A new lightweight in-situ adversarial sample detector for edge deep neural network

Wang, Si; Liu, Wenye; Chang, Chip-Hong

2021

Wang, S., Liu, W. & Chang, C. (2021). A new lightweight in-situ adversarial sample detector for edge deep neural network. IEEE Journal of Emerging and Selected Topics in Circuits and Systems, 11(2), 252-266. <https://dx.doi.org/10.1109/JETCAS.2021.3076101>

<https://hdl.handle.net/10356/148567>

<https://doi.org/10.1109/JETCAS.2021.3076101>

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The published version is available at: <https://doi.org/10.1109/JETCAS.2021.3076101>.

Downloaded on 26 Apr 2023 08:45:42 SGT

A New Lightweight In-situ Adversarial Sample Detector for Edge Deep Neural Network

Si Wang, *Student Member, IEEE*, Wenye Liu, *Student Member, IEEE*, and Chip-Hong Chang, *Fellow, IEEE*

Abstract—The flourishing of Internet of Things (IoT) has rekindled on-premise computing to allow data to be analyzed closer to the source. To support edge Artificial Intelligence (AI), hardware accelerators, open-source AI model compilers and commercially available toolkits have evolved to facilitate the development and deployment of applications that use AI at its core. This paradigm shift in deep learning computations does not, however, reduce the vulnerability of deep neural networks (DNN) against adversarial attacks but introduces a difficult catch-up. This is because existing methodologies rely mainly on offline analysis to detect adversarial inputs, assuming that the deep learning model is implemented on a 32-bit floating-point graphical processing unit (GPU) instance. In this paper, we propose a new hardware-oriented approach for in-situ detection of adversarial inputs feeding through a spatial DNN accelerator architecture or a third-party DNN Intellectual Property (IP) implemented on the edge. Our method exploits controlled glitch injection into the clock signal of the DNN accelerator to maximize the information gain for the discrimination of adversarial and benign inputs. A light gradient boosting machine (lightGBM) is constructed by analyzing the prediction probability of unmutated and mutated models and the label change inconsistency between the adversarial and benign samples in the training dataset. With negligibly small hardware overhead, the glitch injection circuit and the trained lightGBM detector can be easily implemented alongside with the deep learning model on a Xilinx ZU9EG chip. The effectiveness of the proposed detector is validated against four state-of-the-art adversarial attacks on two different types and scales of DNN models, VGG16 and ResNet50, for a thousand-class visual object recognition application. The results show a significant increase in true positive rate and a substantial reduction in false positive rate on the Fast Gradient Sign Method (FGSM), Iterative-FGSM (I-FGSM), C&W and universal perturbation attacks compared with modern software-oriented adversarial sample detection methods.

Index Terms—Deep Neural Network, Deep Learning Accelerator, Edge AI, Adversarial Examples, AI Security.

I. INTRODUCTION

DEEP learning [1] is a key driver that puts artificial intelligence (AI) on the radar screen for technology investment. Deep Neural Network (DNN) automatically learns high-level features directly from raw data in a hierarchical manner, which eliminates manual extraction of effective features in traditional

machine learning solutions. The ability to solve problems end-to-end enables a system to learn complex functions mapping for ill-posed problems with prediction accuracy often exceeding sophisticated statistical models and other machine learning methods. Computer vision is one specific domain that DNN has demonstrated this remarkable abstraction power [2]. Unlike other mainstream classification approaches, DNN can usually achieve better results with more data and larger model. Over the last decade, the model complexity and regulation mechanism of DNN have grown tremendously to overcome the performance plateau and improve the generalization ability.

The cross over from statistical models to data science in early AI evolution is motivated by running powerful deep learning algorithms on the cloud. Despite the economy of scale of cloud computing, data velocity is constrained by the network bandwidth and availability. The boom of Internet of Things (IoT) has changed the way data are generated and curated. It motivates a distributed computing model that enables more user-centric AI applications to be run on the device without being limited by the network latency. Unfortunately, large learning models also require more computations, which make them infeasible for direct integration into the edge devices with limited computing power and memory. Hardware accelerators have thus evolved to support the in-situ computations of deep learning algorithms with fast and accurate inferences [3], [4], [5]. Techniques such as low-cost data flow, data quantization, network compression and dynamic voltage-frequency scaling, have been adopted for the design of DNN accelerators to boost energy efficiency, reduce the memory requirement, and lower the computational complexity [6]. Together with the emerging open-source AI model compilers [7] and commercially available toolkits like Intel®OpenVINO™, pre-trained deep learning models can be optimized and mapped into a reusable hardware abstraction layer for implementation on high-speed and power-efficient edge AI accelerator platforms.

The bridge between edge computing and AI opens out many new application possibilities, including self-driving cars, video surveillance, contact tracing and border control. At the same time, this has also made edge AI an attractive target for adversarial attacks. The black-box nature of such super-intelligence can be easily exploited to turn an essentially benevolent system to malevolent, which poses safety, privacy and security hazards [8]. It is a well-known fact that DNN is susceptible to adversarial examples, which are carefully crafted perturbations added to the benign inputs to fool a well-trained classifier [9]. Adversarial examples can be generated fast at low cost. Their attack surface can also be extended

This research is supported by the National Research Foundation, Singapore, under its National Cybersecurity Research & Development Programme/Cyber-Hardware Forensic & Assurance Evaluation R&D Programme (Award: CHFA-GC1-AW01).

S. Wang, W. Liu and C. H. Chang are with the Department of Electrical and Electronic Engineering, Nanyang Technological University, Singapore 639798. e-mails: (si003@e.ntu.edu.sg, wliu015@e.ntu.edu.sg, echchang@ntu.edu.sg).

Manuscript received December 7, 2020, revised February 2, 2021.

beyond the software boundary and made more robust with high transferability across models. Countermeasures against adversarial examples have been explored and proposed in recent years. However, they are mainly designed and investigated based on software models of DNNs. Most existing adversarial input detection methods assume that the adversarial examples are generated with low to moderate confidence perturbations. Robust high confidence perturbations that target edge AI with DNN models optimized for efficient hardware implementation were not sufficiently evaluated. Although truncated models have been used for evaluations by a few latest schemes [10], [11], the detections are simulated on graphical processing unit (GPU) platform without physically applying the adversarial inputs into the DNN through the hardware accelerator. Most importantly, as these solutions are mostly developed for off-line analysis, their code sizes and computation complexity are too high to be fitted into the edge AI platform processor or reconfigurable fabrics. To allow the adversarial inputs to be identified in-place with the inference machine, a hardware architecture [12] was recently proposed to accelerate several existing adversarial input detection algorithms. The architecture has very high hardware overheads to accommodate multiple detection schemes. Another recent hardware-oriented solution [13] detects adversarial inputs on only small-scale classification task. The solution is not scalable for massive classification.

In this paper, we propose a new lightweight adversarial input detector that can be efficiently implemented alongside a spatial architecture for edge DNN implementation. The idea is inspired by our preliminary layer directed discriminative noise injection scheme [10] but with fundamentally different mutation mechanisms, optimal label change mutation strategies and detector design. Without transforming the input image, and without modifying the edge implementation of the deployed DNN model, maximal label change inconsistency between the benign and adversarial inputs can be triggered by adaptively mutating the DNN computations. The mutation is controlled by injecting glitches of selective intensity at specific timing into the clock signal of the hardware accelerator. The optimal glitch intensity that maximizes the label change inconsistency between the adversarial and benign images for each layer is used for the construction of a light gradient boosting machine (lightGBM) [14]. The test node attributes of the lightGBM are determined by an offline training which maximizes the information gain of the training dataset of each node based on three different types of probabilities, which are the top-1 probabilities predicted by the unmutated model, the mutated model and their difference. The trained lightGBM is a small two-class tree classifier consisting of only a few test nodes. The glitch injection circuit and tree detector are implemented together with the convolution network on a Xilinx ZU9EG chip. The performance of the proposed in-situ detector is evaluated by applying four different popular adversarial attacks on two different DNN architectures trained on the ImageNet dataset. The attacks evaluated are FGSM [15], iterative-FGSM (I-FGSM) [16], C&W [17] and universal perturbation (UP) [18]. The results and overheads are compared against the recently proposed software and hardware oriented adversarial

input detectors.

The rest of the paper is organized as follows. Section II introduces the preliminaries of deep learning architecture and its hardware accelerator, different adversarial example generation algorithms, existing countermeasures against DNN adversarial attacks, and effect of clock glitch. Section III presents the threat model, the motivation of using glitch injection based mutation for adversarial input detection and the proposed breadth-first and best-first decision tree detectors. Section IV presents and discusses the experimental results. Section V concludes the paper.

II. PRELIMINARIES

A. Deep Learning and Hardware Accelerator

Deep learning is a large subfield of machine learning algorithms [6]. One prevalent type of DNNs is Convolutional Neural Networks (CNNs). The shared-weight architecture and translation invariance attributes of CNN allow diverse features to be learnt and generalized well for varied scenarios of image recognition problems [19], [20]. Majority of the CNNs follow a similar construction of consecutive convolutional (Conv) sections. Each Conv section can contain multiple Conv blocks for feature extraction, followed by one or more fully connected (FC) blocks to compute the class scores, as shown in Fig. 1. Typically, each Conv block contains a Conv layer, a rectified learning unit (ReLU) layer and discretionally a normalization layer and a pooling layer, and each FC block consists of a FC layer and a ReLU layer. In a Conv layer, the input feature map (ifmap) is convolved by a set of 3D filters to obtain the output feature map (ofmap). In each convolutional operation, a filter is applied to the windowed ifmap as follows:

$$y_{conv} = \sum_{i=1}^R \sum_{j=1}^S \sum_{k=1}^C w_{i,j,k} \times x_{i,j,k} + b \quad (1)$$

where y_{conv} is an entry in the ofmap, R , S , C refer to the height, width and channel length or depth of the filter, respectively, x is the windowed ifmap, and b is the bias.

In a FC layer, the output feature vector is computed by a matrix-vector multiplication as follows:

$$Y_{fc} = W \times X + b \quad (2)$$

where W is the weight matrix and X is an input feature vector. A ReLU layer applies the activation function $f(x) = \max(0, x)$ to each entry of the ofmap. It is inserted after every Conv or FC layer to introduce non-linearity into the network.

Multiply-and-accumulate (MAC) operation is the main computation of a DNN. It is used extensively in Conv and FC layers, as shown in (1) and (2). In modern deep learning models, hundreds of million to billion MACs are executed per image before an inference is made. This enormous computational requirement has posed great challenge in their implementation on edge devices. Various customized hardware accelerators and domain-specific computing platforms with supporting tool chains have emerged to significantly improve the energy efficiency and reduce the latency. Precision reduction and parallel processing are two common techniques used for DNN hardware acceleration [21], [5]. Precision reduction quantizes

Fig. 1: Conventional configuration of CNN.

Fig. 2: Typical architecture of DNN accelerator.

the inner data from 32-bit floating point to fixed-point. It not only down-scales the power consumption and hardware area, but also relaxes the storage requirement [22]. Parallel processing is realized in temporal and spatial architectures [6]. In temporal architecture, also known as single instruction multiple thread (SIMT) or single instruction multiple data (SIMD), the control logic and memory are shared by multiple arithmetic logic units (ALUs). Each ALU works independently and has no data transfer among them. Operations such as ReLU, normalization and pooling are carried out on this engine [23]. Spatial architecture, in contrast, allows inter-communication between neighboring ALUs and assigns a private control logic and local memory (register file) to every ALU. Such ALU is more commonly called a processing engine (PE), as it is simply a MAC unit made of a fixed-point multiplier and an accumulator. Fig. 2 shows a typical architecture of a DNN accelerator. A host processor is used for scheduling the DNN operations. The data controller schedules the dataflow to save off-chip memory accesses. A separate high frequency clock is employed for the PE array to increase the throughput. A clock gating signal (*en_gating*) is introduced to further reduce the power consumption by disabling the clock signal to the PE array in the idle mode [24].

B. Adversarial Examples

The vulnerability of DNN was first demonstrated by Szegedy et al. [9] using the adversarial examples generated by box-constrained limited-memory-Broyden-Fletcher-Goldfarb-Shanno (LBFGS) algorithm. Today, adversarial examples can be generated by three representative methodologies. For ease

of exposition, the benign input, the malicious input, the deep learning model and the target class are denoted by x , $\hat{x}$, F and t , respectively.

1) *Fast Gradient Sign method (FGSM)*: This method calculates the gradient direction of the cost function for each pixel of an input image to determine the direction of added distortions with the minimum cost. The computation can be expressed mathematically as follows [15]:

$$\hat{x} = x - \epsilon \cdot \text{sign}(\nabla J_{F,t}(x)) \quad (3)$$

where J is a cost function and ϵ is an input variation parameter that controls the amplitude of perturbation. The cost function J evaluates the error between the output of F and the target label t . ϵ should be set sufficiently small to make the perturbation unnoticeable. FGSM is a one-shot attack that aims to be fast rather than optimal. Its variant, iterative FGSM (I-FGSM), applies FGSM recursively with a smaller step size to improve the success rate of an attack [16].

2) *C&W*: This method is named after its inventors, Nicholas Carlini and David Wagner. It takes the original problem from [9]:

$$\begin{aligned} & \text{minimize } D(x, \hat{x}), \\ & \text{s.t. } F(\hat{x}) = t \wedge \hat{x} \in [0, 1]^n, \end{aligned} \quad (4)$$

and reformulates it to (5) [17] for ease of optimization.

$$\text{minimize } \|\delta\|_p + c \cdot f(\hat{x}, t), \text{ s.t. } \hat{x} \in [0, 1]^n, \quad (5)$$

where D , δ and c are the distance metric, the perturbation and a carefully chosen constant, respectively. f is introduced to simplify the problem. The objective function is carefully selected to satisfy the nonlinear constraint $F(\hat{x}) = t$ when $f(\hat{x}, t) \leq 0$. The best f is found to be:

$$f(\hat{x}, t) = \max(\max\{G(\hat{x})_k : k \neq t\} - G(\hat{x})_t, 0), \quad (6)$$

where $G(\hat{x})$ is the logit vector of a classifier.

3) *Universal Perturbation (UP)*: This method is extended from the DeepFool attack [25], which utilizes the geometric characteristics of the classifier to calculate the minimum perturbation required to cross the nearest decision boundary. It derives the image-agnostic adversarial perturbation by the following formulation [18]:

$$\begin{aligned} \delta &:= \{\delta | \text{err}(X_\delta) := \frac{1}{m} \sum_{i=1}^m 1_{F(x_i+\delta) \neq F(x_i)} \geq 1 - \text{acc}\}, \\ & \text{s.t. } \|\delta\|_p \leq \xi. \end{aligned} \quad (7)$$

where X_{clean} , X_δ , acc and ξ are a set of clean images $\{x_1, \dots, x_m\}$, a set of perturbed data inputs $\{x_1 + \delta, \dots, x_m + \delta\}$, the proportion of correctly classified tampered images, and a parameter that limits the extent of distortion δ , respectively. The algorithm iteratively computes the minimum perturbation that will send the current tampered input to the decision boundary, and updates the instance of UP. The UP is quasi-imperceptible to human vision system.

C. Related Works

Existing countermeasures can be broadly divided into two categories: robustness improvement and adversarial detection. The former restores the correct prediction of a malicious input without sacrificing the classification accuracy of the benign samples. The latter identifies the adversarial inputs.

Adversarial training, gradient masking and input transformation are three main approaches to robustness enhancement. Adversarial training increases the model's tolerance towards malicious input distortions. It adds off-the-shelf adversarial data into the original dataset for training [15]. However, this has later shown to be ineffective against new adversarial attack algorithms that were not considered in the adversarial training [26]. Gradient masking hides the gradient information by removing the gradients in specific directions and neighborhoods of the training points. The aim is to flatten the model so that the adversary has no clue to discover which points and directions to perturb an image across its decision boundaries. However, this defense can be easily circumvented by training a substitute model using the inputs chosen by the adversary to imitate the defend model. Due to the transferability of DNN, the adversarial examples can be determined from the gradients of the substitute model to cause the input to be misclassified by the defend model, as demonstrated by the black-box attacks in [27], [28]. Input transformation diminishes the adversarial effect of perturbations by reconstructing the inputs [29], [30], [31], [32]. It is achieved by either dimensionality reduction techniques such as Principal Component Analysis (PCA) [32] or image filters [29], [31], or compressing the inputs by reformer networks [30]. Since PCA obliterates the spatial structure of the input by projecting it onto the space that preserves only the principal components, local information needed by the convolutional layers of modern DNNs for classification is not fully preserved in the transformed space. Therefore, a redesign of the model architecture is required. Other input sanitizers like filter and reformer are evaluated in the grey-box setting. The attacker is assumed to know the target classifier but not the input transformation. They can be defeated by a slight increase in the perturbation strength in a white-box setting, where the mapping of the information carrying variables to the latent space is taken into consideration for the generation of adversarial examples [33].

Since DNNs can never achieve 100% accuracy even with clean inputs, increasing their compatibility with malicious inputs may compromise the accuracy of the benign input classification. By contrast, the detection of adversarial inputs can avoid this problem and allow different actionable options to be taken against the adversaries. Adversarial detection can be achieved by sample statistics, detector training and prediction inconsistency. Sample statistics rely on the collection and analysis of a large scale of sample data to discover the difference between the adversarial and clean data [34], [11], [13]. They are mostly computationally intensive and can only be done offline. Moreover, they are effective only against adversarial examples lying far from the manifolds of the legitimate population. Adversarial examples can also be detected by steganalysis [35]. The steganalysis features

are enhanced by estimating the probability of modifications caused by adversarial attacks. However, this method requires the knowledge of the attack algorithm. A different detection process will have to be carried out for each attack algorithm. As the adversarial example generation algorithm is not known in advance, a brute-force execution of all the different processes is required to detect an unknown adversarial input. This attack-dependent defense achieves very high detection rate at the expense of impractically high computational resource and timing overheads. For detector training, a custom subnetwork is embedded into the original network structure for adversarial sample detection [36]. The training of the subnetwork is computationally expensive. This strategy is also prone to overfitting by including the adversarial training. Prediction inconsistency exploits the disagreement of predictions between adversarial and benign inputs by different models [37], [38], [10], [39]. Variant of this method has been proposed to reduce the latency by comparing the predictions of the original image against the transformed image produced by the input transformation strategy.

The abovementioned approaches mostly target software implementation of DNN on central processing unit (CPU)/GPU platforms. Rouhani et al. [13] proposed DeepFense, the first end-to-end hardware accelerated framework for real-time detection of adversarial examples. It introduces Modular Robust Redundancy (MRR) to characterize the potential adversarial sub-spaces spanned by the output feature maps in each layer of the target DNN model. An automatic customization tool has also been designed for the maximization of the performance in various resource-constrained hardware platforms. However, as each MRR is architecturally identical to the victim deep learning model, it incurs substantial hardware and power overheads, especially when the model size and the number of MRRs are large. Furthermore, DeepFense was evaluated with only small-scale datasets. The method may not scale well with large-scale datasets like ImageNet. Wang et al. [12] devised DNNGuard, an elastic heterogeneous DNN accelerator architecture that can support simultaneous execution of target DNN network and its corresponding detection network. It has the flexibility to adapt to various existing adversarial defense algorithms. Nevertheless, DNNGuard was implemented on a simulated instead of a real deep learning accelerator. Its adaptability, as well as online performance penalty and hardware overheads have not been evaluated with actual integration of DNNGuard into the physical DNN accelerator.

D. Clock Glitch

A clock glitch is a short-lived fault introduced by irregular pulses in the clock signal [40]. It may cause a timing violation of a datapath, as illustrated in Fig. 3, where d_{in} is the data transmitted from a register A through a combinational logic to a register B and d_{out} is the output data of the register B . The existence of a clock glitch reduces the clock period, causing a wrong sampling of d_{in} . This can introduce an error into the register B . Clock glitch injection has been demonstrated to be an effective attack method [41] to propagate incorrect intermediate results in a MAC to the final layer. It leaves no

Fig. 3: Timing diagrams of a typical sequential logic path under (a) a normal clock and (b) a glitched clock.

trace for diagnosis after the inference as the glitch occurs only instantaneously. Due to the undesirable and detrimental effects, clock glitch has never been considered as a useful source of noise by the defender. In this work, we will reverse it into a controllable uncertainty at the advantage of the defender to blindside the adversary.

III. PROPOSED ONLINE ADVERSARIAL INPUT DETECTOR

A. Threat Model

The proposed countermeasure intends to defend against a vigorous attacker who is equipped with the information about the target integrated DNN classifier deployed in the field but has no accessibility to its inner system. The attacker can generate adversarial images that are imperceptible or quasi-imperceptible to human visual systems to fool the DNN using the white-box, gray-box or black-box techniques. We assume that the attacker can obtain the network architecture and weight parameters of the DNN model through various plausible means like eavesdropping the side-channel information of model internal variables [42]. We also assume that the attacker is able to train a substitute model with the knowledge of the model structure to emulate the victim model through query-based model extraction attacks [43], [44]. Except physically tampering the internal nodes and nets of a monolithic integrated circuit, the attacker can make any non-destructive manipulations if they do not leave any apparent evidence. In other words, the malicious acts performed by the attacker should not result in a noticeable rate of interferences and prolong unavailability that can be trivially detected or traced. Under this premise, the attacker cannot probe or tamper the circuit internals to monitor the inner operations of the DNN in order to disable or bypass the built-in defence mechanism.

B. Label Change Inconsistency between Original and Adversarial Inputs on Clock Glitch Injection

This work is motivated by the model-based mutation testing, where the mutants are generated by making slight modifications to the original system model. This technique is originally

used to automate the generation of a high-quality test suite to distinguish the original model from all non-equivalent mutants. We believe that such model-based mutation technique can also be used to test the integrity of the input data to a computational model like a DNN instead of checking the conformance of a model. To confirm this conjecture, we extracted multiple dominant layers that possess high distinguishability between adversarial and benign images with random mutations in [10]. These dominant layers were identified by observing the label change rate of the mutated models when randomly selected weight parameters of each layer are perturbed by deliberately added noise. Although the results can be simulated by software execution on temporal architectures such as GPU or CPU, the adaptive thresholding scheme cannot be efficiently implemented on a spatial architecture used for DNN in application-specific integrated circuit (ASIC) or field programmable gate array (FPGA)-based designs. The DNN hardware will have to be redesigned to incorporate the necessary weight perturbations into selected layers required by the mutated model, which will incur non-trivial area, delay and power consumption overheads. The scheme also cannot be easily adapted to third-party DNN intellectual property (IP) cores. For example, Xilinx deep learning processing unit (DPU) IP core is supplied as an encrypted register transfer level (RTL) file [24], with little or no flexibility for any structural modification. As it takes great amount of engineering effort and know-how to design a custom DNN accelerator to produce state-of-the-art results, it is more cost-effective to use the third-party DNN IP cores for many designs but protect them against adversarial example attacks. To realize online adversarial input detection in spatial architectures and third-party DNN IPs, an alternative hardware-oriented mutation technique is required.

In this section, we propose a new method to enable online detection of adversarial examples using existing hardware accelerators with negligible hardware overhead. It utilizes clock glitch as a fault source to emulate the desired mutation effect in the deep learning accelerator. Our hypothesis is that different layers of the DNN accelerator possess different degrees of sensitivity and immunity to glitch injection. To verify the plausibility of this hypothesis, a systematic approach is established to identify the layers that can cause the outputs of an adversarial input to deviate more significantly than its benign input. We utilize the distinct clock domain dedicated for the PE array of the DNN accelerator for glitch injection so that the injected glitches affect only operations scheduled for computation in the PE array at specific clock cycles. Since different deep learning accelerators may have different clock configurations, we use Xilinx DPU implemented in ZCU102 evaluation board as an example to illustrate the clock glitch injection setup for evaluating the label change inconsistency due to the application of different amount of clock glitches at each CNN layer. A clock glitch can be generated by using a pair of normal and phase-shifted clock signals of the accelerator and a control signal *trigger*, as shown in Fig. 4. In Fig. 4, clk is the input clock for general logic such as register configuration, data controller, global buffer and SIMT functional units of the DNN accelerator. $clk_{2\times}$ is the faster input clock to the PE array to accelerate the MAC operations.

Fig. 4: The generation of a clock glitch on the DNN accelerator.

It is set to twice the frequency of *clk*. *clk_{2x_90}* is a 90° phase shifted clock of *clk_{2x}*. *ce* is a clock gating control signal used to turn off *clk_{2x}* when the PE array is in idle mode. By feeding *trigger* into the select input, *clk_{2x}* and *clk_{2x_90}* to the data inputs of a 2-to-1 multiplexer, and the output *clk_{2x_glitch}* of the multiplexer to the PE array, clock glitches can be selectively injected into the target execution cycles of the PE array. The timing information for each layer of computation in the PE array in terms of the number of cycles in *clk_{2x}* can be extracted by a cycle-accurate hardware profiling. The convolutional operations on the input data are executed sequentially by the MAC units layer by layer. The proportion of clock cycles *clk_{r_l}* expended in a layer *l* for each input classification can be approximated by the following formula:

$$clk_{r_l} = \frac{W_l}{U_l \times \sum_{i=1}^L \frac{W_i}{U_i}} \quad (8)$$

where *W*, *U* and *L* are the workload, utilization rate and total number of layers of the DNN model, respectively.

We can monitor *clk_{2x}* to obtain the total number of active clock cycles for each input classification. The starting time and number of clock cycles spent by the PE array on the execution of convolutional operations for each layer can then be derived from *clk_{r_l}*. A margin of one hundred clock cycles, representing approximately 0.01% of the average number of cycles in each layer for this model, at the beginning and end of each layer's execution time is reserved for operational instability to ensure that the glitches introduced do not influence other layers except the layer of interest.

A set of glitch intensities $G(\%) = \{0.1, 0.2, 0.5, 1, 2, 10, 20\}$ are applied to every layer of the VGG16 model [45]. Glitch intensity is computed by $1/T_{glitch}$, where T_{glitch} is the number of *clk_{2x}* cycles between two successive glitches. 292 correctly classified clean images and 292 successfully misclassified adversarial images crafted using FGSM with $\epsilon = 4/255$ are evaluated for each layer-wise mutated model. As can be observed from Table I, the number of clean images that have their predictions altered ($\#LC$) is generally smaller than the number of adversarial images at any glitch intensity for any perturbed layer of VGG16. This result supports the hypothesis proposed in [37] for ImageNet dataset besides MNIST and CIFAR-10, and for

TABLE I: Number of images that have changed their predictions ($\#LC$) upon injecting different intensities of glitches into each layer of VGG16 model.

Layer	Image type	Glitch intensity (%)						
		20	10	2	1	0.5	0.2	0.1
conv1_1	Normal	287	282	161	127	62	26	9
	Adversarial	288	280	222	185	141	81	50
conv1_2	Normal	276	226	88	43	27	4	2
	Adversarial	280	261	178	147	93	52	35
conv2_1	Normal	110	74	23	9	4	1	1
	Adversarial	186	174	107	61	34	21	15
conv2_2	Normal	289	226	36	11	5	3	0
	Adversarial	290	268	125	75	44	30	21
conv3_1	Normal	118	60	12	1	1	0	1
	Adversarial	199	140	74	40	27	23	12
conv3_2	Normal	256	123	23	4	3	2	1
	Adversarial	275	212	112	57	33	27	14
conv3_3	Normal	286	173	25	9	6	3	0
	Adversarial	289	249	121	63	41	25	14
conv4_1	Normal	72	23	7	1	1	0	1
	Adversarial	158	99	52	23	22	19	19
conv4_2	Normal	244	80	15	5	1	1	1
	Adversarial	266	187	87	44	29	20	19
conv4_3	Normal	249	136	25	9	2	1	0
	Adversarial	278	226	116	54	37	24	13
conv5_1	Normal	90	30	7	0	1	1	1
	Adversarial	191	110	49	31	20	17	11
conv5_2	Normal	131	19	8	4	1	1	1
	Adversarial	204	96	56	27	18	19	14
conv5_3	Normal	11	6	4	0	0	0	0
	Adversarial	77	57	46	28	23	13	13
fc6	Normal	291	291	242	50	23	5	0
	Adversarial	291	289	276	151	86	54	25
fc7	Normal	288	54	11	8	3	1	0
	Adversarial	292	158	78	47	37	18	13
fc8	Normal	246	274	50	17	9	2	2
	Adversarial	260	276	147	87	54	25	27

the hardware executable model besides the software model. The proportions of $\#LC$ for normal and adversarial inputs are denoted as LCP_{norm} and LCP_{adv} , respectively. The difference $\Delta LCP = LCP_{adv} - LCP_{norm}$ is plotted in Fig. 5 for different layers of VGG16 at different glitch intensities. It can be clearly seen that the curves of most layers are concave downwards, indicating a possibility for finding an optimal glitch intensity that maximizes the inconsistent label changes between the adversarial and clean samples for each layer. For example, ΔLCP_{conv1_2} increases when the glitch intensity is less than 1%, but declines thereafter. Therefore, the optimal glitch intensity is estimated to be 1% for *conv1_2* layer of VGG16. The reduced inconsistency in label changes after the optimal points implies the enhanced mutation effect on the clean images. ΔLCP of *conv3_1*, *conv4_1*, *conv5_1* and *conv5_3* layers appear to be monotonically increasing with the glitch intensity. This might be due to the constraint of the maximum glitch intensity of 20%. On the other hand, this is also a sign of the inherently higher immunity of these layers against inconsistent label changes of benign and adversarial samples by glitch injections. Furthermore, the curves for

VGG 16

Fig. 5: Comparison of ΔLCP in different layers of VGG16.

conv1_1 and *conv1_2* layers reach their maximum at a lower glitch intensity. This can be explained by the fact that errors due to a small number of glitches for the first two layers are likely to induce more errors as they propagate forward through other layers to the final stage of inference.

C. Proposed Decision Tree Detector

The proposed online detector of adversarial input is implemented as a simple decision tree classifier. The decision tree is constructed by an offline analysis to find good glitch patterns and prediction probability thresholds to maximally discriminate the adversarial and benign training inputs. After the offline analysis, the number of decision nodes of the binary tree classifier and their attributes are fixed. An example of a tree classifier detector trained by an image set of 292 clean images and 292 adversarial images for VGG16 model implemented on Xilinx ZCU102 board is shown in Fig. 6. The decision tree has three different types of non-terminal nodes, designated as original (O), mutated (M) and difference (D), and two terminal nodes, designated as Adversarial (A) and Benign (B). Each non-terminal node $v_{i,j}$ has a probability threshold $\tau_{i,j}$ and two children nodes. Type M and type D nodes have in addition a glitch pattern (l,g) , where l is the target DNN layer of glitch injection and g is the glitch intensity. When an input is presented to a DNN, the hardware accelerated classifier F will determine its unmutated label $pred_org$ and corresponding prediction confidence level p_org . Then, the detector is activated. Starting from the root $v_{1,1}$, the input image will undergo an attribute test in each non-terminal node. The outcome of the test will determine which child node to follow for another test in the next level until either terminal node is reached. The destination terminal node, A or B, determines if the input is adversarial or benign. The attribute test compares a probability against the node threshold. For node type O, the predicted top-1 probability from the unmutated F is used for the comparison. For node type M, the top-1 probability $p_mut_{(l,g)}$ is obtained from the DNN classifier $F_{l,g}$ upon the application of a glitch pattern (l,g) . It is made negative when the output classification $pred_mut$ is different from the initially predicted label $pred_org$ and positive otherwise. For node type D, the probability is obtained from the difference probability $p_diff_{(l,g)} = p_mut_{(l,g)} - p_org$.

Fig. 6: An example of the proposed tree classifier detector for VGG16. Nodes $v_{4,1}$ and $v_{4,3}$ are terminal nodes B, and nodes $v_{3,4}$ and $v_{4,6}$ are terminal nodes A.

To construct the above decision tree for a given DNN hardware accelerator at design time, a set of training images, I , is required. I consists of N correctly classified benign images and N successfully misclassified adversarial images. Let *Layer* be a list of layer names of the DNN and G be a set of glitch intensities. Each image of I is fed into the original model F and all possible mutated models. A mutated model $F_{(l,g)}$ is produced by injecting a glitch pattern (l,g) to F , where $l \in \text{Layer}$ and $g \in G$. $\Delta LCP_{(l,g)}$ is first calculated for each possible glitch pattern (l,g) . Then, a list of candidate glitch patterns Z that can enlarge the difference between the adversarial and normal samples is obtained from the calculated $\Delta LCP_{(l,g)}$ as follows:

$$Z = \{(l,g) | l \in \text{Layer}, g \in G, \text{ and } \Delta LCP_{(l,g)} \geq \min_{l \in \text{Layer}} (\max_{g \in G} \Delta LCP_{(l,g)})\} \quad (9)$$

The reason for selecting the set of glitch patterns that have $\Delta LCP_{(l,g)} \geq \min_{l \in \text{Layer}} (\max_{g \in G} \Delta LCP_{(l,g)})$, instead of glitch patterns that merely maximize $\Delta LCP_{(l,g)}$ layer-wise, is to prevent underfitting.

Initially, all training images in I are assigned to the root node $v_{1,1}$, i.e., $I_{1,1} = I$. Thereafter, the decision tree is constructed by dividing the training image set $I_{i,j}$ into two disjoint subsets, $I_{i+1,2j-1}$ and $I_{i+1,2j}$, to split node $v_{i,j}$ into two children nodes, $v_{i+1,2j-1}$ and $v_{i+1,2j}$. The splitting is determined by the information gain (IG) of the dataset. To compute the IG, the entropy of $I_{i,j}$ is required, which can be calculated as follows:

$$H(I_{i,j}) = -(Pa_{i,j} \times \log_2 Pa_{i,j} + Pb_{i,j} \times \log_2 Pb_{i,j}) \quad (10)$$

where $Pa_{i,j}$ and $Pb_{i,j}$ are the proportions of adversarial and benign images, respectively in $I_{i,j}$, and $Pa_{i,j} + Pb_{i,j} = 1$. It has a maximum value of 1 if $I_{i,j}$ has equal proportion of clean and adversarial images, and a minimum value of 0 if the images in $I_{i,j}$ are all benign or all adversarial.

Given the list of candidate glitch patterns Z , all three types of probabilities p_{org} , $p_{mut(l,g)}$ and $p_{diff(l,g)}$ are calculated for each image in $I_{i,j}$ of a non-terminal node $v_{i,j}$. As each glitch pattern of Z will produce a $p_{mut(l,g)}$ and a $p_{diff(l,g)}$ for each image, there are altogether $2 \times |Z| + 1$ probability histograms. From each histogram h , a threshold value that maximizes the IG is selected as the optimal threshold τ_h to split $I_{i,j}$, i.e.,

$$\tau_h = \arg \max_{\tau \in T} IG(I_{i,j}, h, \tau) = \arg \max_{\tau \in T} (H(I_{i,j}) - P(I_{i,j}|_{p \leq \tau}) \times H(I_{i,j}|_{p \leq \tau}) - P(I_{i,j}|_{p > \tau}) \times H(I_{i,j}|_{p > \tau})) \quad (11)$$

where T is a set of discrete threshold values for a histogram h , $p \in \{p_{org}, p_{mut(l,g)}, p_{diff(l,g)}\}$ denotes a probability value, and $I_{i,j}|_{p \leq \tau}$, $I_{i,j}|_{p > \tau} \subset I_{i,j}$ are two disjoint subsets of images obtained by thresholding the probabilities of $I_{i,j}$ by τ .

Each histogram h_a corresponds to an attribute $a \in A$, where $A = \{O, M_{z_i}, D_{z_i}\} \forall i = 1, 2, \dots, |Z|$ and $z \in Z$. With the optimal threshold τ_a determined from h_a in (11), the attribute $a_{i,j}$ and its corresponding optimal threshold $\tau_{i,j}$ of node $v_{i,j}$ is selected from the one that maximizes the IG as follows:

$$\begin{cases} a_{i,j} = \arg \max_{a \in A} IG(I_{i,j}, h_a, \tau_a) \\ \tau_{i,j} = \tau_a \hat{=} a_{i,j} \end{cases} \quad (12)$$

Fig. 7 shows an example of three probability histograms corresponding to the node attributes O, M and D, respectively. These histograms are obtained by feeding an image set containing 292 clean images and 292 adversarial images into the original model and the mutated model with the glitch injection pattern $z = (fc7, 10)$. The maximum IG and the corresponding optimal threshold are also indicated for each histogram. For this illustration, since the node attribute D has the largest IG, if there is no other glitch injection pattern $z \in Z$ that has larger IG, this non-terminal node $v_{1,1}$ will be set to type D with $z = (fc7, 10)$, and $\tau_{1,1} = -0.453$.

Upon fixing the best attribute $a_{i,j}$ and $\tau_{i,j}$ for a non-terminal node $v_{i,j}$, $I_{i,j}$ will be split into $I_{i+1,2j-1} = I_{i,j}|_{p \leq \tau_{i,j}}$ and $I_{i+1,2j} = I_{i,j}|_{p > \tau_{i,j}}$, and assigned to the newly generated children nodes, $v_{i+1,2j-1}$ and $v_{i+1,2j}$, respectively. If the subset of images assigned to a child node has zero entropy, then the child node will be made the terminal node A if Pa of its assigned image subset is 1 or B if Pa is 0. Otherwise, the child node generated is a non-terminal node, and is annotated with a temporary class variable t to indicate the majority image type of its assigned subset. $t = 1$ if $Pa \geq 0.5$ and 0 otherwise.

To prevent overfitting and limit the depth of the decision tree, a separate set of validation images I_v is used to calculate the correct classification rate. The correct classification rate p_n of I_v is calculated at each iteration as follows:

$$p_n = \frac{TP_n + TN_n}{|I_v|} \quad (13)$$

Algorithm 1 Node Splitting

Function Node_Splitting(Model (F), Attribute list (A), Training image set ($I_{i,j}$), Parent node ($v_{i,j}$), Decision tree (DT)):

```

1: calculate  $H(I_{i,j})$ 
2: if  $H(I_{i,j}) = 0$  // parent node is a terminal node then
3:   if  $Pa_{i,j} == 1$  then
4:      $v_{i,j}.type \leftarrow terminal\_node(A)$ 
5:   else
6:      $v_{i,j}.type \leftarrow terminal\_node(B)$ 
7:   end if
8:   return  $DT$ 
9: end if
10: calculate  $t_{i,j}$ 
11:  $v_{i+1,2j-1} \leftarrow new\_node()$ 
12:  $v_{i+1,2j} \leftarrow new\_node()$ 
13: // initialize  $IG_{max} \leftarrow 0$ 
14: for each  $a \in A$  do
15:    $data_a \leftarrow F_a(I_{i,j})$ 
16:    $IG_a \leftarrow IG(I_{i,j}, data_a, \tau_a)$ 
17:   if  $DT$  is lightGBM then
18:      $data_a \leftarrow F_a(GOSS(I_{i,j}))$ 
19:      $IG_a \leftarrow IG(GOSS(I_{i,j}), data_a, \tau_a)$ 
20:   end if
21:   if  $IG_{max} < IG_a$  then
22:      $IG_{max} \leftarrow IG_a$ 
23:      $a_{i,j} \leftarrow a$ 
24:      $\tau_{i,j} \leftarrow \tau_a$ 
25:   end if
26: end for
27:  $v_{i,j}.type \leftarrow (a_{i,j}, \tau_{i,j}, t_{i,j})$ 
28:  $DT[i+1, 2j-1] \leftarrow v_{i+1,2j-1}$ 
29:  $DT[i+1, 2j] \leftarrow v_{i+1,2j}$ 
30:  $I_{i+1,2j-1} \leftarrow I_{i,j}|_{p(a_{i,j}) \leq \tau_{i,j}}$ 
31:  $I_{i+1,2j} \leftarrow I_{i,j}|_{p(a_{i,j}) > \tau_{i,j}}$ 
32: return  $DT$ 

```

where TP_n and TN_n denote the number of true positives and true negatives, respectively achieved by the constructed decision tree at the n -th iteration. $|I_v|$ represents the total number of images in I_v .

The tree terminates at a depth d_{max} when there is no improvement in p_n , i.e., $p_n - p_{n-1} \leq 0$. Upon termination, every non-terminal node at the depth $d = d_{max}$ will be converted to either a terminal node A or B according to its temporary class t , A if $t = 1$ and B if $t = 0$. Once the training is completed, the deployed decision tree has no leaf node that is non-terminal. All the paths of the decision tree will be terminated at either an adversarial (A) or a benign (B) leaf node.

Algorithm 1 shows the pseudo code of the above described process for splitting a non-terminal node into two children nodes. Two different algorithms can be used for the construction of a decision tree classifier. They are conventional [46] and lightGBM [14] decision trees, as illustrated in Algorithms 2 and 3, respectively. The main difference between them is the tree growth, which is level-wise for the conventional decision tree and leaf-wise for the lightGBM tree. Specifically, the

Fig. 7: An example that illustrates the optimal threshold and the IG for different splitting attributes.

Algorithm 2 Construction of the conventional decision tree

Input: Model (F), Attribute list (A), Training image set (I), Validation image set (I_v), Correct prediction (c_o)

Output: Decision Tree (DT)

```

// initialize decision tree  $DT \leftarrow \{\}$ 
// initialize index  $i, j \leftarrow 1$ 
// initialize iteration  $n \leftarrow 0$ 
// initialize depth  $d \leftarrow 1$ 
// initialize correct classification rate  $p_n \leftarrow 0$ 
// initialize correct classification rate  $p_{n-1} \leftarrow -1$ 
1:  $c_o \leftarrow F(I)$ 
2:  $v_{i,j} \leftarrow \text{new\_node}()$ 
3:  $DT[i, j] \leftarrow v_{i,j}$ 
4:  $I_{i,j} \leftarrow I$ 
5: while  $p_{n-1} < p_n$  do
6:   for each leaf node  $v_{i,j} \in \text{level } i \text{ of } DT$  do
7:      $DT \leftarrow \text{Node\_Splitting}(F, A, I_{i,j}, v_{i,j}, DT)$ 
8:   end for
9:    $d \leftarrow i$ 
10:   $n \leftarrow n + 1$ 
11:   $p_n \leftarrow DT(I_v)$ 
12:   $i \leftarrow i + 1$ 
13: end while
14: return  $DT$ 
  
```

classical decision tree expands all the children nodes that have $H(I_{i,j}) > 0$ at level i , whereas the lightGBM decision tree only expands the leaf node with the maximum delta loss. Delta loss refers to the gradient of the loss function with respect to the output of the currently constructed decision tree. To reduce the search space, lightGBM uses gradient-based one-side sampling (GOSS) for training. It only randomly samples image instances with small gradients while retaining those instances with large gradients. This is reflected in *Lines* 17-20 of Algorithm 1, where a subset $GOSS(I_{i,j})$ of $I_{i,j}$, as opposed to the entire set of $I_{i,j}$, is used to find the best-split attribute for lightGBM. This sampling reduces the computational complexity of the offline decision tree building process. Being more loss conscious, the decision tree generated by lightGBM is generally more unbalanced and deeper than the classical tree.

Algorithm 3 Construction of the lightGBM decision tree

Input: Model (F), Attribute list (A), Training image set (I), Validation image set (I_v), Correct prediction (c_o)

Output: Decision Tree (DT)

```

// initialize decision tree  $DT \leftarrow \{\}$ 
// initialize index  $i, j \leftarrow 1$ 
// initialize iteration  $n \leftarrow 0$ 
// initialize correct classification rate  $p_n \leftarrow 0$ 
// initialize correct classification rate  $p_{n-1} \leftarrow -1$ 
1:  $c_o \leftarrow F(I)$ 
2:  $v_{i,j} \leftarrow \text{new\_node}()$ 
3:  $DT[i, j] \leftarrow v_{i,j}$ 
4:  $I_{i,j} \leftarrow I$ 
5: while  $p_{n-1} < p_n$  do
6:    $(x, y) \leftarrow \text{index\_of\_max\_loss\_leaf}(DT)$ 
7:    $DT \leftarrow \text{Node\_Splitting}(F, A, I_{x,y}, v_{x,y}, DT)$ 
8:    $n \leftarrow n + 1$ 
9:    $p_n \leftarrow DT(I_v)$ 
10: end while
11: return  $DT$ 
  
```

IV. EXPERIMENTAL RESULTS

A. Hardware Setup

The hardware setup for the experiments is shown in Fig. 8. A Xilinx ZCU102 evaluation board for the Zynq UltraScale+ XCZU9EG-2FFVB1156E Multiprocessor System-on-Chip (MPSoC) is used to emulate the edge device. The ZU9EG chip contains a quad-core ARM CortexTM-A53 processor, 600K logic cells and 2520 DSP slices [24]. The hardware executable DNN model is implemented on the DPU dedicated for CNN acceleration. The DNN model is compiled by the DNN Development Kit (DNNDK) [47]. The toolchain truncates the 32-bit floating point weights and activations of the DNN to 8-bit fixed-point values without compromising the classification accuracy of the model. The blocks shaded in gray are the circuitries required for glitch injection. The clock signal to the DSP slices (the PE array) of the DPU is provided by the global clock buffer BUFGCTRL, which selects between dsp_clk and dsp_clk_shift to dsp_clk_sel . dsp_clk is generated from dpu_clk via the Clocking Wizard. dsp_clk_shift is the 90° phase shifted version of dsp_clk . The frequency of dpu_clk is set to 100MHz while the frequency of dsp_clk and

Fig. 8: Experimental setup for mutation control of DNN implemented on Xilinx ZCU102 evaluation board.

dsp_clk_shift is set to 200MHz. The control signal sel_sig of BUFCTRL is produced by en_glitch and $trigger$. When $en_glitch = 0$, the PE array is clocked by dsp_clk , and the inference is produced by the unmutated DNN model. When $en_glitch = 1$, the PE array runs on dsp_clk_shift , which is dsp_clk with glitches injected when $trigger$ is active. The glitch intensity and injection time are encoded in the $trigger$ signal. Both en_glitch and $trigger$ signals are configurable by the Extended Multiplexed IO (EMIO) pins of the ARM CortexTM-A53 processor (Zynq UltraScale+ block). The configurations of the $trigger$ signals are given by the node attributes of the proposed decision tree detectors implemented on the ARM CortexTM-A53 processor with a clock frequency of 1.2GHz. This ARM processor is a hard IP core that is monolithically integrated into the FPGA chip in which the target DNN is implemented. The program codes are protected against malicious modifications by the security enclave of Zynq UltraScale+ MPSoC. Its memory and peripherals are protected with the Xilinx memory protection unit (XMPU) and the Xilinx peripheral protection unit (XPPU) [48]. Data communication and control between the host PC and the FPGA board are carried out via the UART and Ethernet interfaces.

B. Datasets and Models

ImageNet [49] provided by the ImageNet Large Scale Visual Recognition Challenge 2012 for image classification tasks is used as the input dataset. It contains 1.2 million training images and 50,000 validation images. All of them are human-annotated with 1000 classes. Two models with different DNN architectures are used for the evaluation: VGG16 [45] and ResNet50 [50]. To avoid bias in the training process, the training set consists of an equal number of clean and adversarial images. Table II shows the number of training and testing images and the algorithms used for the generation of their adversarial examples for each DNN model. #Initial refers to the number of clean images initially selected for crafting the adversarial examples. #Clean and #Tampered refer to the number of correctly classified clean images and their corresponding successfully misclassified adversarial images,

TABLE II: Training and testing datasets for VGG16 and ResNet50.

Model	Dataset	Attacks	#Initial	#Clean	#Tampered
VGG16	Training	FGSM ($\epsilon = 4/255$)	700	452	452
		I-FGSM ($\epsilon = 4/255$)	350	241	241
	Testing	FGSM ($\epsilon = 4/255$)	500	295	295
		FGSM ($\epsilon = 8/255$)	250	163	163
		I-FGSM ($\epsilon = 4/255$)	250	161	161
		C&W ($\kappa = 20$)	300	185	185
		UP	400	264	264
ResNet50	Training	FGSM ($\epsilon = 4/255$)	600	320	320
		I-FGSM ($\epsilon = 28/255$)	400	320	320
	Testing	FGSM ($\epsilon = 4/255$)	430	230	230
		FGSM ($\epsilon = 8/255$)	420	230	230
		I-FGSM ($\epsilon = 4/255$)	360	230	230
		I-FGSM ($\epsilon = 28/255$)	320	230	230
		C&W ($\kappa = 20$)	550	391	391
		UP	320	230	230

respectively. For each model, the same training set is used for the construction of the two proposed decision tree detectors. 75% of them are used for the training and 25% of them are used for the validation.

The reason for adding I-FGSM-eps4 (I-FGSM with $\epsilon = 4/255$) images to the training dataset for VGG16 is because the prediction probabilities for adversarial images generated by FGSM-eps4 (FGSM with $\epsilon = 4/255$), FGSM-eps8 (FGSM with $\epsilon = 8/255$) and UP are found to be much lower than those generated by I-FGSM-eps4 and C&W, as shown in the box-and-whisker plot in Fig 9, where the $\square$, $-$ and $\times$ symbols denote the mean, maximum and minimum values, and 1 and 99 percentiles, respectively. Most of the adversarial samples generated by FGSM-eps4, FGSM-eps8 and UP tend to lie near the decision boundaries while most of the adversarial samples generated by I-FGSM-eps4 and C&W lie far from the decision boundaries. Therefore, if only FGSM-eps4 is considered in the training process, the solution will tend to pick glitch injections with smaller mutations, which have almost no influence on the high-confidence adversarial images. Similarly, if only I-FGSM-eps4 is considered, the solution will tend to pick glitch injections with larger mutations, which can distort the low-confidence clean samples. For the detection of adversarial inputs with different confidence levels, the training set should cover both mutation strategies. Similarly, for ResNet50, the adversarial images generated by I-FGSM-eps28 (I-FGSM with $\epsilon = 28/255$) are added to the training set. By comparing the box-and-whisker plots in Fig. 10 for ResNet50 and Fig. 9 for VGG16, C&W and I-FGSM are able to generate adversarial examples with much higher confidence for VGG16 than ResNet50 under the same perturbation parameter setting. This implies that VGG16 is more vulnerable to the C&W and I-FGSM attacks than ResNet50. The tree depth, number of non-terminal nodes, non-terminal node types, glitch patterns and threshold values in the first three levels of the trained classical and lightGBM detectors are summarized in Table III. Glitch patterns, ($fc7, 10$) and ($fc8, 2$), have the largest ΔLCP for VGG16 in Fig. 5. As expected, they appear most frequently in the mutation tests of both decision tree detectors for VGG16.

Fig. 9: Box-and-whisker plot of the top-1 probabilities for the clean images and the adversarial images generated by various attacks for VGG16

Fig. 10: Box-and-whisker plot of the top-1 probabilities for the clean images and the adversarial images generated by various attacks for ResNet50

C. Results

Table IV shows the results of the two proposed decision trees on unseen adversarial inputs presented to the hardware accelerated VGG16 and ResNet50. The results are compared in terms of TPR (also known as detection or recall rate) and FPR, which are the acronyms for true positive rate and false positive rate, respectively. They are computed by:

$$TPR(Recall) = \frac{TP}{TP + FN} \quad (14)$$

$$FPR = \frac{FP}{FP + TN} \quad (15)$$

where TP, TN, FP and FN are the acronyms for true positive, true negative, false positive and false negative, respectively. In the context of adversarial input detection, an adversarial sample is a positive and a benign sample is a negative. Therefore, a TP refers to the correct detection of an adversarial sample; a TN refers to the correct classification of a benign image; a FP refers to a benign image being wrongly recognized as adversarial; and a FN refers to an adversarial input being wrongly interpreted as benign.

TABLE III: Parameters of the final classical and lightGBM decision trees (DTs) for VGG16 and ResNet50.

Model	DTs	Depth	#Nodes	Node Type	τ
VGG16	Classical	2	3	$v_{1,1}: D_{(fc7,10)}$	-0.579
				$v_{2,1}: D_{(fc8,2)}$	-0.023
				$v_{2,2}: D_{(fc8,2)}$	-0.314
	LightGBM	5	20	$v_{1,1}: D_{(fc7,10)}$	-0.580
				$v_{2,1}: D_{(fc8,2)}$	0.321
				$v_{2,2}: D_{(fc8,2)}$	0.312
				$v_{3,1}: D_{(conv4_1,20)}$	-1.065
				$v_{3,2}: D_{(fc8,2)}$	0.021
				$v_{3,3}: M_{(fc8,2)}$	0.070
				$v_{3,4}: M_{(fc7,10)}$	-0.009
ResNet50	Classical	6	17	$v_{1,1}: D_{(res2a_branch2a,0.2)}$	0.111
				$v_{2,1}: M_{(res2a_branch2b,0.2)}$	0.797
				$v_{2,2}: D_{(res5c_branch2a,2)}$	0.079
				$v_{3,1}: M_{(res2c_branch2a,20)}$	-0.244
				$v_{3,2}: D_{(res5a_branch2b,0.2)}$	-0.149
	LightGBM	4	9	$v_{3,3}: M_{(res2a_branch2a,10)}$	0.244
				$v_{1,1}: D_{(res2a_branch2c,0.1)}$	0.075
				$v_{2,1}: M_{(res2b_branch2a,20)}$	-0.151
				$v_{2,2}: D_{(res4a_branch2c,2)}$	0.026
				$v_{3,1}: D_{(res2b_branch2a,20)}$	-0.625
				$v_{3,2}: D_{(res2b_branch2a,20)}$	-1.145
				$v_{3,4}: D_{(res2c_branch2a,2)}$	0.069

The results show that LightGBM has a much lower FPR than the classical tree detector for all the attacks on both models. This is owing to the inherent loss control of the lightGBM training algorithm. For ResNet50, lightGBM achieves higher TPR than the classical tree detector for all the attacks. For VGG16, the classical decision tree algorithm achieves approximately 23.7%, 16.7% and 8.4% higher detection rates than lightGBM for the FGSM-eps4, FGSM-eps8 and UP attacks, respectively. However, the detection rates of lightGBM are significantly higher for the I-FGSM-eps4 and C&W attacks. They are increased by nearly 75% and 389.1%, respectively compared with the classical detector. This shows that the mutation strategies chosen by level-wise growth are more effective in discriminating the low-confidence adversarial examples but not the high-confidence adversarial examples like the I-FGSM-eps4 and C&W attacks for VGG16. The difference between the level-first and best-first strategies can be observed from their different improvements in TPR from the initial TPR after the first attribute test at the root node computed based on the temporary classes. The final TPR of the classical tree detector for VGG16 has been increased by nearly 30.5% and 154.5% for the I-FGSM and C&W attacks, respectively, but remains unchanged for the FGSM and UP attacks. On the contrary, the final TPR of the lightGBM detector for VGG16 has been increased by approximately 22.6%, 50%, 87.5%, 356.5% and 13.6% from the TPR of the root node for the FGSM-eps4, FGSM-eps8, I-FGSM-eps4, C&W and UP attacks, respectively. Similarly, the final TPR of the classical tree detector for ResNet50 has been increased by nearly 37.1%, 35.7%, 41%, 65.5%, 241.8% and 6% from the TPR of the root node for the FGSM-eps4, FGSM-eps8, I-FGSM-eps4, I-FGSM-eps28, C&W and UP attacks, respectively. The final TPR of the lightGBM detector for ResNet50 has been increased by nearly 31.7%, 31.7%, 26.1%, 55.7%, 160.9% and 6.9% from the TPR of the root node for the FGSM-eps4, FGSM-eps8, I-FGSM-eps4, I-FGSM-eps28, C&W and UP attacks, respectively. For both models, both

TABLE IV: Comparison of classical and lightGBM tree detectors in true positive rates (TPRs) and false positive rates (FPRs) for VGG16 and ResNet50.

Models	DTs	Depth	Attacks	FPR (%)	TPR (%)
VGG16	Classical	2	FGSM($\epsilon = 4/255$)	21.36	79.66
			FGSM($\epsilon = 8/255$)	18.40	77.30
			I-FGSM($\epsilon = 4/255$)	18.63	37.27
			C&W($\kappa = 20$)	13.51	15.14
			UP	15.91	92.80
	LightGBM	5	FGSM($\epsilon = 4/255$)	11.53	64.41
			FGSM($\epsilon = 8/255$)	5.52	66.26
			I-FGSM($\epsilon = 4/255$)	6.83	65.22
			C&W($\kappa = 20$)	10.81	74.05
			UP	4.55	85.61
ResNet50	Classical	6	FGSM($\epsilon = 4/255$)	6.09	85.22
			FGSM($\epsilon = 8/255$)	6.97	84.35
			I-FGSM($\epsilon = 4/255$)	3.91	88.26
			I-FGSM($\epsilon = 28/255$)	5.65	81.30
			C&W($\kappa = 20$)	3.84	79.54
			UP	4.35	91.74
	LightGBM	4	FGSM($\epsilon = 4/255$)	0.43	92.17
			FGSM($\epsilon = 8/255$)	0.43	93.91
			I-FGSM($\epsilon = 4/255$)	0	90.43
			I-FGSM($\epsilon = 28/255$)	0.43	88.70
			C&W($\kappa = 20$)	0	92.07
			UP	0.87	93.91

detectors have a greater propensity to select mutation strategies to discriminate the low-confidence adversarial examples before the high-confidence adversarial examples. This corroborates the necessity to include high-confidence adversarial inputs in the training set for the decision tree construction. It is also observed that both proposed decision tree detectors are most effective against the UP attack regardless of the model complexity. UP is input-agnostic. It is the most viable non-invasive attack to fool deep learning models executed on hardware accelerator platform.

Both detectors for ResNet50 have much lower FPRs and higher TPRs than VGG16 on all attacks. This implies that the proposed method works better for deeper neural networks on the same image classification application. Since the glitch patterns are derived and applied layer-wise, a deeper network provides more choices for optimizing the effective glitch injection patterns to discriminate the adversarial examples from the benign inputs in building the decision tree detector.

Table V lists the hardware overhead of the proposed detector. No additional LUTRAM, BRAM and DSP are required for the glitch injection circuit other than four additional architecture-independent global clock buffers (BUFGs) and a negligibly small number of LUTs and FFs. The trained decision tree detector is implemented on the ARM CortexTM-A53 processor. Without specific code optimization, the code sizes of the classical tree detector are ~ 16 KB and ~ 85 KB for VGG16 and ResNet50, respectively; and the code sizes of the lightGBM detector are ~ 94 KB and ~ 50 KB for VGG16 and ResNet50, respectively. The final tree detector can also be implemented as a small finite-state machine and configured onto the same FPGA chip as the deep learning accelerator. A mixture of 100 clean images and adversarial images generated by various attacks are used for assessing the processing time. Due to the complexity of VGG16, the original design takes 0.13s on average to complete the inference of one image. The sanity check of the inference can be completed by the classical and lightGBM tree detectors in 0.28s and 0.38s, respectively.

TABLE V: Hardware overhead of DNN with proposed clock glitch mutation control

Resource	Original design	With Glitch Control	Overhead
LUT	108048	108114	0.06 %
LUTRAM	10939	10939	0 %
FF	203051	203095	0.02 %
BRAM	518	518	0 %
DSP	1394	1394	0 %
BUFG	9	13	44 %
MMCM	1	1	0 %

For ResNet50, the original design takes 34.5ms on average to make an inference of one image. The sanity check can be completed by the classical and lightGBM tree detectors in 78ms and 68.8ms, respectively. Since each attribute test of the detector involves merely checking a prediction probability against a threshold, the latency is determined mainly by the time required to process an input image by the DNN for all the attribute tests with distinct glitch patterns. Furthermore, the decision tree is not balanced. The longest branch may not be exercised for many images. Also, the prediction probability does not need to be re-computed by the CNN if the node type is O or the glitch pattern of type M or D node has already been applied. Therefore, the speed ratios $\frac{Speed_{Classical}}{Speed_{Original}}$ and $\frac{Speed_{LightGBM}}{Speed_{Original}}$ are much smaller than the depths of the decision trees. In our experiments, the clock frequency of the DSP slices of the DPU is set to 200MHz to conserve power. It can be increased to 600MHz without affecting the prediction accuracy to accelerate the inference with and without the sanity check by a factor of 3. Recently, a new hardware-oriented CNN compression strategy [51] was reported to achieve 83.0 frames per second (FPS) for VGG16 implemented on a higher capacity and performance Virtex@UltraScale+TM FPGA VCU118 board. For this throughput rate of VGG16 in Xilinx VCU118, ~ 27 FPS is envisaged with the same sanity check. Since our current CNN implementation consumes $3 \times$ less FPGA resources than that of [51], there is room in VCU118 to add a shadow CNN of our implementation for computing the prediction probability of an input when a new mutation is required. This way there is no need to stall the input image stream of the original CNN as the detector can check every alternate frame for ResNet50 and once every three frames for VGG16 using the shadow CNN.

D. Comparisons and Discussions

Table VI compares the proposed lightGBM detector for VGG16 with several latest fully software-based adversarial input detection methods. All these existing methods require computationally intensive off-line analyses that are not amendable for edge device implementation, particularly for Xu2017 [29] and Liang2018 [39]. Both methods rely on comparing the prediction results between the original input and its filtered input for adversarial sample detection. These input filters require compute-intensive division operations. It should be noted that the results reported in Table VI for Xu2017 [29]

TABLE VI: Comparison of the proposed lightGBM detector for VGG16 and existing adversarial sample detection methods.

Metrics	Wang2020 [11]	Wang2019 [10]	Liang2018GPU [39]	Liang2018FPGA [39]	Xu2017 [29]	This work
Implementation	GPU	GPU	GPU	FPGA	GPU	FPGA
Defense Category	Sample Statistics	Prediction Inconsistency	Prediction Inconsistency	Prediction Inconsistency	Prediction Inconsistency	Prediction Inconsistency
Method	Fired Neuron Rate	Weight Mutation	Adaptive Noise Reduction	Adaptive Noise Reduction	Feature Squeezing	Glitch Injection
Dataset	ImageNet	ImageNet	ImageNet	ImageNet	ImageNet	ImageNet
Model	(8-bit quantized) VGG16	(8-bit quantized) VGG16	(32-bit) VGG16	(8-bit quantized) VGG16	(32-bit) MobileNet / (32-bit) VGG16	(8-bit quantized) VGG16
TPR (%)	FGSM: N.A.	N.A.	65.73	60.92	43.40 / 22.86	65.34
	I-FGSM: N.A.	N.A.	53.62	42.86	64.40 / 35.47	65.22
	C&W: N.A.	N.A.	67.31	55.14	97.93 / 89.33	74.05
	UP: 91.90	82.00	90.41	92.80	78.60 / 74.41 (DeepFool)	85.61
FPR (%)	FGSM: N.A.	N.A.	37.08	33.41	5.37	8.53
	I-FGSM: N.A.	N.A.	18.84	16.15	5.60	6.83
	C&W: N.A.	N.A.	19.23	14.59	8.33 (overall) 5.15	10.81
	UP: 6.70	5.67	47.95	34.47	5.24 (DeepFool)	4.55

are evaluated on MobileNet, which is 176 times smaller than the VGG16 we evaluated. Besides, except an overall FPR, the FPR for each individual attack is not reported in Xu2017 [29]. For a fair comparison, the evaluation results of retargeting the Xu2017 [29] algorithm to the 32-bit floating-point VGG16 model for the FGSM-eps4, FGSM-eps8, I-FGSM-eps4, C&W ($\kappa = 0$) and DeepFool attacks reported by [35] are also listed in Table VI. It is observed that the TPR of Xu2017 has degraded for all the attacks when the same method is evaluated on a larger model. In particular, its original TPRs for the FGSM and I-FGSM attacks have been reduced by 47.3% and 44.9%, respectively on VGG16. This is because larger perturbation strengths are adopted in [35] to produce the adversarial examples for these two attacks compared with the original evaluation of Xu2017. It can be argued that the TPR will be worse for our C&W adversarial examples with $\kappa = 20$. Comparing to the results of Xu2017 on VGG16, our detector has increased the TPR significantly by 185.8%, 83.9% and 15.1% for the FGSM, I-FGSM and DeepFool-originated (including DeepFool and UP) attacks, respectively. Our work also has a lower FPR for the DeepFool-originated attack. The results of Xu2017 are expected to be worse if the DNN model is implemented on hardware-accelerated platform due to the fixed-point arithmetic and quantization.

The results reported in Liang2018 [39] are based on three classes of ImageNet images and the 32-bit floating-point inception models in GPU implementation. For a fair comparison, we retarget Liang2018's algorithm to a 32-bit floating-point VGG16 model and test it with our images randomly selected from 1000 classes of ImageNet dataset. The retarget results are shown under the column 'Liang2018GPU' in Table VI. We also retarget Liang2018 [39] to our DPU-accelerated 8-bit quantized VGG16 model implemented on FPGA. Unfortunately, the Python code provided by [39] to construct the filter is too complex to be ported onto the ARM A53 processor in our evaluation platform. Therefore, this code is run on a PC equipped with an E5-1630 v4 3.70GHz CPU, 16GB system memory, and a GeForce GTX 1070Ti to produce the filtered test images. Both the original and corresponding filtered test images are then fed into our deep learning accelerator for inference. The detection results so obtained are listed under the column 'Liang2018FPGA' in Table VI. As expected, the TPR and FPR of Liang2018 have degraded for all the attacks when the same method is evaluated on a larger model with

TABLE VII: Comparison of the largest scale of DNN evaluated between the proposed lightGBM detector and DeepFense [13].

	Input Size	#Model Weight	Classification Scale
This Work	224×224×3	138M	1000
DeepFense [13]	32×32×3	1.37M	10

more different classes of test images. In particular, the TPRs for the FGSM and C&W attacks have been reduced from those of Liang2018 by 21.6% and 32.7%, respectively for Liang2018GPU, and by 27.3% and 44.9%, respectively for Liang2018FPGA. This is because the adversarial images used in Liang2018 [39] fall in the low-confidence region. Their adversarial inputs are produced with weaker adversarial perturbation strengths (FGSM-eps1, FGSM-eps2, C&W with $\kappa \in [0, 4]$). Our test images include both the low-confidence and the high-confidence adversarial inputs. Consequently, the TPRs of Liang2018GPU and Liang2018FPGA for the UP attack are not affected as significantly as it is for the C&W ($\kappa = 20$ for our test images) attack. Comparing with Liang2018FPGA, our detector has approximately 52.2% and 34.3% higher TPR for I-FGSM and C&W, respectively. More importantly, the FPRs of our detector are significantly lower than those of Liang2018FPGA by 74.5%, 57.7%, 25.9% and 86.8% for FGSM, I-FGSM, C&W and UP, respectively. Comparing to Wang2020 [11] and Wang2019 [10], our detector has a lower FPR. Our TPR is higher than that of Wang2019 for the UP attack. Unlike our proposed tree detector, Wang2020's and Wang2019's methods consider only the UP attack but not the FGSM, I-FGSM and C&W attacks.

Two hardware-oriented adversarial input detection schemes, namely DeepFense [13] and DNNGuard [12], have been reported recently. DeepFense [13] was only evaluated on small-scale (10 classes) classification problem with very small input data size ($\leq 32 \times 32$). Table VII shows the differences between our work and DeepFense in terms of the input image size, model size and classification scale. Our method has been evaluated on DNN classification of model size and class scale up to $100\times$ larger than those of [13]. Since the scale of DeepFense is largely dependent on the classification complexity and model size, its effectiveness and efficiency are expected to drop substantially when it is scaled up for 1000-class applications and larger models. DNNGuard [12] is an ASIC design in 65nm CMOS technology based on the

TABLE VIII: Comparison of hardware overhead between the proposed lightGBM detector and DNNGuard [12].

	Implementation	LUT Overhead (%)	FF Overhead (%)
This Work	Physical	0.06	0.02
DNNGuard [12]	Simulated	> 8.1	> 5.3

NVIDIA Deep Learning Accelerator (NVDLA) architecture. Strictly speaking, DNNGuard is preached as a deep learning accelerator architecture that can simultaneously run the victim and existing detection algorithms/networks rather than a new detection method. There is no validation against adversarial input detection accuracy but only simulated hardware figures of merit reported in [12]. The reported hardware overhead is based on the synthesized design of DNNGuard against the non-optimized two-thread parallel computing architecture. Table VIII shows the comparison of the mutation circuit overhead of our proposed lightGBM in FPGA implementation and the reported hardware overhead of DNNGuard [12]. Compared with DNNGuard, the LUT (Combinational cell) and register (Sequential cell) overheads of our proposed detector are at least $135\times$ and $265\times$ lower, respectively. Taking into consideration that the results reported in [12] are obtained by pre-layout simulation, the actual performance and hardware area of DNNGuard will be worse.

V. CONCLUSION

This paper presents the design methodology of a new lightweight adversarial input detector for large-scale DNN application. As an exemplification, the proposed detectors for VGG16 and ResNet50 were implemented on the DPU of a Xilinx Zynq UltraScale+ MPSoC. It is demonstrated that with very limited manipulable space allowed by the DNN accelerator and without modifying the commercial DPU IP, clock glitches can be exploited advantageously to discriminate adversarial inputs from clean inputs by introducing biased computational disruptions in selected convolutional layers. A tree detector can be trained to determine the optimal intensity and timing of glitches and the probability threshold to apply at each node. The accuracy of the proposed detector has been evaluated by applying four prevalent attacks on randomly selected images from ImageNet dataset. A significant improvement in detection rate by up to 185.8%, 83.9%, 34.3% and 15.1%, and a substantial reduction in false positive rate by up to 74.5%, 57.7%, 25.9% and 86.8% for FGSM, I-FGSM, C&W and UP, respectively, have been achieved over existing software-oriented approaches. With only 0.06% of LUT and 0.02% of FF overheads and a compact detector code size of less than 100 KB for a thousand-class DNN, this is perhaps the smallest adversarial input detector that has ever been physically demonstrated on a FPGA edge computing platform.

REFERENCES

- [1] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.
- [2] Y. Bengio, "Learning deep architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, Jan. 2009. [Online]. Available: <http://dx.doi.org/10.1561/22000000006>
- [3] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proc. IEEE Conf. Comput. Vision Pattern Recogn. Workshops*, Columbus, Ohio, USA, Jun. 2014, pp. 682–687.
- [4] Y. Chen *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proc. 47th Annual IEEE/ACM Int. Symposium Microarchitecture*, Cambridge, UK, Dec. 2014, pp. 609–622.
- [5] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.
- [6] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.
- [7] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 11, pp. 5784–5789, 2018.
- [8] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *IEEE Access*, vol. 6, pp. 14410–14430, 2018.
- [9] C. Szegedy *et al.*, "Intriguing properties of neural networks," presented at Int. Conf. Learning Representations (ICLR), Scottsdale, Arizona, USA, May 2013. [Online]. Available: <https://arxiv.org/abs/1312.6199>
- [10] S. Wang, W. Liu, and C. H. Chang, "Detecting adversarial examples for deep neural networks via layer directed discriminative noise injection," in *Proc. 2019 Asian Hardware Oriented Secur. Trust Symposium (AsianHOST)*, Xi'an, China, Dec. 2019, pp. 1–6.
- [11] —, "Fired neuron rate based decision tree for detection of adversarial examples in dnns," in *Proc. 2020 IEEE Int. Symp. Circuits Systems (ISCAS-2020)*, Oct. 2020.
- [12] X. Wang *et al.*, "DNNGuard: An elastic heterogeneous dnn accelerator architecture against adversarial attacks," in *Proc. 25th Int. Conf. Architectural Support Programming Languages Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland, Mar. 2020, pp. 19–34.
- [13] B. D. Rouhani, M. Samragh, M. Javaheripi, T. Javidi, and F. Koushanfar, "Deepfense: Online accelerated defense against adversarial deep learning," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD'18)*, San Diego, California, Nov. 2018. [Online]. Available: <https://doi.org/10.1145/3240765.3240791>
- [14] G. Ke *et al.*, "LightGBM: A highly efficient gradient boosting decision tree," in *Advances Neural Information Processing Systems* 30, 2017, pp. 3146–3154. [Online]. Available: <http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf>
- [15] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," presented at Int. Conf. Learning Representations (ICLR), San Diego, CA, USA, May 2014. [Online]. Available: <http://arxiv.org/abs/1412.6572>
- [16] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," in *Proc. Int. Conf. Learning Representations (ICLR)*, Toulon, France, Apr. 2017.
- [17] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Proc. 2017 IEEE Symp. Security Privacy (SP)*, San Jose, CA, USA, May 2017, pp. 39–57.
- [18] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, "Universal adversarial perturbations," in *Proc. IEEE Conf. Comp. Vision Pattern Recogn.*, Honolulu, Hawaii, USA, Jul. 2017, pp. 1765–1773.
- [19] W. Zhang, "Shift-invariant pattern recognition neural network and its optical architecture," in *Proc. annual conf. Japan Society Applied Physics*, Japan, May 1988.
- [20] W. Zhang, K. Itoh, J. Tanida, and Y. Ichioka, "Parallel distributed processing model with local space-invariant interconnections and its optical architecture," *Applied Optics*, vol. 29, no. 32, pp. 4790–4797, 1990.
- [21] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 2017 ACM/IEEE 44th Annu. Int. Symp. Comput. Architecture (ISCA)*, Toronto, ON, Canada, Jun. 2017, pp. 1–12. [Online]. Available: <https://doi.org/10.1145/3079856.3080246>
- [22] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *Proc. 2014 IEEE Int. Solid-State Circuits Conf. Digest Technical Papers (ISSCC)*, San Francisco, CA, USA, Feb. 2014, pp. 10–14.
- [23] B. Moons and M. Verhelst, "An energy-efficient precision-scalable convnet processor in 40-nm cmos," *IEEE J. Solid-State Circuits*, vol. 52, no. 4, pp. 903–914, Apr. 2017.
- [24] Xilinx, *DPU for Convolutional Neural Network v3.0*.
- [25] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proc. IEEE Conf.*

- Computer Vision Pattern Recogn.*, Las Vegas, Nevada., USA, Jun. 2016, pp. 2574–2582.
- [26] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman, “Sok: Security and privacy in machine learning,” in *Proc. 2018 IEEE European Symp. Secur. Privacy (EuroSP)*, London, UK, Apr. 2018, pp. 399–414.
- [27] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, “Distillation as a defense to adversarial perturbations against deep neural networks,” in *Proc. 2016 IEEE Symp. Security Privacy (SP)*, San Jose, CA, USA, May 2016, pp. 582–597.
- [28] N. Papernot *et al.*, “Practical black-box attacks against machine learning,” in *Proc. 2017 ACM Asia Conf. Comp. Comm. Security*, Abu Dhabi, United Arab Emirates, 2017 Apr., pp. 506–519.
- [29] W. Xu, D. Evans, and Y. Qi, “Feature squeezing: Detecting adversarial examples in deep neural networks,” presented at 2018 Net. Distributed Syst. Security Symp. (NDSS), San Diego, CA, USA, Feb. 2018. [Online]. Available: <https://arxiv.org/abs/1704.01155>
- [30] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, “On detecting adversarial perturbations,” presented at 5th Int. Conf. Learning Representations (ICLR), Toulon, France, Apr. 2017. [Online]. Available: <https://arxiv.org/abs/1702.04267>
- [31] J. Wang, J. Sun, P. Zhang, and X. Wang, “Detecting adversarial samples for deep neural networks through mutation testing,” 2018. [Online]. Available: <http://arxiv.org/abs/1805.05010>
- [32] A. N. Bhagoji, D. Cullina, C. Sitawarin, and P. Mittal, “Enhancing robustness of machine learning systems via data transformations,” in *Proc. 2018 52nd Annual Conf. Info. Sci. Syst. (CISS)*, Princeton, New Jersey, USA, Mar. 2018, pp. 1–5.
- [33] N. Carlini and D. Wagner, “Magnet and ”efficient defenses against adversarial attacks” are not robust to adversarial examples,” 2017. [Online]. Available: <https://arxiv.org/abs/1711.08478>
- [34] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner, “Detecting adversarial samples from artifacts,” 2017. [Online]. Available: <https://arxiv.org/abs/1703.00410>
- [35] J. Liu *et al.*, “Detection based defense against adversarial examples from the steganalysis point of view,” in *Proc. IEEE Conf. Comput. Vision Pattern Recognition, CVPR 2019, Long Beach, CA, USA, Jun. 16-20, 2019*, 2019, pp. 4825–4834.
- [36] K. Grosse, P. Manoharan, N. Papernot, M. Backes, and P. McDaniel, “On the (statistical) detection of adversarial examples,” 2017. [Online]. Available: <https://arxiv.org/abs/1702.06280>
- [37] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang, “Adversarial sample detection for deep neural network through model mutation testing,” in *Proc. 41st Int. Conf. Software Engineering*, QC, Canada, May 2019, pp. 1245–1256.
- [38] L. Ma *et al.*, “Deepmutation: Mutation testing of deep learning systems,” in *Proc. 2018 IEEE 29th Int. Symp. Software Reliability Engineering (ISSRE)*, Memphis, TN, USA, Oct. 2018, pp. 100–111.
- [39] B. Liang *et al.*, “Detecting adversarial image examples in deep neural networks with adaptive noise reduction,” *IEEE Trans. Dependable Secure Comput.*, May 2018.
- [40] M. Agoyan, J.-M. Dutertre, D. Naccache, B. Robisson, and A. Tria, “When clocks fail: On critical paths and clock faults,” in *Proc. Smart Card Res. Adv. Appl.*, Berlin, Heidelberg, Apr. 2010, pp. 182–193.
- [41] W. Liu, C. H. Chang, F. Zhang, and X. Lou, “Imperceptible misclassification attack on deep learning accelerator by glitch injection,” in *Proc. 57th Annu. Des. Automat. Conf. (DAC)*, Jul. 20-24, 2020.
- [42] W. Hua, Z. Zhang, and G. E. Suh, “Reverse engineering convolutional neural networks through side-channel information leaks,” in *Proc. 2018 55th ACM/ESDA/IEEE Design Automation Conf. (DAC)*, San Francisco, California, USA, Jun. 2018, pp. 1–6.
- [43] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Stealing machine learning models via prediction APIs,” in *Proc. 25th USENIX Security Symp. (USENIX Security 16)*, Vancouver, BC, Canada, Aug. 2016, pp. 601–618.
- [44] M. Fredrikson, S. Jha, and T. Ristenpart, “Model inversion attacks that exploit confidence information and basic countermeasures,” in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, Denver, Colorado, USA, Oct. 2015, pp. 1322–1333.
- [45] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” 2014. [Online]. Available: <https://arxiv.org/abs/1409.1556>
- [46] S. R. Safavian and D. Landgrebe, “A survey of decision tree classifier methodology,” *IEEE Trans. systems man cybernetics*, vol. 21, no. 3, pp. 660–674, May/Jun. 1991.
- [47] Xilinx, *DNNDK User Guide*.
- [48] —, *Isolation Methods in Zynq UltraScale+ MPSoCs Application Note*.
- [49] J. Deng *et al.*, “ImageNet: A large-scale hierarchical image database,” in *Proc. IEEE Comput. Vision Pattern Recognit. (CVPR)*, Miami, FL, USA, Jun. 20-21, 2009, pp. 248–255.
- [50] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, Las Vegas, NV, USA, Jun. 2016, pp. 770–778.
- [51] T. Yuan, W. Liu, J. Han, and F. Lombardi, “High performance cnn accelerators based on hardware and algorithm co-optimization,” *IEEE Trans. Circuits Syst. I: Regular Papers*, vol. 68, no. 1, pp. 250–263, Jan. 2021.

Si Wang (S'19) received the B.Eng. (First Class Honor) degree from Nanyang Technological University (NTU) in Singapore in 2016. She is currently pursuing the Ph.D degree with the School of Electrical and Electronic Engineering (EEE) of NTU. Her current research interests include machine learning accelerator and machine learning security.

Wenye Liu (S'17) received the B.S. degree in microelectronics from Shenzhen University, China, in 2014, the B.S. degree in physics from Umea University, Sweden, in 2014, and the M.S. degree in IC design engineering from Hong Kong University of Science and Technology. He is currently pursuing the Ph.D degree with the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. His current research interests include hardware security, machine learning accelerator and fault injection attack.

Chip-Hong Chang (S'92-M'98-SM'03-F'18) received the B.Eng. (Hons.) degree from the National University of Singapore in 1989, and the M. Eng. and Ph.D. degrees from Nanyang Technological University (NTU) of Singapore in 1993 and 1998, respectively. He is an Associate Professor of the School of Electrical and Electronic Engineering (EEE) of NTU. He held joint appointments with the university as Assistant Chair of Alumni from 2008 to 2014, Deputy Director of the Center for High Performance Embedded Systems from 2000 to 2011, and Program Director of the Center for Integrated Circuits and Systems from 2003 to 2009. He has coedited 4 books, published 13 book chapters, more than 100 international journal papers (more than 70 are in IEEE) and more than 180 refereed international conference papers (mostly in IEEE), and delivered over 40 colloquia. His current research interests include hardware security, unconventional number systems, and low-power and fault-tolerant digital signal processing algorithms and architectures.

Dr. Chang currently serves as the Senior Area Editor of IEEE Transactions on Information Forensic and Security, and Associate Editor of the IEEE Transactions on Circuits and Systems-I and IEEE Transactions on Very Large Scale Integration (VLSI) Systems. He also served as the Associate Editor of the IEEE Transactions on Information Forensic and Security and IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems from 2016 to 2019, IEEE Access from 2013 to 2019, IEEE Transactions on Circuits and Systems-I from 2010 to 2013, Integration, the VLSI Journal from 2013 to 2015, Springer Journal of Hardware and System Security from 2016 to 2020 and Microelectronics Journal from 2014 to 2020. He guest edited several special issues and served in the organizing and technical program committee of more than 60 international conferences (mostly IEEE). He is an IEEE Fellow, IET Fellow, and 2018-2019 Distinguished Lecturer of IEEE Circuits and Systems Society.