TU Delft

Delft University of Technology

Deterministic and Statistical Strategies to Protect ANNs against Fault Injection Attacks

Köylü, T.C.; Reinbrecht, Cezar; Hamdioui, S.; Taouil, M.

DOI
10.1109/PST52912.2021.9647763

Publication date
2021

Document Version
Final published version

Published in
2021 18th International Conference on Privacy, Security and Trust (PST)

Citation (APA)
Köylü, T. C., Reinbrecht, C., Hamdioui, S., & Taouil, M. (2021). Deterministic and Statistical Strategies to Protect ANNs against Fault Injection Attacks. In *2021 18th International Conference on Privacy, Security and Trust (PST)* (pp. 1-10). (2021 18th International Conference on Privacy, Security and Trust, PST 2021). IEEE. https://doi.org/10.1109/PST52912.2021.9647763

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Deterministic and Statistical Strategies to Protect ANNs against Fault Injection Attacks

Troya Çağıl Köylü, Cezar Rodolfo Wedig Reinbrecht, Said Hamdioui, Mottaqiallah Taouil
Computer Engineering, Delft University of Technology
Delft, the Netherlands

2021 18th International Conference on Privacy, Security and Trust (PST) | 978-1-6654-0184-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/PST52912.2021.9647763

***Abstract*—Artificial neural networks are currently used for many tasks, including safety critical ones such as automated driving. Hence, it is very important to protect them against faults and fault attacks. In this work, we propose two fault injection attack detection mechanisms: one based on using output labels for a reference input, and the other on the activations of neurons. First, we calibrate our detectors during normal conditions. Thereafter, we verify them to maximize fault detection performance. To prove the effectiveness of our solution, we consider highly employed neural networks (AlexNet, GoogleNet, and VGG) with their associated dataset ImageNet. Our results show that for both detectors we are able to obtain a high rate of coverage against faults, typically above 96%. Moreover, the hardware and software implementations of our detector indicate an extremely low area and time overhead.**

***Index Terms*—fault injection, countermeasures, artificial neural networks, machine learning**

I. Introduction

Artificial neural networks (ANNs) first started to be used for the modelling of biological neurons [1] and are in the meantime used for many tasks (e.g., image processing [2], speech recognition [3], and big data applications [4]). Nowadays, they are also being used in many automated and critical tasks such as real-time object detection and decision making, as it is the case for autonomous driving [5]. The reliable operation of these ANNs are of paramount importance. A security violation such as tampering with these networks can lead to drastic consequences like accidents [6], [7]. Today, this is especially of concern due to the prominence of *fault attacks*. Although classical fault injection techniques like underfeeding, heating, or shooting EM waves and lasers [8] might not represent a huge threat due to their physical proximity requirements, logical approaches are still possible. Moreover, it also important to protect against faults caused by radiation [9]. Three main strategies can be used to compromise the system integrity: (i) direct adulteration: an attacker hacks into the system and gains enough privileges to deliberately modify the memory content [10]; (ii) indirect adulteration: an attacker gets access to the system with no privilege, and performs a buffer overflow attack to overwrite some memory contents [11]; (iii) Rowhammer attacks: an attacker gets access into the system with no privilege, and performs a Rowhammer attack to force bit-flips in the main memory [12]. Therefore, a critical system running an ANN can get corrupted by these logical fault injection attacks.

Several researchers focused on protecting ANNs against fault attacks. We can divide their protection schemes into two groups, i.e., *intrinsic* and *extrinsic*. *Intrinsic* protection schemes use the inherent properties of neural networks for protection. There are a couple of ways to attain this. First, feedback mechanisms in recurrent neural networks and Hopfield networks help with tolerating injecting faults [13]. However, these types of networks only constitute a small subset of ANNs. The second intrinsic method uses fault aware training, either by injecting faults in the neurons during training [14], or using modified training algorithms [15]–[18]. These techniques present a more fundamental solution. However, many applications are built on top of already trained networks and hence, it is not always possible to gather the dataset and do retraining. Moreover, using custom learning algorithms is not convenient when using machine learning frameworks. In [19], the authors use fault tolerant training to maximize the fault tolerance of RRAM-based neural network implementations. This makes the protection hardware-aware, however further limits its general application. Another intrinsic protection method is to evaluate the fitness of the trained network (e.g., weights) for fault tolerance [20], [21]. Although such an approach is very beneficial to determine and construct a fault tolerant network beforehand, it again requires to do retraining. In addition, using deterministic formulations for assessment does not scale well for very deep networks with a lot of parameters. In a more recent example, [22] proposed a fault injection framework to evaluate the fault tolerance of deep neural networks (DNNs) after training. Their results show that DNNs offer limited intrinsic protection, especially against certain faults. The last intrinsic protection type, which can also be considered extrinsic for some applications, is redundancy. We refer intrinsic redundancy to the case where redundancy is included in the architecture. In [23], the authors achieve this by replicating the hidden layer-to-output unit structure in the architecture before training. On the other hand, [24] and [25] include redundant neurons after training, where they adjust the other weights to preserve input-output

Authorized licensed use limited to: TU Delft Library. Downloaded on January 04,2022 at 07:43:33 UTC from IEEE Xplore. Restrictions apply.

mapping of the layer. All these methods require modifications to the original network. Extrinsic protection schemes on the other hand consider neural networks as a part of a system, and protect the whole system from attacks. For physical attacks, several protection schemes have been introduced such as shielding and sensors [26]. For logical attacks, only redundancy-based techniques have been proposed, such as dual or triple modular redundancy [27]. Such techniques are very effective in detecting fault attacks but require extra resources like dedicated hardware [27] or affect the performance [28]. The area overhead and/or performance loss are costly, especially for large ANN architectures. Clearly, an effective yet efficient countermeasure that addresses these limitations is needed against fault attacks, which also takes ANN characteristics into consideration.

In this paper, we present two fault injection attack detection mechanisms: a deterministic and a statistical one. The deterministic detector evaluates a reference input to detect persistent fault attacks. The statistical detector on the other hand checks the neuron activation rates in the layers of an ANN to detect unexpected behavior manifesting from faults. Given a trained ANN, both mechanisms do not require any modification to the network or a costly retraining: once calibrated with a subset of data, they can generally be used for fault detection. In summary, the contributions of this paper are the following:

- Proposal of an effective and efficient deterministic fault attack detector based on periodic inference verification.
- Proposal of an effective and efficient statistical fault attack detector based on neuron activation rates.
- Evaluation of the proposed detectors with three state-of-the-art ANN architectures, namely AlexNET, VGG, and GoogleNet.
- Implementation and overhead analysis of our detectors both in hardware and software.

The remainder of the paper is organized as follows. Section II provides an introduction to existing attacks and explains our threat model. Section III introduces the two detectors. Section IV describes the experimental setup and presents the results. Finally, Section V concludes this paper.

II. Attack and Threat Models

In addition to the increasing threat of fault attacks, there are many studies that utilize faults to tamper with ANNs. In this section, we describe some relevant attacks and the considered threat model.

A. ANN Attacks

Several fault attacks on ANNs have been performed. For example, the authors in [29] present two attacks: single bias and gradient descent attacks. The single bias attack aims at changing the decisions that need to be taken in the last layer. To realize that, the attacker changes the biases of the neurons responsible to make the final decisions. The gradient descent attack on the other hand aims at changing weights and biases in such a way that a misclassification occurs, while targeting minimum changes in the network. To realize this, it first uses the gradient descent algorithm to calculate updates on some of the parameters (weights and biases) of the layer with the most neurons to obtain a desired misclassification scheme. Thereafter, it only utilizes a minimum number of these adversarial updates while still ensuring the desired misclassification.

In [30], the authors focused on injecting faults to different activation functions that are typically used in ANN architectures: rectified linear function (ReLU), sigmoid, hyperbolic tangent (tanh), and softmax (as part of the decision layer). Their results show that when the outputs of ReLU, sigmoid, and tanh are affected, misclassifications can occur.

Another study was presented in [31]. In this study, the authors investigated the effects of faults on DNN accelerators. They injected transient faults and single event upsets [32] in the datapath and buffers. Consequently, they investigated when these faults cause a misclassification. Their study highlights the importance of protecting the integrity of these networks as a false classification can result in devastating consequences.

B. ANN Threat Model

This work focuses on ANNs that are applied for safety critical applications, such as in self-driving cars. In the previous sections, we have established that logical fault injection can be used to tamper with the parameters of the ANN and that they can disrupt the ANN operation unnoticed. Generally, fault injection attacks allow attackers to gain privileges to indirectly compromise the data in the memory, or directly corrupt the memory. With respect to the security, we consider the following three scenarios:

- No security: When a system has no added security measures, the attacker can raise their privilege to change memory content [10]. In this scenario, the attacker has full control and may change values at any location.
- Low to medium security: In this scenario, the attacker is not directly able to change data. However, a buffer overflow attack [11] can be used to overwrite parts of the memory and in turn affect the ANN. In this scenario, attackers have partial control: they can inject faults but have no control over the exact location.
- High security: In a very secure scenario, a more sophisticated attack that exploits hardware vulnerabilities is required. Hence, Rowhammer attacks can be a valid option [12]. As a result, the attacker can cause bit-flips in random locations around the target

Authorized licensed use limited to: TU Delft Library. Downloaded on January 04,2022 at 07:43:33 UTC from IEEE Xplore. Restrictions apply.

memory space. The attacker in this scenario has very limited control on the fault value and location.

In all considered attack scenarios, the memory is the main target of the attack. When focusing on ANNs, the memory is responsible for storing the network parameters, such as weights and biases. Tampering with the parameters is also possible when ANNs are implemented in hardware. Hence, we assume that an attacker can modify weights and/or biases during a classification operation. Note that our considered scenario does not comprehend attacks to the internal operations (e.g., activation functions) or intermediate outputs. Nevertheless, faults injected in such places will most likely have equivalent effects to faults in weights and biases for many cases, if not all. Additionally, we do not consider attacks that tamper with the input (i.e., using adversarial inputs). We assume that due to hard real-time constraints, the input of such systems (i.e., image from a camera) will use a dedicated buffer, bypassing the typical memory hierarchy [10]. Finally, we assume that the aim of the attacker is to disrupt reliable operation, and hence, we assume no limitation on the value or location of weights and biases that are targeted with faults.

III. Proposed Detectors

In this section, we describe our two detection strategies to protect ANNs for the aforementioned threat model. Both detectors are described in three steps. First, we present the concept behind their detection method. Next, we present the general functionality. Lastly, we provide detector implementation details for both software and hardware ANNs.

A. Deterministic Strategy - The Δ-detector

1) Concept: After an ANN is trained, the internal variables of the model are fixed (i.e., weights and biases) in the deployment. This means for a specific input, the intermediate and final outputs will always be the same. Consequently, this deterministic behavior can be used to detect faults in the system. Consider a specific input, whose ANN inference intermediates or output is stored as a reference. Then, in the field, we can regularly supply the same reference as input to the ANN and obtain an inference value. If this value is different than the reference, one or more faults are present in the network. Such a detector would detect a very large portion of all faults that persist during the check. Namely, if the faults have an effect on the last layer (this can still be the case when faults are injected in preceding layers), a detector that monitors this layer can detect the faults, irrespective whether they affect the ANN decision or not. Thereafter, the system can reload the ANN or reboot the application. For the case of a self-driving car, this can result in signaling the driver that the automated driving will be disengaged due to potential failures in the ANN.

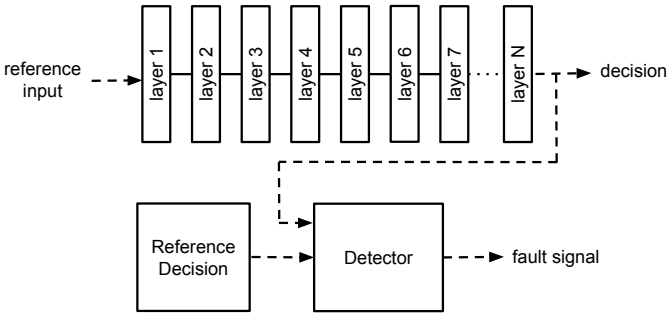

Fig. 1. Conceptual Architecture of the Δ-Detector

The selection of the variables to be used as reference may vary depending on the application. For example, image classifiers present many different output probabilities as they assign each input to an image category. Consequently, in this work we only focus on the output values after an image classification/inference. This strategy depends on the property that a modification in a weight or bias can alter one or more output probability values.

2) Functionality: We name our deterministic detector as the Δ (Delta)-detector: this detector checks for the difference between the reference and actual values of the ANN inference output. The Δ-detector initially selects a sample image as the reference input. Next, it runs the inference of this input in the deployed ANN and saves all the probabilities of each output label. Because of the dataset we use in this work (see Section IV-A), all ANNs used in this paper present 1000 output labels. This means the detector stores 1000 floating point numbers as a reference, which are typically class probabilities (when softmax is used on the output). The resulting conceptual architecture is shown in Figure 1.

3) Implementation: Next, we present both software and hardware implementations of the Δ-detector.

Software Implementation: In case the ANN is part of a software application, the detector must also be employed in software. This means that besides the ANN, an additional function needs to be called for the fault detection. The function of the Δ-detector comprises three steps, as shown in Algorithm 1. First, it loads the reference input and output labels. Thereafter, it runs the inference process in the ANN. Last, it collects all output class labels and compares it to their reference values. Any mismatch raises a fault signal.

Hardware Implementation: In case the ANN is employed as a hardware accelerator, it is more efficient to implement the detector also in hardware. The hardware implementation of the Δ-detector follows the same steps described in Algorithm 1. For the loading process (both reference input and output), the hardware can use the system's main memory. The usage of dedicated memories in tamper-proof locations to store the references is

Authorized licensed use limited to: TU Delft Library. Downloaded on January 04,2022 at 07:43:33 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Pseudo-code of the Δ-detector

Input: reference $input_{ref}$, ann, reference $\overline{output_{ref}}$
Output: Fault signal $fault$

1: $\overline{output} \leftarrow ann(input_{ref})$ ▷ inference operation
2: $fault \leftarrow 0$
3: **for each** $output_i$ in $\overline{output}$ **do** ▷ $output_i$: output label of class i
4: **if** $output_i \neq output_{ref_i}$ **then**
5: $fault \leftarrow 1$
6: **end if**
7: **end for**

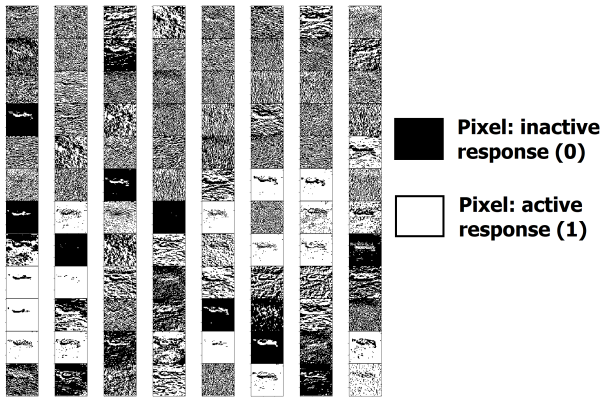

Fig. 2. Example Activation Map of a Convolutional Layer of AlexNet

also justified for maximum security. Lastly, the comparison operation (line 4) can be easily implemented through a bit-wise XOR. If the result is zero among all output class labels, then there is no fault presence. Otherwise, it raises the fault signal (line 5).

B. Statistical Strategy - The Σ-detector

1) Concept: The training process of an ANN updates its internal parameters that are composed of weights and biases. As a result, a trained ANN will exhibit specific internal patterns during inference. One way to analyze such patterns is by evaluating the number of neurons that are activated in a layer. Our hypothesis is that the ratio of activated neurons generally lies in specific bounds during normal conditions (i.e., when no fault attacks are present), as learning algorithms are expected to regularize neuron behavior into a pre-determined input-output mapping. This hypothesis follows from the "firing neuron rate (FNR)" idea presented in [33], which indeed shows a different activation pattern for regular and adversary inputs.

When an input is applied to the ANN, the Σ-detector obtains the binary activation map for each layer, which are of different shapes for each layer. Figure 2 illustrates an example of a $55 \times 55 \times 96$ activation map (see convolution (1) layer in Table I(a)) obtained from the outputs of a convolutional layer. Then, in order to summarize these maps, Σ-detector calculates the ratio of activations to the total number of neurons within a layer. For example, if 100k neurons are activated (i.e., produced a number greater than 0) in the aforementioned convolutional layer, its activation rate is $100\text{k}/(55 \cdot 55 \cdot 96) = 0.344$. We store one such rate per layer.

We determine those rates of whether or not a neuron is activated by the following equation:

$$activation_{n_{i,j}} = \begin{cases} 0, & \text{if } out_{n_{i,j}} <= 0. \\ 1, & \text{if } out_{n_{i,j}} > 0, \end{cases} \quad (1)$$

Here, $n_{i,j}$ is the jth neuron of the ith layer and $out_{n_{i,j}}$ is its output.

2) Functionality: The Σ-detector processes the activations as the ANN executes. Figure 3 illustrates its conceptual architecture. When an input is supplied to the ANN, the detector collects the activation rates of each of the layers. The detector consequently investigates if these ratios are in expected boundaries, and otherwise generates a fault signal (which is 1 if a fault is detected and 0 otherwise). The ANN generates an inference decision in parallel. The final output of the system consists of the decision of ANN and value of fault signal. When a fault is detected, we raise an alarm instead. Similar to the Δ-detector, the nature of the alarm and the response to it can vary from application to application.

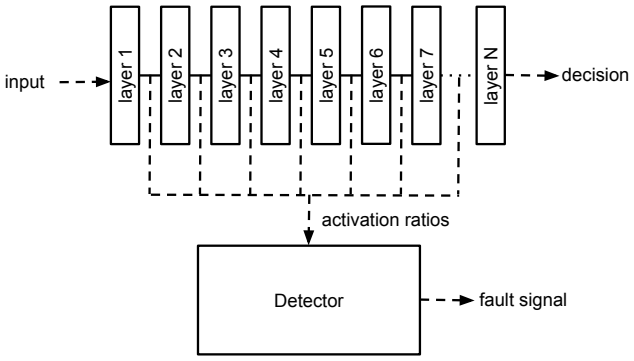

Fig. 3. Conceptual Architecture of the Σ-Detector

We name our detector as the Σ (Sigma)-detector as it compares activation rates with the *mean* (μ) and *standard deviation* (σ), where both μ and σ were pre-calculated from non-faulty data (only a subset of the original dataset is enough). If the value is outside the expected range (e.g., 3σ), it raises a warning. When one or more warnings are raised, the output fault signal is set.

3) Implementation: Next, we describe both software and hardware implementations of the Σ-detector.

Software Implementation: Algorithm 2 details the pseudo-code of the software implementation of the basic detector. After obtaining the activations for an input image (line 2), the detector checks layer-by-layer if the activation value is in the determined boundary (line 4).

Authorized licensed use limited to: TU Delft Library. Downloaded on January 04,2022 at 07:43:33 UTC from IEEE Xplore. Restrictions apply.

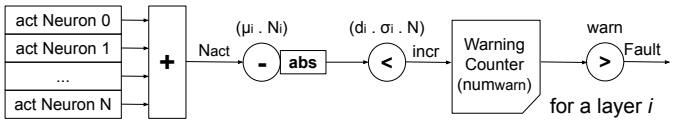

Fig. 4. Hardware Architecture of Σ-Detector

If not, it raises a warning (line 5). When the warning threshold is reached, a fault is signaled (line 8-9).

Algorithm 2 Pseudo-code of the Σ-detector

Input: $input$, ann, calculated $\overline{\mu}$, calculated standard deviation $\overline{\sigma}$, calculated maximum allowed distance in terms of standard deviation $\overline{d}$, number of warnings to set the fault signal $warn$

Output: Fault signal $fault$

1: $num_{warn} \leftarrow 0$ ▷ initialization of warning
2: $\overline{act} \leftarrow ann(input)$ ▷ $\overline{act}$: activation ratios
3: **for each** act_i in $\overline{act}$ **do** ▷ act_i: activation ratio of layer i
4: **if** $\text{abs}(act_i - \mu_i) \leq d_i \cdot \sigma_i$ **then**
5: $num_{warn} \leftarrow num_{warn} + 1$
6: **end if**
7: **end for**
8: **if** $num_{warn} >= warn$ **then** ▷ fault condition
9: $fault \leftarrow 1$
10: **else** ▷ no fault condition
11: $fault \leftarrow 0$
12: **end if**

Hardware Implementation Figure 4 illustrates the hardware architecture of the Σ-detector. Our proposed scheme can evaluate a layer in a single cycle, which means it can be reused for all layers when it is designed for the layer with the most number of neurons. The only change along the layers are the mean and standard deviation values, which are implemented as constants (i.e., their bits are tied to the V_{dd} when 1, or ground when 0).

As observed in the figure, the hardware collects and adds the activation results of the currently executed layer. Instead of dividing by the number of neurons per layer, we multiply the equation on line 4 in Algorithm 2 by the number of neurons. Hence, the total number of active neurons N_{act} is evaluated using the equation $abs(N_{act} - \mu_i \cdot N) \leq d_i \cdot \sigma_i \cdot N$, where N equals the total amount of neurons in a certain layer and $N_{act} = N \cdot act_i$. If the equation is not satisfied, the warning counter num_{warn} increments. Finally, comparing the result with the threshold $warn$ sets the fault signal.

C. Combining both strategies

In the previous subsections, we have proposed two different fault attack detection strategies. The first one (i.e., Δ-detector) is very effective, given its fault assumptions hold (i.e., a fault will persist during the check). Furthermore, it does not have any false alarms. The detector can be considered as a ANN-aware redundancy that is costly in terms of performance (when implemented as a software implementation) and resources (when implemented as a hardware implementation). The reason is that for each inference, the Δ-detector should conduct an additional inference and check the results. In some cases such a cost would be unacceptable, such as in automated driving with a constant stream of images.

The second strategy (i.e., Σ-detector) does not require any costly operation during deployment time. Therefore, it is suitable to be used continuously. However, as any statistical method, it is inevitably prone to missing some fault attacks or may even generate false alarms. As such, either strategy can be chosen depending on the high security versus efficiency needs. Moreover, a combination of two strategies is also possible. Namely, if the aim is to eliminate all false fault alarms while allowing some faults, the check of the Δ-detector can be initiated as soon as the continuously running Σ-detector detects a fault. The presence of a fault attack can be guaranteed when both detectors raise the fault signal. Using this approach, the overhead caused by the Δ-detector remains low as it only need to be executed when the Σ-detector raises a fault alarm (true or false positive). Another strategy is to mainly rely on the Σ-detector for detecting transient faults (e.g., that affect the registers), while using the Δ-detector for periodic self-checks to detect more persistent faults, such as the ones that affect the main memory.

IV. Experimental Results

In this section, we present the experiments that we used to prove the effectiveness and efficiency of our detectors. First, Subsection IV-A describes the details of our setup. Next, Subsections IV-B and IV-C describe the conducted experiments and present their results.

A. Experimental Setup

This subsection describes the target ANNs and datasets, the target platforms, our fault injection framework, and the performed experiments.

Target ANNs and datasets: In this work, to prove the generality of our detectors, we use three widely employed ANN architectures: AlexNet [2], VGG (CNN S) [34], and GoogleNet [35]. All ANNs are convolutional neural network (CNN) architectures that were trained on the ImageNet Large Scale Visual Recognition Challenge 2012 [36]. We use the validation repository of this dataset, which consists of 50k images in total. Table I shows the size of output of each relevant layer of these networks. As can be observed, VGG uses slightly different parameters than AlexNet, while GoogleNet is a very deep architecture. We conducted all the experiments using the Python language together with the Caffe toolbox [37]. This toolbox provides slight variations of

Authorized licensed use limited to: TU Delft Library. Downloaded on January 04,2022 at 07:43:33 UTC from IEEE Xplore. Restrictions apply.

all three networks. They were all pre-trained to achieve 80%, 87%, and 90% accuracy on the validation repository.

To identify suitable parameters and evaluate both detectors, we created three datasets (per AlexNet, VGG, and GoogleNet). The first dataset is called *calibration set*. This set contains the first 1000 images from the aforementioned ImageNet validation repository, and it is used to understand the faulty and non-faulty behavior, and hence, to define the parameters of the detectors. In Δ-detector, the parameters are the output label values. In Σ-detector the parameters are the mean and standard deviation of activations for each layer. Next, the second dataset, which we label as the *verification set*, is used to validate the chosen parameters. It contains the images 1001 to 2000 from the repository. The last dataset, i.e., the *evaluation set*, contains images 2001 to 3000 and it is used to evaluate the effectiveness of the detectors under practical attack scenarios. Note that the *evaluation set* is completely independent from the first two sets, and they constitute unseen images for our detectors. Hence, the detection results on this set provide the generalized effectiveness of our detectors.

Target Platforms: We evaluate the detector overhead for the selected ANNs both in software (i.e., desktop or server) and in hardware (i.e., GPU or FPGA-based platforms). Our software implementation runs on a server with a 2.1GHz processor and 96GB memory. We make the hardware overhead comparisons with a synthesized ANN accelerator [38] for the Virtex-7 VC707 FPGA board [39]. Note that both hardware and software ANN implementations are relevant in the context of self-driving cars and hence, their operation are vital for the safety [40], [41].

Fault Injection Framework: In line with the considered threat model (see Subsection II-B), our framework is able to modify the weights and/or biases of the neural network during the inference process. The framework injects faults according to two aspects:

- *Fault locations* - A fault can affect any layer of the ANN with adjustable (learned) parameters. In our case, these are the layers indicated in Table I.
- *Fault types* - We consider bit and byte-level faults to the weights and biases. Referring to the scenarios in our threat model (see Section II-B), byte-level or a larger amount of faults are applicable for the low/medium security scenario, where the attacker can define new values for the weights. In contrast, a bit-level fault could take place in the high security scenario, where a Rowhammer attack [12] can take place. The location of the bit faults are randomly selected and hence could have a low or large impact.

In addition to the fault type, we experiment also with the number of faults. Note that radiation induced error can also cause single or multiple bit flipping [32] and hence, our detector can be used against them as well.

B. Δ-Detector Experiments

In this section, we provide the calibration, performance analysis and implementation overhead details for Δ-detector.

1) Detector Calibration: During the detector calibration, we used some images from the *calibration set* to validate its functionality. As such, we started our test with conducting inference with an image without any faults and saved the output labels. Thereafter, we conducted inference with faulty and non-faulty instances of the same image. The output label comparison of the Δ-detector yielded correct results (i.e., no false alarm in the correct case, and fault detection in the faulty case). Therefore, we have proven the validity of the Δ-detector, and can continue with larger scale verification.

2) Performance analysis: In this analysis, we used the *evaluation set* images. For each of the 1000 images, we tested the Δ-detector with comparing the output labels of correct versions to versions where we injected 0, 1, 5, and 10 faults randomly. Table II shows the results. In this table (and also in later ones), we follow a coverage based analysis of our results, inspired by [42]. First, we present the false alarm rate for 0 faults. Next, we present the detection results for the faulty cases using three categories: detection, top5 coverage, and top1 coverage. Detection represents the ratio of fault cases that are detected by the detector. Top5 coverage represents the coverage against faults that make the top5 decision faulty, and the ratio is calculated by $(\text{detected} + \text{undetected that does not affect top5 accuracy})/1000$. Similarly, top1 coverage represents the coverage against faulty top1 decisions. Here, the top$k$ inference of an ANN is correct if one of the maximum k inference labels is equal to the actual image class. Lastly, the *misc.* (misclassification) next to each coverage indicates the percentage that faults affect the top$k$ inference. Hence, the difference between this misc. value and (100 - top1 or top5 coverage) indicates the protection level of our detector.

It can be observed from Table II that the Δ-detector indeed does not raise any fault alarms and also has full coverage except for one case (GoogleNet 1 fault) for decision affecting faults. The investigation of that case showed that it is a very rare computational overflow error, that even prevailed when no faults were present to make the classification wrong to begin with. Overall, the Δ-detector accounts for detecting up to 31% for some cases (i.e., 10 faults in GoogleNet top5 coverage) for faults that lead to misclassifications.

On the other hand, the overall detection rate is lower, especially when 1 fault is injected in AlexNet (66%) and VGG (64%). As the detection rate is much higher for higher faults, and also for 1 fault case in GoogleNet, we can re-verify that the undetected faults are indeed the ineffective ones. To elaborate, these faults are likely

Authorized licensed use limited to: TU Delft Library. Downloaded on January 04,2022 at 07:43:33 UTC from IEEE Xplore. Restrictions apply.

TABLE I
STRUCTURES OF (A) ALEXNET, (B) VGG, AND (C) GOOGLENET

layer	size
convolution (1)	$55 \times 55 \times 96$
convolution (2)	$27 \times 27 \times 256$
convolution (3)	$13 \times 13 \times 384$
convolution (4)	$13 \times 13 \times 384$
convolution (5)	$13 \times 13 \times 256$
dense (1)	$1 \times 1 \times 4096$
dense (2)	$1 \times 1 \times 4096$
dense (3)	$1 \times 1 \times 1000$

(a)

layer	size
convolution (1)	$109 \times 109 \times 96$
convolution (2)	$33 \times 33 \times 256$
convolution (3)	$17 \times 17 \times 512$
convolution (4)	$17 \times 17 \times 512$
convolution (5)	$17 \times 17 \times 512$
dense (1)	$1 \times 1 \times 4096$
dense (2)	$1 \times 1 \times 4096$
dense (3)	$1 \times 1 \times 1000$

(b)

layer	size
convolution (1)	$112 \times 112 \times 64$
convolution (2)	$56 \times 56 \times 64$
convolution (3)	$56 \times 56 \times 192$
...	...
inception (13)	$14 \times 14 \times 192$
...	...
inception (54)	$7 \times 7 \times 128$
dense (1)	$1 \times 1 \times 1000$

(c)

TABLE II
RESULTS OF THE EVALUATION OF Δ-DETECTOR.

AlexNet			
0 faults	0%		
	Detection	Top5 Coverage / misc.	Top1 Coverage / misc.
1 fault	66%	100% / 3.8%	100% / 2.8%
5 faults	99.4%	100% / 12.7%	100% / 9.5%
10 faults	100%	100% / 19.6%	100% / 13.8%
VGG			
0 faults	0%		
	Detection	Top5 Coverage / misc.	Top1 Coverage / misc.
1 fault	64.3%	100% / 2.8%	100% / 2.1%
5 faults	98.6%	100% / 11.7%	100% / 9.5%
10 faults	99.9%	100% / 18.9%	100% / 13.7%
GoogleNet			
0 faults	0%		
	Detection	Top5 Coverage / misc.	Top1 Coverage / misc.
1 fault	83.5%	99.9% / 3.6%	99.9% / 3.4%
5 faults	100%	100% / 17.2%	100% / 13.9%
10 faults	100%	100% / 30.7%	100% / 24.7%

the ones that disappear in the ANN, e.g., filtered out by the negative inputs of the ReLU activation [43]. As GoogleNet is a much larger network, a single fault has more chance to create exponential changes that affect the inference result. Thus, our detector detected more of them. In conclusion, the Δ-detector is very effective in covering against dangerous faults, given the assumption that the fault persists during the check.

3) Implementation Overhead: As mentioned previously, the Δ-detector can be employed both in software and hardware. In software, the detector includes an additional inference operation plus a check, for each of the inference operations. To see the added latency, we have recorded the total elapsed time over 1000 image inferences, separately for the original and the Δ-detector operations. We have used the process time function of Python, which discards sleep time. The results, as expected, show that the detector creates 99-100% latency to conduct the additional inference and check for all of the ANNs.

In hardware, the Δ-detector consists of a dedicated memory to store reference input and output labels, and a checker. Our synthesis resulted in 4kB of embedded memory and less than 1% of logic for detector control and comparison. Note that the reference values can also be stored in the system memory (with reduced security) to avoid a memory overhead.

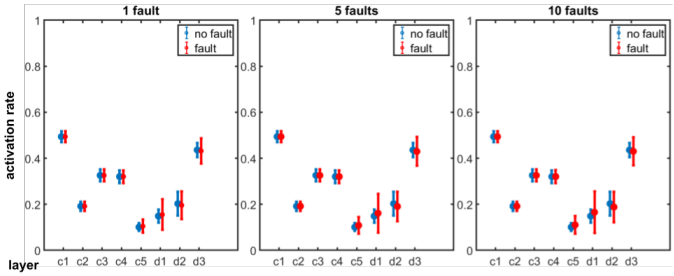

Fig. 5. Neuron Activation Rates for Fault Injection into Convolution 4 (c4) Layer of AlexNet

C. Σ-Detector Experiments

In this section, we first present the analysis regarding the activation rate behavior of ANN layers during non-faulty and faulty inference operations. Thereafter, we provide the calibration, performance analysis and implementation overhead details for the Σ-detector.

1) Evaluation of the Neuron Activation Rate: In this experiment, we perform a detailed fault analysis on the *calibration set*. For each image, we conduct a fault injection campaign into AlexNet by injecting $\{0, 1, 5, 10\}$ faults on each of the considered layers (convolution (1), (2), (3), (4), (5); dense (1), (2), (3)) randomly. Each time we consider faults in a single layer only. For each fault campaign we evaluate 1000 images. As an example, the results of injecting faults to convolution layer 4 are illustrated in Figure 5. In the figure, the center points indicate the mean of the activation rate for that particular layer, while the arrow lengths the standard deviation.

There are a couple of observations from this graph. First, we observe that the faults injected in a layer (convolution 4 in the figure) indeed disrupt the expected activation behaviour in the proceeding layers (convolution 5 and dense 1, 2, 3). The propagation of the fault through many connections into later layers results in a much worse behavior at these later layers. Consequently, this also implies that faults injected into the last layers are harder to detect. Second, the level of disruption increases with the number of injected faults. Due to space limitations, we have omitted the results for the other layers. Nevertheless, their results are similar.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 04,2022 at 07:43:33 UTC from IEEE Xplore. Restrictions apply.

TABLE III
TWO BEST PARAMETER SELECTIONS FOR TARGETS ANNS.

		accuracy	d	$warn$	layers
AlexNet	s1	0.563	3	1	last half
	s2	0.561	3	1	all
VGG	s1	0.549	3	1	all
	s2	0.549	d_{max}	1	all
GoogleNet	s1	0.588	d_{max}	1	last half
	s2	0.587	d_{max}	1	all

2) Detector Calibration: After the initial investigation, we calibrate our Σ-detector data from the *calibration set* without injecting faults. We calculate μ, σ, and d_{max} (the highest distance of a non-faulty activation rate from the mean in terms of standard deviations) for each layer of the ANNs.

To calculate the parameters of d and $warn$, as well as which layers to consider in the detector, we conduct another set of experiments by injecting $\{0, 1, 5, 10\}$ faults into the weights and biases when images of the *verification set* are considered. However, this time faults are randomly injected into any layer during a run. Hence, this creates 4 sets of activation rates, each consisting of 1000 images. To find the optimal settings for the detector, we try the following values: $d \in \{1, 2, 3, d_{max}\}$, $warn \in \{1, 2, 3, 4\}$ (for AlexNet and VGG) and $warn \in \{1, 5, 10, 25\}$ (for GoogleNet), considered layers $\in$ {all, last half, only convolutional, only dense} (for all ANNs, which results in different number of layers for GoogleNet). A set of parameters is selected based on the accurate labeling of 4000 images, where we adjusted the classification impact of non-faulty instances for a fair comparison. Table III shows the parameters for the two best selections s1 and s2 for the three ANNs. As can be observed from GoogleNet, finding the optimal values scales well for larger neural networks as the detector parameters are layer independent.

3) Performance Analysis: To evaluate our detector, we follow the same presentation scheme in Section IV-B. We first evaluate our correct labelling for non-faulty and faulty instances. For faulty instances, we further report our coverage for faults that affect a top5 or top1 classification of the ANN.

To conduct this evaluation, we use the *evaluation set* to create four sets of activations in an identical manner used in Σ-detector calibration (see Section IV-C2). Table IV shows the detection results for the best two parameter set configurations s1 and s2.

The results show that all ANNs and both s1 and s2 have similar results. Typically, the best parameter (s1) attains a false positive rate smaller than 4%. We can detect very few of the 1 fault cases (>3%), significantly more of the 5 fault cases (>14%), and yet more of the 10 fault cases (>19%). However, the coverage for top5 and top1 affecting faults for all cases are quite high (>96%), where top1 coverage is a bit higher than top5. This means that faults injected into the neural network are detected with a probability of 96% (or higher) when it affects the ANN inference result. This can be explained as follows: we observed that only ∼2% of 1 fault cases cause a misclassification in top5 and top1, which explains the low detection rate. This ratio increases to ∼12% for 5 fault cases and ∼20% for 10 fault cases. Overall, the Σ-detector can nullify most of the effects of *dangerous* faults (i.e., faults that cause misclassifications), covering up to 28% for some cases (i.e., 10 faults in GoogleNet top5 coverage). Another important point is that we observed that our detector performs very similarly to other more complex classifiers that we experimented with (i.e., naïve Bayes, support vector machine, and a 2-layer multilayer perceptron) on AlexNet.

TABLE IV
RESULTS OF THE EVALUATION OF Σ-DETECTOR.

AlexNet			
	s1 / s2		
0 faults	1.6% / 3%		
	Detection	Top5 Coverage	Top1 Coverage
	s1 / s2	s1 / s2 / misc.	s1 / s2 / misc.
1 fault	3.6% / 5.2%	99.1% / 99.3% / 2.8%	99.2% / 99.3% / 1.7%
5 faults	14.2% / 15.8%	98.3% / 98.7% / 12.6%	98.4% / 98.9% / 9.9%
10 faults	19.8% / 22%	97% / 97.9% / 18.3%	96.9% / 97.6% / 14%
VGG			
	s1 / s2		
0 faults	3.8% / 0.7%		
	Detection	Top5 Coverage	Top1 Coverage
	s1 / s2	s1 / s2 / misc.	s1 / s2 / misc.
1 fault	6% / 2.9%	99.3% / 99.2% / 2.5%	99.2% / 99.1% / 2.2%
5 faults	14.4% / 11.2%	96.8% / 96.5% / 13.3%	96.9% / 96.8% / 10.8%
10 faults	23.3% / 20.2%	96.6% / 96% / 20.7%	96.9% / 96.4% / 16.4%
GoogleNet			
	s1 / s2		
0 faults	1.9% / 4.4%		
	Detection	Top5 Coverage	Top1 Coverage
	s1 / s2	s1 / s2 / misc.	s1 / s2 / misc.
1 fault	5.4% / 7.8%	99.9% / 99.9% / 3.4%	99.7% / 99.7% / 3%
5 faults	21.2% / 23.2%	98.6% / 98.7% / 18.6%	98.7% / 98.8% / 15.4%
10 faults	33.8% / 35.6%	98.2% / 98.3% / 30.1%	98.5% / 98.5% / 23.8%

Compared to the protection scheme in [33], which tries to detect adversarial inputs on AlexNet with a detector, our detector performs quite well. They obtain a high detection rate (approximately 90%), but at the expense of a high false alarm rate (approximately 17%). Still, our 4% false alarm rate is a considerable value. As such, our combined solutions can be considered as a corollary, which would obtain 0% false alarm rate while retaining the detection rates of the Σ-detector, given that the fault persists during both detectors' checks (see Section III-C). Note that we did do not perform any experiments for the combined solution, as conclusions can be straightforwardly derived from their individual experiments.

Another protection scheme, which proposes to modify the activation functions attain 95.3% top1 coverage, when the faults reduce the top1 classification accuracy by 21.6% in AlexNet [44]. The closest AlexNet scenario is our 10 faults case (with 14% misclassification), which we provide 96.9% top1 coverage. However, that study experiments with a different dataset and only injects bit faults to test fault tolerance, and hence a direct comparison is not possible.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 04,2022 at 07:43:33 UTC from IEEE Xplore. Restrictions apply.

4) Implementation Overhead: As we consider both software and hardware implementations for our Σ-detector, both must comply with some constraints. In software, it is important for the Σ-detector to produce a result quickly after the ANN produces a decision: any extra time will lead to an inaction latency. To determine the timing that our detector requires, we collected both ANN inference time and detector processing time over 1000 images. Accordingly, ANN requires <2% extra time to extract activation ratios, and a very insignificant $<2e^{-4}$% extra time for fault detection.

To evaluate the hardware overhead, we designed the detector for AlexNet's convolution (1) layer, which has the most amount of activations in AlexNet. We omit designs for VGG and GoogleNet, as our detector is reused for all layers, so the number of layers are not a source for overhead (see Section III-B3). Our synthesis resulted in area requirement of 60528 LUTs and 4 registers. Additionally, our detector met all timing constraints, meaning that it can produce a single warning signal per cycle. We also evaluated a second implementation, where the adder (see Figure 4) is replaced by a ROM-based decoder. In this scenario, the area utilization reduced significantly to only 870 LUTs, 4 registers and 290 kbits of ROM. When we compare these implementations to the reference CNN hardware accelerator [38], our detector results in 32.49% overhead for LUT-only, and 0.46% for ROM-based version.

V. Conclusion

In this work, we presented two effective detection algorithms for fault injection attacks on artificial neural networks, which do not require any modifications on the employed network. The results show that it is possible to cover the far majority of faults that lead to wrong decisions. To eliminate the false alarms, we also proposed a combined strategy that involves both detectors. Lastly, we presented ways to efficiently implement our detectors in software and hardware.

One point that we do not cover in this study is an attack against our detector itself. There are a couple of valid attack strategies against our detector: (i) inserting faults anywhere during the calculation or (ii) changing the reference or stored values (e.g., num_{warn}, d_i, or σ_i). The first attack (i) might be successful, if a fault simultaneously affects the ANN, while another affects the detector calculation in such a way that it misses to detect an unexpected value. In software, accomplishing this attack might be easier, with an instruction skip. However this attack is mostly impractical due to the need of synchronized faults, and it is far more likely that such an attack will start causing the detector to raise random fault signals. The second attack (ii) on the other hand can provide more success, especially for the Σ-detector: for instance, an attack that makes the value of num_{warn} larger. Although such an attack will still require a high level of granularity, a num_{warn} value larger than the number of layers will render our protection obsolete. Thus, we would recommend more safety in storing the detector parameters (e.g., using hardened memory locations).

Finally, we demonstrated the validity of our detectors in three ANNs using a single dataset. This can raise the question of applicability for different inputs. We leave this point for future work, although the ImageNet dataset that we experimented on is sufficiently broad, and the real time object classification scenario that we consider (in automated driving for instance) is one of the most relevant for such a protection.

VI. Acknowledgment

This work was labelled by the EUREKA cluster PENTA and funded by Dutch authorities under grant agreement PENTA-2018e-17004-SunRISE.

References

[1] W. S. McCulloch *et al.*, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[2] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, pp. 84–90, 2017.

[3] Y. Miao *et al.*, "Eesen: End-to-end speech recognition using deep rnn models and wfst-based decoding," in *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*. IEEE, 2015, pp. 167–174.

[4] J. Qiu *et al.*, "A survey of machine learning for big data processing," *EURASIP Journal on Advances in Signal Processing*, vol. 2016, pp. 1–16, 2016.

[5] M. Al-Qizwini *et al.*, "Deep learning algorithm for autonomous driving using googlenet," in *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2017, pp. 89–96.

[6] C. Miller *et al.*, "Remote exploitation of an unaltered passenger vehicle," Aug 2015. [Online]. Available: http://illmatics.com/Remote\%20Car\%20Hacking.pdf

[7] F. F. dos Santos *et al.*, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, vol. 68, pp. 663–677, 2018.

[8] T. C. Koylu *et al.*, "Rnn-based detection of fault attacks on rsa," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5.

[9] F. F. dos Santos *et al.*, "Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2017, pp. 169–176.

[10] S. Parkinson *et al.*, "Cyber threats facing autonomous and connected vehicles: Future challenges," *IEEE Transactions on Intelligent Transportation Systems*, 2017.

[11] A. Zhiyuan *et al.*, "Realization of buffer overflow," in *International Forum on Information Technology and Applications*, 2010.

[12] Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 361–372, 2014.

[13] G. Bolt, "Investigating fault tolerance in artificial neural networks," 1991.

[14] T. Ito *et al.*, "On fault injection approaches for fault tolerance of feedforward neural networks," in *Proceedings Sixth Asian Test Symposium (ATS'97)*. IEEE, 1997, pp. 88–93.

[15] F. Su *et al.*, "The superior fault tolerance of artificial neural network training with a fault/noise injection-based genetic algorithm," *Protein & cell*, vol. 7, pp. 735–748, 2016.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 04,2022 at 07:43:33 UTC from IEEE Xplore. Restrictions apply.

[16] S. Cavalieri *et al.*, "A novel learning algorithm which improves the partial fault tolerance of multilayer neural networks," *Neural Networks*, vol. 12, pp. 91–106, 1999.
[17] S. K. Mak *et al.*, "Regularizers for fault tolerant multilayer feedforward networks," *Neurocomputing*, vol. 74, pp. 2028–2040, 2011.
[18] Y. Tan *et al.*, "A fault-tolerant multilayer neural network model and its properties," *Systems and computers in Japan*, vol. 25, pp. 33–43, 1994.
[19] L. Xia *et al.*, "Fault-tolerant training with on-line fault detection for rram-based neural computing systems," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.
[20] C. Neti *et al.*, "Maximally fault tolerant neural networks," *IEEE Transactions on Neural Networks*, vol. 3, pp. 14–23, 1992.
[21] J. Sum *et al.*, "Prediction error of a fault tolerant neural network," *Neurocomputing*, vol. 72, pp. 653–658, 2008.
[22] B. Reagen *et al.*, "Ares: A framework for quantifying the resilience of deep neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.
[23] D. A. Medler *et al.*, "Training redundant artificial neural networks: Imposing biology on technology," *Psychological Research*, vol. 57, pp. 54–62, 1994.
[24] M. D. Emmerson *et al.*, "Determining and improving the fault tolerance of multilayer perceptrons in a pattern-recognition application," *IEEE transactions on neural networks*, vol. 4, pp. 788–793, 1993.
[25] D. S. Phatak *et al.*, "Complete and partial fault tolerance of feedforward neural nets," *IEEE Transactions on Neural Networks*, vol. 6, pp. 446–456, 1995.
[26] H. Bar-El *et al.*, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, pp. 370–382, 2006.
[27] R. E. Lyons *et al.*, "The use of triple-modular redundancy to improve computer reliability," *IBM journal of research and development*, vol. 6, pp. 200–209, 1962.
[28] L. Anghel *et al.*, "Cost reduction and evaluation of a temporary faults-detecting technique," in *Design, Automation, and Test in Europe*. Springer, 2008, pp. 423–438.
[29] Y. Liu *et al.*, "Fault injection attack on deep neural network," in
[42] A. Bosio *et al.*, "A reliability analysis of a deep neural network," in *2019 IEEE Latin American Test Symposium (LATS)*. IEEE, 2019, pp. 1–6.
2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2017, pp. 131–138.
[30] J. Breier *et al.*, "Practical fault attack on deep neural networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2204–2206.
[31] G. Li *et al.*, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.
[32] M. Bushnell *et al.*, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. Springer Science & Business Media, 2004, vol. 17.
[33] S. Wang *et al.*, "Fired neuron rate based decision tree for detection of adversarial examples in dnns," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020, pp. 1–5.
[34] K. Chatfield *et al.*, "Return of the devil in the details: Delving deep into convolutional nets," *arXiv preprint arXiv:1405.3531*, 2014.
[35] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.
[36] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, pp. 211–252, 2015.
[37] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.
[38] C. Zhang *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.
[39] *Xilinx Virtex-7 FPGA VC707 Evaluation Kit*, Xilinx, 2021.
[40] M. Bojarski *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.
[41] R. J. Gillela, "Design of hardware cnn accelerators for audio and image classification," 2020.
[43] B. Xu *et al.*, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015.
[44] L.-H. Hoang *et al.*, "Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1241–1246.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 04,2022 at 07:43:33 UTC from IEEE Xplore. Restrictions apply.