileNet-v2, it is not as effective as the other models. Note that we evaluate DeepShuffle on MobileNet-v2 to make a fair defense evaluation with the baseline Deep-Dup attack, which also has reported the extreme vulnerability of MobileNet-v2. However, MobileNet-v2 is a unique DNN architecture in our test, designed to be lightweight and efficient, i.e., it is naturally more vulnerable to Deep-Dup. Even so, our propose DeepShuffle still enhances its robustness, by 10x.

Based on these results, we draw the following conclusions: DeepShuffle can defend the Deep-Dup attack on most dense model architectures (e.g., ResNet-18, ResNet-50, VGG-11), as increasing the attack complexity near 1,000 makes the attack practically very expensive (i.e., high DPI). However, some specially optimized models are more vulnerable to Deep-Dup because of their compact nature and architectural design. Even though DeepShuffle can not fully defend Deep-Dup on compact models, we still observed a large (e.g., $\sim$10$\times$) improvement in robustness after applying

TABLE 2. TARGETED ATTACK ON CLASS-8 (SHIP CLASS) OF CIFAR-10 DATASET.

Model	Top-1 TA (%)	Target Class Top-1 TA (%)	Baseline (No Defense)				DeepShuffle (Proposed Defense)			
			No. of Attack Rounds	DPI (Attack Overhead)	Target Class Post-Attack TA (%)	Post-Attack Model TA (%)	No. of Attack Rounds	DPI (Attack Overhead)	Target Class Post- Attack TA (%)	Post-Attack Model TA (%)
VGG-11	90.17	93.2	434	4,444,160	1.80	51.63	1,000	10,240,000	95.00	88.58
ResNet-20	90.79	93.0	188	1,925,120	1.88	27.5	1,000	10,240,000	69.75	72.6

our DeepShuffle.

5.3. Evaluation of DeepShuffle Against Trageted Deep-Dup Attack

Setup: Attack Evolution z = 5; Max No. of Attack Rounds: 1,000; Shuffle frequency: every inference.

We examine the performance of DeepShuffle in targeted attacks using CIFAR-10, see Tab. 2. We test targeted attacks aimed at class-8 (i.e., Ship) on VGG-11 and ResNet-20 models. Post-Attack Target Class TA represents the probability of correctly predicting the target class samples after an attack. And Post-Attack Model TA represents the impact of targeted attacks on the overall model inference Top-1 accuracy. The targeted Deep-Dup attack is very threatening, which can successfully miss-classify most target class samples (i.e., $\sim$1% target class accuracy after an attack). However, after applying DeepShuffle as a defense, even after 1000 rounds of attack, the model performance on VGG-11 does not deteriorate, with a Post-Attack target class TA holding at 95% and an overall Top-1 Post-Attack TA holding at 88.58%, ultimately eliminating the efficacy of Deep-Dup attack. On top of that, even for a compact vulnerable Resnet-20 model, DeepShuffle resists the targeted attack well, by not allowing the attacker to degrade the target class accuracy below 72%, even after 1,000 rounds of attack. In conclusion, our proposed DeepShuffle successfully eliminates the risk of a Deep-Dup attack by resisting it after many attack iterations, increasing attack complexity and overhead as highlighted in the DPI (proportional to attack overhead) column of Tab. 2.

5.4. Observation of Attack Evolution with Increasing Attack Rounds

Setup: Attack Evolution z = 5; Max No. of Attack Rounds: 400; Shuffle frequency: every inference.

We demonstrate the performance of DeepShuffle on ResNet-18 on the ImageNet dataset by gradually increasing the attack rounds and reporting the accuracy degradation at each attack step. Similar to Fig. 1, the clean (i.e., original) model Top-1 and Top-5 TA are 69.5% and 89.92%, respectively. After attacking the model using Deep-Dup for 200 rounds, the Post-Attack Top-1 and Top-5 accuracy drop to 0.22% and 0.84%. Next, we exhibit the performance of DeepShuffle. As shown in Fig. 7, with the protection of DeepShuffle, the Post-Attack Top-1 and Top-5 accuracy remain around $\sim$35% and $\sim$60% respectively, after approximately 400 attack rounds.

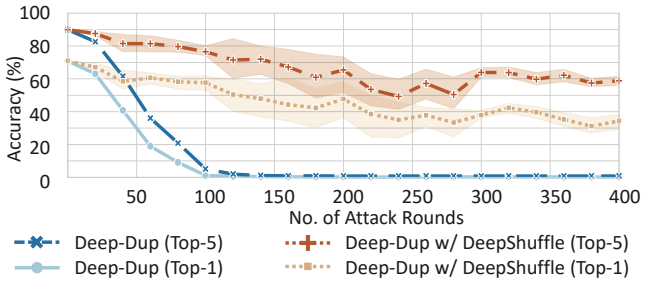


Figure 7. DeepShuffle applied on ResNet-18 with ImageNet dataset.

In this experiment of Fig. 7, we randomly shuffle the channel order of the model for every inference on ResNet-18 to infer the ImageNet dataset. According to the architecture of ResNet-18 shown in Fig. 1, 14 shuffling rules (SRs) are used, and the searching space for all SRs on ResNet-18 is $2 \times 64! + 4 \times 128! + 4 \times 256! + 4 \times 512!$. On the attacker's side, we set the parameters for the Deep-Dup attack as z = 5, with four mutation strategies (M = 4) and a maximum of 400 attack rounds.

To observe the statistical patterns of DeepShuffle, we conduct 20 independent experiments, starting from a clean model, and apply the P-DES algorithm in the Deep-Dup attack by automatically selecting and optimizing the attacked DNN layer and the attack moments of the weight transmission. We plot the standard deviation and mean of TA in Fig. 7. A relatively more significant variance exists during 100 to 300 attack rounds, and the DeepShuffle protection converges after the attack rounds increase beyond 300. This experiment indicates that DeepShuffle fiercely resists Deep-Dup throughout the attack evolution stages. Compared with the random attack degradation line, it is clear that DeepShuffle mitigates the effect of P-DES on Deep-Dup attacks. In summary, DeepShuffle offers improved resistance against the Deep-Dup attack by maintaining a clear improvement margin throughout the attack evolution.

5.5. Feature Similarity Analysis on the Output of a Single Layer

To further demonstrate the Deep-Dup attack and the effectiveness of DeepShuffle defense, we select attacks explicitly targeting the 2^{nd} Conv2D layer of a ResNet-18 model. We report the defense performance by only protecting the 2^{nd} Conv2D layer in Fig. 8. Note that this experiment is explicitly for fine-grained analysis purpose, as the Deep-Dup attack could not accurately manipulate a specific layer, in a black-box setup. In this experiment, we analyze the

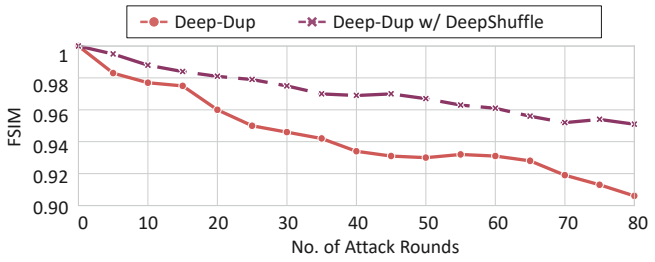


Figure 8. Feature-based similarity index (FSIM) with the number of attacks: output feature maps collected from the 2^{nd} Conv2D layer.

similarity between the feature maps obtained from the 64 output channels of this layer and the clean model's feature map from the same layer. We use the feature-based similarity indexing method (FSIM) to measure the feature similarity. We expect that the similarity of the features for two different scenarios, i.e., without or with defense will demonstrate how proposed DeepShuffle is minimizing the error propagation resulting from the attack. We observe that in a DNN model without the protection of DeepShuffle, the overall FSIM of the 64 channels decreased to around ∼0.9 after nearly 80 rounds of attack. In contrast, under the protection of DeepShuffle, the similarity of the layer is maintained at around 0.95. A larger score indicates that the proposed DeepShuffle prevents significant change in the output values of a given layer compared to the baseline (Deep-Dup) attack. Hence, the model prediction result and accuracy should hold better using our defense strategy.

5.6. Layer-Wise Profiling of DeepShuffle

Setup: Attack Evolution z = 5; Max No. of Attack Rounds: 400; Shuffle frequency: every inference.

We profile the effect of DeepShuffle for each model layer by independently attacking one layer's weight and recording the accuracy impact for preset attack rounds (e.g., 400). The results are shown in Fig. 9, which demonstrate the attack effects of the Deep-Dup attack deployed separately on i^{th} Conv2D layer of ResNet-20, where $i \in [1, 19]$. Fig. 9 (a) demonstrates the **No. of Attack Rounds** to reach random-guess accuracy or maximum preset number (e.g., 400), and Fig. 9 (b) demonstrates the inference accuracy after the attack, a.k.a **Post-Attack TA**. For example, when the Deep-Dup attack is launched on the 1^{st} Conv2D layer, after 74 attack rounds, the inference accuracy drops from 90.79% to 11.74%. In contrast, even after 400 rounds of attack, the attack fails to degrade the accuracy below 20% for the same layer with our defense.

In Fig. 9 (a), it is evident that the attacker can cause a drop in the Post-Attack accuracy to $\sim 10\%$ for the inference, regardless of the layer targeted with the Deep-Dup attack. In ResNet-20, the last (i.e., the 19^{th}) layer and layer 7 are particularly vulnerable and only need $\sim$ 30 attack rounds to achieve the random guessing goal. The 10^{th} layer, which is the middle layer, is the most robust; after 400 attack rounds, the Post-Attack accuracy is 14%. After applying

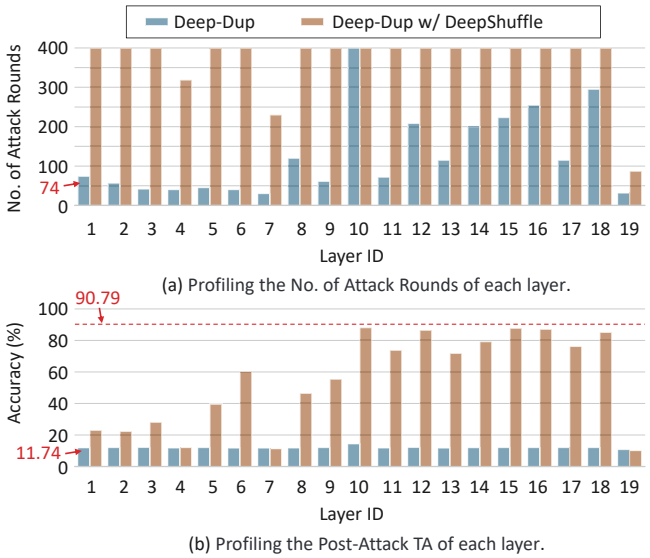


Figure 9. Deep-Dup effects on each layer individually. (a) shows the results of the Deep-Dup attack w/o any protection. (b) shows the results of the Deep-Dup attack met the DeepShuffle protection scheme.

our proposed DeepShuffle, we found that the robustness properties of the layers themselves affect the effectiveness of DeepShuffle. Still, it can consistently improve the robustness across all the layers. For example, layers 10-18 require more than 200 attack rounds to achieve $\sim 10\%$ of Post-Attack accuracy. However, our proposed DeepShuffle defends the attack by holding accuracy at or above 60 % even after 400 rounds of attack at these noise-resilient layers. Our prior hypothesis supports this observation that layers 10-18 have many parameters, and we expect dense layers to be more resilient to noise than compact layers (e.g., 1-4 & 19). Hence, the protection of DeepShuffle is amplified for dense layers compared to compact ones. These experimental results are expected and also discussed in prior works [18], [54], that noise added to the early layers has the potential to be multiplied/amplified as it propagates through the model.

5.7. The Performance Overhead of DeepShuffle

Setup: Shuffle frequency: every inference.

We test four ResNet architectures with increasing depth, namely ResNet-18, 34, 50, 101, on ImageNet. It is worth noting that ResNet-18 and 34 have a direct scaling relationship, while ResNet-50 and 101 also have a direct scaling relationship. Therefore, in the experiments, we only select ResNet-18 and 50 for experiment. Here, we mainly explore the scalability of model depth and DeepShuffle overhead.

In this experiment, we still apply DeepShuffle for each inference because this is the worst-case performance overhead. Since we only apply channel-wise shuffling, the data in the external memory (DDR) buffer did not need to be refreshed, which greatly increases the feasibility of DeepShuffle. We only need to change the channel-wise pointer order from external memory to the FPGA on-chip buffer during

TABLE 3. THE PERFORMANCE OVERHEAD AND SCALABILITY WHILE DNN IS IMPLEMENTED ON VTA

	ResNet Model Architectures			
	ResNet-18	ResNet-34	ResNet-50	ResNet-101
No. of Conv2D	19	35	51	102
Inference Time (ms)	500	900	1300	2400
DeepShuffle Time(ms)	12.1	23.4	33.6	67.9
Overhead Ratio(%)	+2.42	+2.6	+2.58	+2.83

weight transmission, and most of the consumption come from the time spent in generating all the shuffling layer pointer orders. As shown in Tab. 3, we find that this time consumption is linear and scalable because the time required to generate SRs increases proportionally with the depth of the DNN architecture. Overall, the performance overhead of DeepShuffle is $\sim 2.5\times$ of the inference time consumed on VTA. For DNNs running on VTA, the overhead of DeepShuffle is negligible. As this paper focuses on the effectiveness and scalability of DeepShuffle, we do not consider the performance of DeepShuffle on faster accelerators but consider this as future work.

6. Adaptive Attack & Defense

6.1. DeepShuffle against Strong Attackers

Setup: Attack Evolution z is a variable; Max No. of Attack Rounds: Unlimited; Shuffle frequency: every inference.

In the following experiments, we assume the attacker knows a defense strategy is being deployed to protect against Deep-Dup. To improve the efficiency of Deep-Dup given a knowledgeable attacker, they may increase the Deep-Dup evolution number z to overcome the protection of DeepShuffle. Since the Deep-Dup attack's fundamental principle relies on the evolutionary search algorithm, thus, an adaptive attacker's only tool to bypass randomness in the search step is increasing the attack evolution z. Tab. 4 shows the correlation between z and the required attack rounds to nullify DeepShuffle. We observe that by increasing the attack evolution z the attacker can reduce the number of attack rounds to achieve the same level of accuracy degradation as the baseline case. However, increasing the value of z again comes at the cost of attacking overhead (i.e., DPI). Hence, even though an adaptive attacker may achieve a similar attack success rate with lower attack rounds by increasing the value of z; it will still cost the same overhead due to the increasing number of inference calls. In conclusion, DeepShuffle can increase the attacker's effort and overhead even after considering a white-box threat model, where a knowledgeable attacker knows the principle of our defense.

6.2. Effect of Shuffling Frequency on DeepShuffle

Setup: Attack Evolution z = 20; Max No. of Attack Rounds: Unlimited; Shuffle frequency: variable.

TABLE 4. THE IMPACT OF ATTACK EVOLUTION z ON THE DEEPSHUFFLE.

	Baseline (No Defense)			**DeepShuffle (Proposed Defense)**		
z	Post-Attack TA (%)	No. of Attack Rounds	DPI	Post-Attack TA (%)	No. of Attack Rounds	DPI
5	11.58	68	**174,080**	11.8	456	**1,167,360**
20	11.86	16	**163,840**	11.8	193	**1,976,320**
100	10.85	11	563,200	11.93	112	5,734,400
500	10.52	7	1,792,000	11.55	106	27,136,000

TABLE 5. EFFECT OF VARYING SHUFFLING FREQUENCY ON THE DEFENSIVE PERFORMANCE. HERE z IS THE ATTACK EVOLUTION.

Shuffle Frequency	Post-Attack TA (%)	No. of Attack Rounds
Every Inference	11.8	193
Every 2 Inference	11.88	93
Every 5 Inference	11.78	73
No defense	11.86	16

We investigate the defender's tools to impact the defense performance and overhead. A critical consideration for our proposed DeepShuffle defense is the choice of shuffling frequency and its impact on the defense performance. Suppose the attacker selects $z = 20$, and we gradually reduce the frequency of applying DeepShuffle from every inference to every two inferences until there is no defense. We find that periodically reducing the shuffling frequency of DeepShuffle decreases its effectiveness. Therefore, to achieve the best defense effect, the defender must update the channels at every inference but pay the highest overhead of approximately $\sim 3\%$ on VTA, as shown in Sec. 5.7.

7. Future Work and Conclusion

DeepShuffle is a general, training-free defense methodology for a wide range of deep learning models in various domains, such as computer vision, natural language processing, and robotics. As expected, we also observe that DeepShuffle is less effective when attackers keep increasing their attacking rounds. This is since as a type of "moving-target-defense" (MTD), DeepShuffle strives to randomize the well-profiled attacking opportunities by a strong adversary, so as to increases the number of attack rounds and reduce the likelihood of successful attacks [55], [56]. To date, no prior works have successfully demonstrated the Deep-Dup attack on large-language models due to their size (i.e., billions of parameters) and the associated computational and memory overhead in FPGA. Moreover, these potential attacks need a set of newly defined attack objectives, searching algorithm, and evaluation criteria for attack success. We acknowledge that these present a limitation of the original Deep-Dup attack, not the effectiveness of DeepShuffle. We envision that DeepShuffle can be applied to protect these application with further optimization, e.g., to address the challenges posed by recent large-language processing models like BERT [57]. We plan to investigate these directions in our future work.

Based on the experimental evaluation results, we conclude that some design properties of the VTA accelerators,

e.g., placing the first layer on the CPU for high parallelism, also enhance the defensive capabilities of DNN models. The last FC layer is computed using floating-point calculations and placed on the CPU to ensure the accuracy of the quantized model. However, as shown in Fig. 9, these two layers are relatively vulnerable within the overall model. Therefore, using VTA to run DNNs has, to some extent, improved the baseline robustness. We believe that in cloud scenarios where CPUs are usually quite powerful, performing some vulnerable computations on CPU for security purposes can effectively avoid attacks like Deep-Dup. This paper provides a preliminary exploration in Fig. 1 on which layers are suitable to be placed on the CPU. Additionally, we found that traditional, sequential models have stronger robustness in resisting Deep-Dup, and DeepShuffle greatly amplifies this defensive effect.

Acknowledgments

The authors would like to thank the shepherd from the IEEE S&P'24 committee and the anonymous reviewers, for their insightful comments that greatly enriched the quality of our work. This work is supported in part by the U.S. National Science Foundation under Grants CNS-2019548, CNS-2153525, OAC-2319962, CNS-2239672, CNS-2153690, CNS-2326597, and CNS-2247892.

References

[1] A. S. Rakin, Y. Luo, X. Xu, and D. Fan, "Deep-dup: An adversarial weight duplication attack framework to crush deep neural network in multi-tenantfpga," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1919–1936.

[2] "What is paas?" [Online]. Available: https://azure.microsoft.com/en-gb/resources/cloud-computing-dictionary/what-is-paas/

[3] "What is iaas?" [Online]. Available: https://azure.microsoft.com/en-gb/resources/cloud-computing-dictionary/what-is-iaas/

[4] "Cloud computing services - amazon web services (aws)." [Online]. Available: https://aws.amazon.com/

[5] "Microsoft azure: Cloud computing services." [Online]. Available: https://azure.microsoft.com/en-us

[6] "What is a hypervisor?" [Online]. Available: https://aws.amazon.com/what-is/hypervisor/

[7] "Vck5000 versal development card." [Online]. Available: https://www.xilinx.com/products/boards-and-kits/vck5000.html

[8] "Nvidia a100 tensor core gpu." [Online]. Available: https://www.nvidia.com/en-us/data-center/a100/

[9] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 446–459.

[10] "Amd/xilinx takes aim at nvidia with improved vck5000 inferencing card." [Online]. Available: https://www.hpcwire.com/2022/03/08/amd-xilinx-takes-aim-at-nvidia-with-improved-vck5000-inferencing-card/

[11] Y. Zha and J. Li, "Virtualizing fpgas in the cloud," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 845–858.

[12] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, protection, and compatibility for reconfigurable fabric with amorphos," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 107–127.

[13] D. Korolija, T. Roscoe, and G. Alonso, "Do os abstractions make sense on fpgas?" in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020, pp. 991–1010.

[14] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci, "A hypervisor for shared-memory fpga platforms," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 827–844.

[15] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin *et al.*, "A hardware–software blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, 2019.

[16] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[17] E. D. Cubuk, B. Zoph, S. S. Schoenholz, and Q. V. Le, "Intriguing properties of adversarial examples," *ICLR workshop*, 2018.

[18] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[19] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJzIBfZAb

[20] Z. He, A. S. Rakin, and D. Fan, "Parametric noise injection: Trainable randomness to improve deep neural network robustness against adversarial attack," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 588–597.

[21] A. Raghunathan, J. Steinhardt, and P. Liang, "Certified defenses against adversarial examples," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=Bys4ob-Rb

[22] F. Yao, A. Rakin, and D. Fan, "Deephammer: Depleting the intelligence of deep neural networksthrough targeted chain of bit flips," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[23] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 361–372.

[24] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn dnn architectures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2003–2020.

[25] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, "Hermes attack: Steal dnn models with lossless inference accuracy," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[26] J. Li, A. S. Rakin, Y. Xiong, L. Chang, Z. He, D. Fan, and C. Chakrabarti, "Defending bit-flip attack through dnn weight reconstruction," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[27] L. Liu, Y. Guo, Y. Cheng, Y. Zhang, and J. Yang, "Generating robust dnn with resistance to bit-flip based adversarial weight attack," *IEEE Transactions on Computers*, 2022.

[28] G. Saileshwar, B. Wang, M. Qureshi, and P. J. Nair, "Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 1056–1069.

[29] J. Woo, G. Saileshwar, and P. J. Nair, "Scalable and secure row-swap: Efficient and safe row hammer mitigation in memory systems," *arXiv preprint arXiv:2212.12613*, 2022.

[30] Z. He, A. S. Rakin, J. Li, C. Chakrabarti, and D. Fan, "Defending and harnessing the bit-flip based adversarial weight attack," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 14 095–14 103.

[31] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Vta: an open hardware-software stack for deep learning," *arXiv preprint arXiv:1807.04188*, 2018.

[32] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020, pp. 863–879.

[33] "Xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency." [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp380_Stacked_Silicon_Interconnect_Technolog

[34] "Vivado design suite user guide: Dynamic function exchange (ug909)." [Online]. Available: https://docs.xilinx.com/r/en-US/ug909-vivado-partial-reconfiguration/Introduction-to-Dynamic-Function-eXchange

[35] "Alveo u280 data center accelerator card data sheet." [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds963\protect\discretionary{\char\hyphenchar\font}{}{}u280.pdf

[36] T. M. La, K. Matas, N. Grunchevski, K. D. Pham, and D. Koch, "Fpgadefender: Malicious self-oscillator scanning for xilinx ultrascale+ fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 3, pp. 1–31, 2020.

[37] Y. Luo, C. Gongye, Y. Fei, and X. Xu, "Deepstrike: Remotely-guided fault injection attacks on dnn accelerator in cloud-fpga," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 295–300.

[38] J. Krautter, D. R. Gnad, and M. B. Tahoori, "Fpgahammer: Remote voltage fault attacks on shared fpgas, suitable for dfa on aes," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 44–68, 2018.

[39] S. Tian, S. Moini, A. Wolnikowski, D. Holcomb, R. Tessier, and J. Szefer, "Remote power attacks on the versatile tensor accelerator in multi-tenant fpgas," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 242–246.

[40] "Zcu104 board user guide (ug1267)." [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1267-zcu104-eval-bd

[41] Developer preview – ec2 instances (f1) with programmable hardware. https://aws.amazon.com/cn/blogs/aws/developer-preview-ec2-instances-f1-with-programmable-hardware/.

[42] Create an f3 instance. https://www.alibabacloud.com/help/en/elastic-compute-service/latest/create-an-f3-instance.

[43] H. Yu, H. Ma, K. Yang, Y. Zhao, and Y. Jin, "Deepem: Deep neural networks model recovery through em side-channel information leakage," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2020, pp. 209–218.

[44] F. Benevenuti, F. Libano, V. Pouget, F. L. Kastensmidt, and P. Rech, "Comparative analysis of inference errors in a neural network implemented in sram-based fpga induced by neutron irradiation and fault injection methods," in *2018 31st Symposium on Integrated Circuits and Systems Design (SBCCI)*. IEEE, 2018, pp. 1–6.

[45] Pytorch, "Conv2d." [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

[46] "Vitis ai model zoo," https://github.com/Xilinx/Vitis-AI/tree/master/model_zoo.

[47] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[48] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[49] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[50] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[51] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: The next generation of on-device computer vision networks," in *CVPR*, 2018.

[52] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[53] L. Zhang, L. Zhang, X. Mou, and D. Zhang, "Fsim: A feature similarity index for image quality assessment," *IEEE transactions on Image Processing*, vol. 20, no. 8, pp. 2378–2386, 2011.

[54] A. S. Rakin, Z. He, and D. Fan, "Bit-flip attack: Crushing neural network with progressive bit search," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 1211–1220.

[55] H. Okhravi, W. W. Streilein, and K. S. Bauer, "Moving target techniques: leveraging uncertainty for cyber defense," *Lincoln Laboratory Journal*, vol. 22, no. 1, pp. 100–109, 2016.

[56] D. Evans, A. Nguyen-Tuong, and J. Knight, "Effectiveness of moving target defenses," *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, pp. 29–48, 2011.

[57] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[58] W. Dai, C. Dai, S. Qu, J. Li, and S. Das, "Very deep convolutional neural networks for raw waveforms," in *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2017, pp. 421–425.

[59] "Speech command classification with torchaudio," https://pytorch.org/tutorials/intermediate/speech_command_classification_with_torchaudio_tutorial.html.

[60] A. S. Rakin, Z. He, and D. Fan, "Bit-flip attack: Crushing neural network with progressive bit search," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2019, pp. 1211–1220.

Appendix A.
The complete output of the 2^{nd} Conv2D layer

In Sec. 4.4, we demonstrated the workflow of DeepShuffle, where we intercepted the output of the 2^{nd} Conv2D layer in a ResNet-18 implementation. This model inference a wolf image belonging to the ImageNet. In Fig. 10, we provide the complete output of all 64 channels of the 2^{nd} Conv2D layer, highlighting the three different effects introduced by the Deep-Dup attack, as shown in Fig. 5. We observe that the Deep-Dup injects faults into the weight transmission process, which then propagates to the output feature maps. The information-rich feature maps exhibit three different changes: brightening (red channels), disappearance (yellow channels), and appearance (blue channels). These channel-wise changes in information will further propagate in subsequent layers of inference and eventually lead to misclassification.

(a) Clean model output feature maps of the 2^{nd} Conv2D layer (Baseline)

(b) Output feature maps w/ Deep-Dup on the 2^{nd} Conv2D layer

Figure 10. The internal result of the 2^{nd} Conv2D layer in ResNet-18 while applying a wolf image from the ImageNet. (a) The outputs w/o any attack. (b) Applying Deep-Dup attacks on the corresponding layer and output all channels' results.

Appendix B. Evaluation on Speech Dataset

In addition to these image processing tasks/applications, we also evaluate the performance of DeepShuffle on Google's Speech Command Dataset, a popular vocabulary recognition dataset [52], with M5 architecture [58]. It takes input speech audio and classifies them into 35 different output classes. We follow the experimental evaluation setup as the official Pytorch website [59]. We summarize the results of our experimental evaluation in Tab. 6.

TABLE 6. EVALUATION OF DEEPSHUFFLE ON GOOGLE'S SPEECH COMMAND DATASET. IT HAS 35 DIFFERENT CLASSES, HENCE THE ATTACK OBJECTIVE IS TO REACH 3 % (I.E., RANDOM LEVEL) ACCURACY. WE USED $z = 10$ FOR THIS EXPERIMENT.

Method	Clean Accuracy	No. of Attack Rounds	Accuracy Under Attack
Baseline	84.0	49	3.0
DeepShuffle	84.0	*351 (7×)*	3.0

Our evaluation demonstrates that the original Deep-Dup attack also remains equally effective on attacking speech datasets. Nevertheless, our proposed DeepShuffle improves model robustness against the Deep-Dup attack by increasing the attacker's efforts, i.e., the required number of attacking rounds to degrade the accuracy to a random level (e.g., 3%) by *7×*. This result is consistent with our evaluation of the vision dataset and demonstrates the effectiveness of DeepShuffle against Deep-Dup across multiple domain tasks.

Appendix C. Applicability of DeepShuffle

In Sec. 4.3, we exhibit DeepShuffle's applicability beyond convolutional layers through its implementation on VGG-11's fully connected (FC) layers. This highlights DeepShuffle's generality, i.e., extending to architectures such as RNN, LSTM, DenseNet, and GRU. Principally, a layer or operator meets two criteria to harness DeepShuffle: (1) exhibits multi-channel structure with consistent operations across channels, and (2) contains trainable parameters.

Appendix D. Meta-Review

D.1. Summary

This paper presents DeepShuffle to protect neural networks against fault injection attacks on multi-tenant cloud FPGAs. It achieves the goal by dynamically shuffling model parameters at runtime, effectively preventing attackers from obtaining critical information required for the attacks. DeepShuffle requires no hardware modification or additional training.

D.2. Scientific Contributions

- Addresses a Long-Known Issue.
- Provides a Valuable Step Forward in an Established Field.
- Creates a New Tool to Enable Future Science.

D.3. Reasons for Acceptance

1) The paper addresses a long-known issue. The proposed framework effectively thwarts the recently-proposed adversarial fault injection attacks (Deep-Dup [1], USENIX Security'21) on cloud FPGAs that significantly degrade the performance of DNN models.
2) The paper provides a valuable step forward in an established field. The authors highlight the limitations of Deep-Dup on fault injection attacks and re-implement the end-to-end attack flow using a multi-tenant cloud-FPGA setup. They re-calibrate the attack on the VTA-generated DNN models and conduct a thorough analysis of DNN model behaviors under the work. Based on the analysis, the authors propose a systematic defense framework against Deep-Dup like adversarial fault injection attacks.
3) The paper creates a new tool to enable future science. The proposed solution requires no hardware modification or extra training, which can be a good tool for facilitating future research.

D.4. Noteworthy Concerns

1) One concern shared by reviewers is the defense generalizability—whether DeepShuffle can achieve similar results for a broader range of neural networks other than image processing models, whether it can scale to large networks, and whether it can defend against similar attacks other than Deep-Dup.
2) Another concern is the significance of the solution as some results in the paper indicate that DeepShuffle is less effective when adversarial increase their attacking strength.

Appendix E. Response to the Meta-Review

E.1. Concern about the defense generalizability

E.1.1. Apply DeepShuffle on a broader range of neural networks other than image processing models. Principally, the applicability of DeepShuffle is not dependent on a specific application/task, as the fault injection attack will not be limited to any specific models. In addition to image processing models, we evaluated DeepShuffle on Google's Speech Command Dataset with M5 architecture [58] using a combination of CNN and FC layers in Appendix B.

E.1.2. DeepShuffle can scale to large networks. Until now, the original Deep-Dup attack [1] has only been validated/demonstrated on architectures using fully-connected and convolutional layers, which is also our choice, for fair defense evaluation purposes. Concerning large networks, we evaluated DeepShuffle on ResNet-50 using the ImageNet dataset, which represents large image processing networks, i.e., both the model and dataset are large following the standard evaluation practice in most DNN security papers.

E.1.3. DeepShuffle can defend against other relevant adversarial fault injection attacks. Following the threat model (Sec.3.1), DeepShuffle is geared against Deep-Dup attack (i.e., adversarial fault injection during off-chip data communication) in a black-box setting. Since most SOTA adversarial weight attacks can only be executed in white/gray-box attacks, such as the Bit Flip Attack (BFA) [60]; therefore, we envision that DeepShuffle can bolster the robustness of DNN models and potentially protect them against similar adversarial fault injection attacks other than Deep-Dup, as long as the same defense philosophy (i.e., moving-target-defense) is applied to challenge the attacker.

E.2. Concern about the effectiveness of DeepShuffle when adversaries increases their attacking strength.

We agree with the reviewers about this concern. However, the basic defense philosophy of "moving-target-defense" (MTD) lies in "randomizing cyber system components to reduce the likelihood of successful attacks" [55], so as to "make it much more difficult for an attacker to exploit a vulnerable system by changing aspects of that system to present attackers with a varying attack surface" [56]. As an example defense strategy of MTD, DeepShuffle increases the number of attack rounds, making it infeasible for an attacker to consistently degrade the model's accuracy, as confirmed in most studies on adversarial weight attacks and defenses [26], [30].

Meanwhile, we acknowledge that increasing attack rounds introduces additional overhead in terms of resources and time, making such attacks more susceptible to detection. Therefore, DeepShuffle proves effective and successful in only a few cases where the post-attack accuracy is around 40%, significantly amplifying the attack overhead by multiple folds (3-10$\times$).

In the future, we plan to explore ways to enhance the internal robustness of the DNN through improved model architecture and training algorithms, which can further support the effectiveness of DeepShuffle.