Fault Injection Attack on Deep Neural Network

Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu
Department of Computer Science & Engineering, The Chinese University of Hong Kong
Shenzhen Research Institute, The Chinese University of Hong Kong
Email: {ynliu, lxwei, boluo, qxu}@cse.cuhk.edu.hk

Abstract*—Deep neural network (DNN), being able to effectively learn from a training set and provide highly accurate classification results, has become the de-facto technique used in many mission-critical systems. The security of DNN itself is therefore of great concern. In this paper, we investigate the impact of fault injection attacks on DNN, wherein attackers try to misclassify a specified input pattern into an adversarial class by modifying the parameters used in DNN via fault injection. We propose two kinds of fault injection attacks to achieve this objective. Without considering stealthiness of the attack, single bias attack (SBA) only requires to modify one parameter in DNN for misclassification, based on the observation that the outputs of DNN may linearly depend on some parameters. Gradient descent attack (GDA) takes stealthiness into consideration. By controlling the amount of modification to DNN parameters, GDA is able to minimize the fault injection impact on input patterns other than the specified one. Experimental results demonstrate the effectiveness and efficiency of the proposed attacks.

***Index Terms*—fault injection, neural network, misclassification**

I. Introduction

In recent years, deep neural networks (DNNs) comprising multiple hidden layers have been shown to produce state-of-the-art results on various perception tasks. Given a large training set, they are able to learn effectively and classify unseen input patterns with extremely high accuracy. As a result, DNNs are widely used in many security-sensitive systems, such as malware detection [1], autonomous driving [2], and biometric authentication [3]. The security of DNNs themselves is therefore of great concern and has received lots of attention. Several recent works [4]–[7] revealed the so-called "adversarial example" problem, i.e., a small perturbation on the input is able to fool DNN to generate adversarial output.

As the outputs of a DNN depend on both input patterns and its own internal parameters, theoretically speaking, we could misclassify a given input pattern into any specific class by manipulating some parameters in the DNN. Practically speaking, while DNNs can be implemented in different platforms (e.g., CPU/GPU [8] and dedicated accelerator [9]), the parameters are usually stored in memory and with the development of precise memory fault injection techniques such as *laser beam fault injection* [10]–[12] and *row hammer attack* [13], [14], it is possible to launch effective fault injection attacks on DNNs. To the best of our knowledge, such threats have not been investigated in the literature.

While it is simple to inject faults to modify the parameters used in DNNs, achieving misclassification for a particular input pattern is a challenging task. First, the large amount of parameters in common DNN hinder designing **efficient** attack method with minimum number of injected faults. Second, changing DNN parameters not only misclassifies the targeted input pattern, but also may misclassify some other unspecified input patterns. Hence, the attack may not be **stealthy**. Given such stealthiness is not always required in different attack scenarios, in this work, we propose two kinds of fault injection attacks on DNN.

- We propose *single bias attack* (SBA) to achieve misclassification by only modifying one parameter of DNN. Since the outputs of DNN linearly depend on the biases in the output layer, SBA can be simply implemented by enlarging the corresponding bias in output layer to the corresponding adversarial class. For DNNs with the widely-used rectified linear unit (ReLU) or similar functions as activation function, we find their outputs also linearly depend on the bias in hidden layers, thereby making SBA effective for fault injection in hidden layers. Note that, as such linear relationship is independent from the inputs of DNNs, it is possible to launch real-time SBA without knowing the exact input patterns, which cannot be achieved with other attacks that requires detailed analysis for the inputs.
- We propose *gradient descent attack* (GDA) to force stealthy misclassification, which tries to preserve the classification accuracy on input patterns other than the targeted one. Given the fact that a DNN inherently tolerates small perturbations on its parameters, GDA introduces slight changes to the DNN's parameters to achieve misclassification, thereby mitigating the impact on the other input patterns. We also propose layer-wise searching and modification compression to further improve the stealthiness and the efficiency, by searching the perturbation at finer granularity and removing insignificant part of the obtained perturbation, respectively.

In our experiments, we evaluate the proposed fault injection attack methods with the well-known MNIST and CIFAR10 datasets. We demonstrate that SBA can achieve misclassification by imposing moderate increment on single bias in any layer of a DNN using ReLU activation function. We also show that GBA can force misclassification and only degrades 3.86 percent and 2.35 percent classification accuracy on the two datasets, respectively.

The remainder of the paper is organized as follows. Sec. II presents the preliminaries of this work. In Sec. III, we give an overview of the proposed SBA and GDA attacks. Sec. IV and Sec. V detail these two kinds of attacks, respectively. Experimental results are then shown in Sec. VI. Next, we discuss future works in Sec. VII and introduce related works in Sec. VIII. Finally, we conclude this work in Sec. IX.

II. Preliminaries

In this section, we first introduce the basics of DNN. Next, we present the commonly-used techniques for memory fault injection. At last, we discuss the threat model and attack objective of this work.

A. Basics of Deep Neural Network

Deep neural networks (DNNs) are large neural networks organized into layers of neurons, corresponding to successive representations of the input data, as shown in Figure 1. The first layer and the final layer in the network are called *input layer* and *output layer* respectively, and the layers in the middle are called *hidden layers*. In the context of classification, each neuron in the output layer corresponds to one class in classification task, i.e., m output neurons for m candidate classes. To predict the probability distribution over classes, a softmax function is appended after the output layer and the class having the largest probability is reported as the final predicted class. Formally,

978-1-5386-3093-8/17/$31.00 ©2017 IEEE

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 23,2022 at 18:29:28 UTC from IEEE Xplore. Restrictions apply.

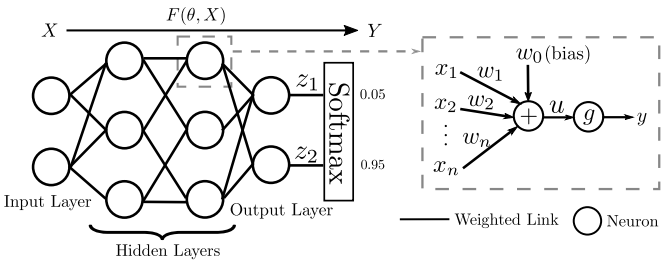

Fig. 1. A 4-layer deep neural network example and the general structure for a neuron.

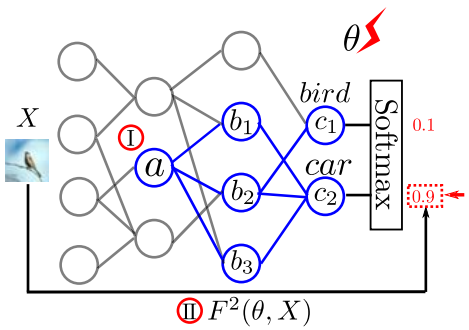

Fig. 2. Fault injection attack on DNN.

the softmax function calculates the probability of given input pattern belonging to class i by

$$softmax(i) = \frac{exp(z_i)}{\sum_j exp(z_j)}, \tag{1}$$

where z_j is the output of j^{th} neuron in output layer.

Neurons are connected by links with different *weights* and *biases*, characterizing the strength between neurons. A neuron receives inputs from many neurons in the previous layer, applies its *activation function* on these inputs, and transmits the result to neurons in next layer. Formally, a neuron's output y is given by

$$y = g(u) \quad \text{and} \quad u = \sum_{1 \le i \le n} w_i x_i + w_0, \tag{2}$$

where $x_1, x_2, \ldots, x_n$ are neuron outputs in the previous layer, $w_1, w_2, \ldots, w_n$ are weights on respective links, w_0 represents the bias, and g is the activation function. In this paper, we also use a neuron's name to indicate its output y for simplicity. If we treat the input layer and final predicted probability distribution on candidate classes as two multidimensional vectors X and Y respectively, then the whole DNN is a parameterized function F, and $Y = F(\theta, X)$, where θ represents all the weights and biases in DNN.

In modern neural networks, the rectified linear unit (ReLU) [15]–[17] is the default recommendation for activation function, defined by $g(u) = max\{0, u\}$. Because the derivatives through ReLU are both large and consistent during backpropagation, it makes the training process easier compared to other kinds of activation functions like $sigmoid$ and $tanh$. There are also some varieties of ReLU which use a non-zero slope α when $u < 0$. Uniformly, all these ReLU-like activation functions can be defined as

$$g(u) = \begin{cases} u & \text{u} \ge 0 \\ \alpha u & \text{u} < 0 \end{cases}. \tag{3}$$

For instance, α is 0 for the original ReLU and α is 0.01 for leaky ReLU [20]. In DNNs for classification, to preserve enough information for softmax function, output neurons(i.e., neurons in output layer) usually use identity function, i.e., $g(u) = u$, as the activation function [18], [19]. The neurons in all hidden layers usually use the same type of activation function. We name DNN using ReLU-like activation function in hidden layers as ReLU-like DNN for short.

B. Fault Injection Techniques

Laser beam and row hammer attacks are two common techniques used for injecting faults into memory. Both of them can alter the logic values in memory with high precision, e.g., single bit flip.

Laser beam can inject fault into SRAM. By exposing the silicon under laser beam, a temporary conductive channel is formed in the dielectric, which in turn causes the transistor to switch state in a precise and controlled manner [10]. By carefully adjusting the parameters of laser beam, like its diameter, emitted energy, and impact coordinate, attacker can precisely change any single bit in SRAM [11], [12]. Previously, laser beam was widely used together with differential fault analysis to extract private key of cipher chips [21].

Row hammer can inject fault into DRAM. It exploits the electrical interactions between neighboring memory cells [13]. By rapidly and repeatedly accessing a given physical memory location, a bit in its adjacent location may flip. By profiling the bit flip patterns in DRAM module and abusing the memory manage features, row hammer can reliably flip a single bit at any address in the software stack [14]. The row hammer has been used in many attacks, e.g., cracking the memory isolation in virtualization [22] and obtaining root privileges in Android system [23].

With the above, existing techniques can inject faults into memory with extremely high precision. However, to inject multiple faults, the laser beam fault injection needs to readjust the laser beam setup [12] and row hammer attack needs to relocate the targeted data in memory [23]. Given the cost of readjustment and relocation, minimizing the number of injected faults is essential to make the attack practical.

C. Threat Model and Attack Objective

Threat Model. In this work, we consider an attacker aiming at changing the output of a DNN-based classification system. We consider white-box attack in this work, that is the attacker has the knowledge of targeted DNN's network structure, the benign parameters and low-level implementation details, e.g., parameters' location in memory. Note that, we do not assume the attacker know the data samples for training or testing. We also assume the attacker can modify any parameter in DNN (i.e., any weight or any bias in DNN) by injecting faults into the memory where the parameters are stored. Given existing fault injection techniques can precisely flip any bit of the data in memory, we assume the attacker can modify any parameter in DNN to any adversarial value that is in the valid range of used arithmetic format.

Attack Objective. Our objective is to find the modification on parameters to misclassify a given input pattern into a specific adversarial class, e.g., misclassifying the bird in the given image as a car as described in Fig. 2. Upon achieving misclassification, we also consider the following two issues:

- **Efficiency.** To make the attack practical, we should modify as few parameters as possible. Though DNN can be implemented on different platforms recording parameters in different arithmetic formats, modifying more parameters tends to require

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 23,2022 at 18:29:28 UTC from IEEE Xplore. Restrictions apply.

injecting more faults. Hence, minimizing the number of modified parameters can reduce the fault injection cost on all platforms.

- **Stealthiness.** Parameter modification may not only misclassify the given input pattern (e.g., the given image), but also misclassify the rest unspecified input patterns (e.g., all the other images). Keeping the attack stealthy for these input patterns is equivalent to preserving original DNN's classification accuracy on these unspecified input patterns as much as possible.

In various attack scenarios, the adversaries may have different requirements on the efficiency and stealthiness. For instance, when adversaries intend to evade malware detection and crash the system, they do not care about the stealthiness and only concern the efficiency of the attack. When adversaries intend to evade a face recognition system without affecting other users, the stealthiness becomes the main concern for the attack. Motivated by the above, we design respective attack methods.

III. Overview

Given attackers may have different requirements on the efficiency and stealthiness, we propose two attack methods, i.e., SBA and GDA. SBA concentrates on achieving high efficiency during attack, while GDA focuses on achieving high stealthiness. In this section, we introduce the basic idea behind each method and the main challenges.

A. Single Bias Attack

According to Eq. 1, the output neuron with the largest output has the largest probability after softmax function. So we can force the output of DNN to be class i, by making the i^{th} output neuron has the largest output among all output neurons. In DNNs for classification, because identify function is used as activation function in output layer, each output neuron's output linearly depends on its bias. Hence, for any input pattern, we can change DNN's output to class i by enlarging the bias of the i^{th} output neuron.

Moreover, we can attack not only the biases in output layer but also the biases in hidden layers, if the activation function g used in hidden layers is a linear function. Consider the attack scenario indicated by ① in Figure 2, where a is a hidden neuron, c_1 and c_2 are the output neurons, and b_1, b_2, and b_3 are the neurons on the paths from a to output layer. Supposing an attacker increasing a's bias, then the outputs of neurons marked by blue are all affected. According to Eq. 2, if g is linear, a neuron's output y linearly depends on the inputs, weights and bias. In this case, a linearly depends on its bias, b_1, b_2, and b_3 also linearly depend on a's bias, and finally c_1 and c_2 linearly depend a's bias as well. In this case, if the derivate of c_2 w.r.t. a's bias is larger than c_1's, c_2 must be larger than c_1 when a's bias becomes sufficient large, and DNN's output is changed to class 2.

Challenges. The practical activation function used in the hidden layers is non-linear, which makes extending SBA to hidden layers non-trivial. However, we find the output neurons in ReLU-like DNN linearly depend on a bias in hidden layers when the bias is sufficiently large, which we call *one side linear phenomenon* as detailed in Section IV-A. Such property enables us to attack biases in hidden layers.

B. Gradient Descent Attack

Because of the inherent fault tolerance capability of DNN, small perturbation on DNN's parameters will not degrade DNN's classification accuracy significantly. Motivated by this, to achieve high stealthiness, we should minimize the perturbation on parameters during attack.

To find such small perturbation, one naive method is using gradient descent to gradually impose perturbation on parameters. To be specific, we can first initialize θ (i.e., all parameters in DNN) with its benign value, denoted by θ_b, and gradually modify θ in the direction of gradient $\frac{dF^i(\theta,x)}{d\theta}$ to enlarge $F^i(\theta, x)$, where $F^i(\theta, x)$ represents the output probability of class i as indicated by ⑪ in Figure 2. Moreover, we can use L1-norm regulator to restrict the amount of perturbation during searching. Formally, the naive method aims at maximizing the following objective function with gradient descent,

$$J(\theta) = F^i(\theta, x) - \lambda|\theta - \theta_b|. \tag{4}$$

Challenges. Because this naive method operates in a greedy way, the obtained result is usually not optimal. First, we find such naive method trends to modify parameters in layer having few parameters and optimizes the objective function in coarse granularity. To tackle this, we propose layer-wise searching to search perturbation in a hidden layer with abundant tunable parameters, so that we can optimize $J(\theta)$ at finer granularity and achieve better stealthiness, detailed in Section V-A. Second, huge number of parameters are modified in naive method. Because the gradient value for most parameters in DNN is not zero, many parameters are likely to be modified with naive method. To tackle this, we propose modification compression to remove insignificant part in obtained perturbation, so that the number of modified parameters is reduced and the stealthiness is improved, detailed in Section V-B.

IV. Single Bias Attack

In this section, we first introduce the one side linear phenomenon and prove that vulnerable biases widely exist in ReLU-like DNNs. Then, we show the overall flow of SBA. At last, we show SBA is easy to launch, analyze SBA's stealthiness, and discuss how to force DNN generating specific adversarial output without knowing the input pattern.

A. One Side Linear and Sink Class

Let us first give the formal definition of one side linear.

Definition 1. *Given two variables x and y, if there exist two constants ϵ and δ such that $\frac{dy}{dx} = \delta$ when $x > \epsilon$, we say y is* **one side linear** *to x and the one side linear slope is δ.*

If multiple variables $y_1, y_2, \ldots, y_n$ are all one side linear to variable x with different slopes, then $y_1, y_2, \ldots, y_n$ will change at different rates when x increases. If the increment on x is larger enough, the variable in $y_1, y_2, \ldots, y_n$ increasing at the fastest rate would finally become the largest one, which is formally proved by Theorem 1.

Theorem 1. *Given variables $y_1, y_2, \ldots, y_n$ which are all one side liner to variable x with slopes $\delta_1, \delta_2, \ldots, \delta_n$ respectively. If $\delta_n > \delta_i$ for $\forall i \neq n$, there must exist a constant ϵ_{sink} such that $y_n > y_i$ for $\forall i \neq n$ when $x > \epsilon_{sink}$.*

Proof. According to Definition 1, for any $i \in [1, n]$, there must exist a constant ϵ_i such that $\frac{dy_i}{dx} = \delta_i$ when $x > \epsilon_i$. Suppose $\epsilon_{max} = max\{\epsilon_1, \ldots, \epsilon_n\}$, and the values of $y_1, y_2, \ldots, y_n$ are $v_1, v_2, \ldots, v_n$ respectively when $x = \epsilon_{max}$. When $x > \epsilon_{max}$, we have $y_i = v_i + \delta_i(x - \epsilon_{max})$.

In this case, to make $y_n > y_i$ for $\forall i \neq n$, we have

$$v_n + \delta_n(x - \epsilon_{max}) > v_i + \delta_i(x - \epsilon_{max})$$
$$x > \epsilon_{max} + \frac{v_i - v_n}{(\delta_n - \delta_i)}.$$

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 23,2022 at 18:29:28 UTC from IEEE Xplore. Restrictions apply.

So when $x > \epsilon_{sink} = max\{\epsilon_{max} + \frac{v_i - v_n}{(\delta_n - \delta_i)} | i \in [1, n-1]\}$, $y_n > y_i$ for $\forall i \neq n$. With above, Theorem 1 is proved. □

In the context of ReLU-like DNN, we have two observations related to one side linear phenomenon: 1) each neuron's output is one side linear to its bias, given by Lemma 1, and 2) if a neuron's all inputs are one side linear to certain bias, then this neuron's output is also one side linear to the bias, given by lemma 2.

Lemma 1. *If h is a neuron in the hidden layers of ReLU-like DNN, then h is one side linear to its bias m with one side linear slope 1.*

Proof. According to Eq. 2, we have $h = g(\sum_{1 \leq i \leq n} w_i x_i + m)$. Given g is ReLU-like function defined in Eq. 3, if $m > |\sum_{1 \leq i \leq n} w_i x_i|$, $h = \sum_{1 \leq i \leq n} w_i x_i + m$. Hence, h is one side linear to m and the slope is 1. □

Lemma 2. *Given a neuron h and a bias m in ReLU-like DNN, if all h's inputs $x_1, x_2, \ldots, x_n$ are one side linear to m with slopes $\delta_1, \delta_2, \ldots, \delta_n$ respectively, then h is also one side linear to m with slope*

$$g\left(\sum_{1 \leq i \leq n} w_i \delta_i\right),$$

where g is the activation function and w_i is the weight corresponding to i^{th} input.

Proof. According to Definition 1, for any $i \in [1, n]$, there must exist a constant ϵ_i such that $\frac{dx_i}{dm} = \delta_i$ when $m > \epsilon_i$. Suppose $\epsilon_{max} = max\{\epsilon_1, \ldots, \epsilon_n\}$ and $u = \sum_{1 \leq i \leq n} w_i x_i + w_0$, where w_0 is h's bias. When $m > \epsilon_{max}$, we have

$$\frac{du}{dm} = \frac{d\left(\sum_{1 \leq i \leq n} w_i x_i + w_0\right)}{m} = \sum_{1 \leq i \leq n} w_i \cdot \frac{dx_i}{dm} = \sum_{1 \leq i \leq n} w_i \delta_i.$$

Suppose $u = u'$ when $m = \epsilon_{max}$. When $m > \epsilon_{max}$, we have

$$u = u' + (m - \epsilon_{max}) \sum_{1 \leq i \leq n} w_i \delta_i.$$

If g is identity function in output layer, $h = u$. Apparently, h is one side linear to m with slope $g(\sum_{1 \leq i \leq n} w_i \delta_i)$. If g is ReLU-like function in hidden layer, we discuss the case that $\sum_{1 \leq i \leq n} w_i \delta_i$ is positive, negative and zero separately.

When $\sum_{1 \leq i \leq n} w_i \delta_i > 0$ and $m > \epsilon_{max} + |\frac{u'}{\sum_{1 \leq i \leq n} w_i \delta_i}|$, we have $u = u' + (m - \epsilon_{max}) \sum_{1 \leq i \leq n} w_i \delta_i > 0$ and $h = u$. Then h is one side linear to m and the slope is $g(\sum_{1 \leq i \leq n} w_i \delta_i)$.

When $\sum_{1 \leq i \leq n} w_i \delta_i < 0$ and $m > \epsilon_{max} + |\frac{u'}{\sum_{1 \leq i \leq n} w_i \delta_i}|$, we have $u = u' + (m - \epsilon_{max}) \sum_{1 \leq i \leq n} w_i \delta_i < 0$ and $h = \alpha u$. So h is one side linear to m and the slope is $g(\sum_{1 \leq i \leq n} w_i \delta_i)$

At last, when $\sum_{1 \leq i \leq n} w_i \delta_i = 0$, apparently h is also one side linear to m and the slope is also $g(\sum_{1 \leq i \leq n} w_i \delta_i)$.

With above, we show h is one side linear to m in all cases and the slope is $g(\sum_{1 \leq i \leq n} w_i \delta_i)$ □

In fact, conclusion similar to Lemma 2 holds for other types of activation function. For instance, the neuron h in widely used max pooling layer simply outputs its largest input, i.e., $h = max(x_1, x_2, \ldots, x_n)$. Suppose $x_1, x_2, \ldots, x_n$ are all one side linear to a bias m and x_n has the largest one side linear slope. According to Theorem 1, there must exist ϵ_{sink} such that x_n is consistently larger than $x_1, x_2, \ldots, x_{n-1}$ when $m > \epsilon_{sink}$, which means $h = x_n$ in this case. So h is also one side linear to m and the slope is the same as x_n's.

With above, we show the one side linear property can propagate from previous layer to next layer. By induction, the output neurons are one side linear to any bias.

Theorem 2. *In ReLU-like DNN, for a bias m in hidden layers, every output neuron is one side linear to m.*

Proof. Suppose the DNN has N layers and m is in the i^{th} layer. The neurons in the same layer are independent. Given a neuron in the i^{th} hidden layer, it is either independent with m or using m as its bias. A neuron independent with m is one side linear to m with slope 0, and a neuron using m as bias is one side linear to m given Lemma 1.

With above, all the neurons in i^{th} layer are one side linear to m. Because the neurons in $(i+1)^{th}$ layer only depends on neurons in i^{th} layer, all neurons in $(i+1)^{th}$ layer are also one side linear to m according to Lemma 2. By induction, all neurons in the output layer are one side linear to m. □

Sink Class. With Theorem 1 and Theorem 2, if one output neuron has the largest one side linear slope w.r.t. a given bias among all the output neurons, increasing this bias would finally make the DNN's output converge at the class indicated by this output neuron. We use **sink class** to represent the output class at which DNN's output converges when a bias increases. During proving Theorem 2 by induction, we find the one side line slope of a output neuron w.r.t. a bias is independent with the input pattern. Hence, one bias' sink class is the same for different input patterns.

B. Overall Attack Flow

The overall flow of single bias attack consists of two steps, as shown in Figure 3(a). First, we profile the sink class for each bias for the given DNN. Next, according to the specific adversarial mapping, we only need to choose one bias whose sink class is the same as the targeted class in the adversarial mapping and find the required increment on this bias to saturate DNN's output.

Profiling Sink Class. To obtain the sink class for a bias, we only need each output neuron's one side linear slope w.r.t. this bias and check whether unique largest slope exists. We can calculate the slope for each output neuron w.r.t. a given bias m by induction. At first, we need to initialize the slopes for all neurons in i^{th} layer where m is located. If a neuron uses m as its bias, we set the slope for this neuron to be 1, otherwise it is 0. Then we can calculate the slopes layer by layer according to Lemma 2. Finally, we can obtain the one side linear slope for each output neuron. If unique largest slope exists in the output layer, we can determine the sink class for m.

Saturating DNN. To achieve the adversarial mapping, we can choose any bias m whose sink class is the same as the targeted class in the specific adversarial mapping. Then to make DNN output the m's sink class, we only need to modify m so that m is larger the ϵ_{sink} value defined in Theorem 1. We can determine the exact ϵ_{sink} value for a given input pattern and selected bias m, by scanning m values and monitoring when DNN's output converges. Given we can check the ϵ_{sink} for each candidate bias, we can also use the bias with the smallest ϵ_{sink} during attack.

C. Discussion

During practical attack, we do not need to precisely modify the bias, i.e., enlarging the bias beyond ϵ_{sink} is sufficient. To achieve this, we may only need to inject several single bit faults on the most significant bits of the bias. For instance, when floating point format

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 23,2022 at 18:29:28 UTC from IEEE Xplore. Restrictions apply.

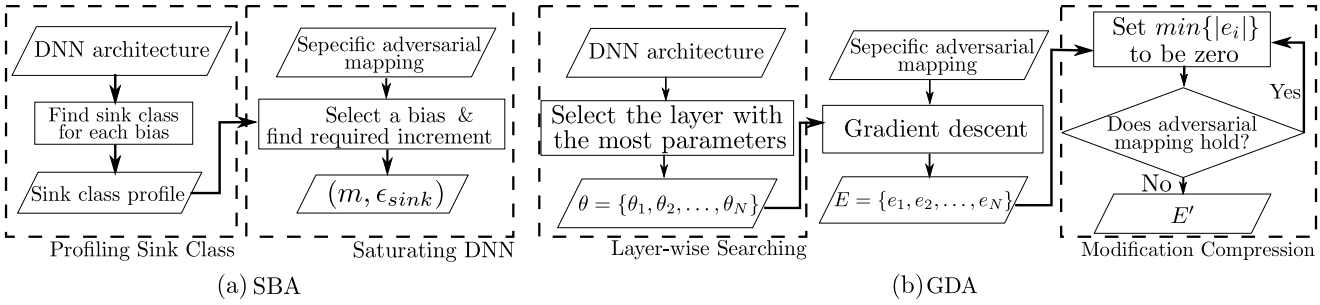

Fig. 3. The overall flow for (a)SBA and (b)GDA.

is used to store parameter, we can efficiently enlarge the bias by magnitudes via making its sign bit positive and setting several most significant bits in the exponent field.

The stealthiness of single bias attack depends on the ϵ_{sink} value for the input pattern. Suppose an attacker can precisely modify the bias to be slightly larger than the ϵ_{sink}. Different input patterns may require different ϵ_{sink} values to saturate DNN's output when attacking the same bias. When misclassifying a input pattern with $\epsilon_{sink} = k$, the outputs for other input patterns with $\epsilon_{sink} < k$ on this bias must converge and the outputs for input patterns with $\epsilon_{sink} > k$ may be also affected although not converge. Hence, the accuracy degradation is larger when attacking input pattern with larger ϵ_{sink} on a bias. To mitigate this, one possible solution is to recover the DNN to benign state with additional faults in addition to the few faults injected during SBA.

Note that, even without knowing the exact input pattern, it is still possible to force DNN generating specific adversarial output with SBA. Because the sink class of a bias is independent from input patterns, we could saturate DNN by modifying the bias to be a very large valid value allowed by the arithmetic format used. When it is larger than the required ϵ_{sink} for an input pattern, we can successfully change the DNN's output. Such attack can be launched at real-time (without necessarily conducting input pattern analysis), leading to possible denial-of-service.

V. Gradient Descent Attack

The overall flow of gradient descent attack consists of three steps, as shown in Figure 3(b). At first, we determine the set of parameters to be updated in gradient descent. Instead of updating all the parameters in DNN and searching the modification global-wise, we search modification within a single layer, i.e., layer-wise searching. Next, we optimize Eq. 4 with gradient descent to find the modification on each selected parameter, denoted by $E = \{e_1, e_2, \ldots, e_N\}$. At last, we repeatedly replace the smallest element in E with zero until the final parameter modification E' is obtained, i.e, modification compression.

In the following two subsections, we discuss layer-wise searching and modification compression in detail.

A. Layer-wise Searching

We found the magnitude of parameter's gradient is not uniformly distributed among layers, and parameters in layer having fewer parameters trend to have gradient of larger magnitude. For instance, Figure 4 shows the average magnitude of parameter's gradient and parameter count in each layer for the MNIST model detailed in Section VI-A, and `layer2` with the fewest parameters has the largest gradient. Because gradient descent modifies each parameter according

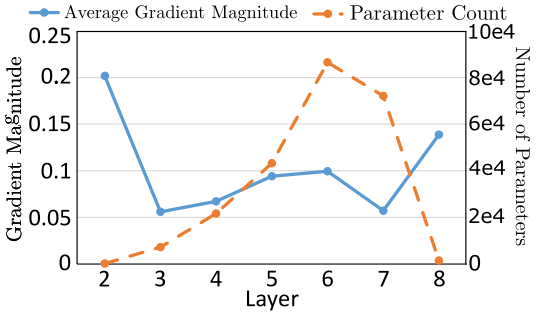

Fig. 4. The distribution of parameter's gradient and parameter count.

to its gradient, parameter with gradient of larger magnitude receives larger modification during searching. As a consequence, the naive method which is supposed to find optimal modification global-wise in fact focuses on modifying the few parameters in the smallest layer, e.g., `layer2` in Figure 4. In this way, the naive method optimizes the objective function by adjusting the mapping F in coarse granularity and the obtained solution is not optimal.

To tackle this problem, we propose layer-wise searching. To be specific, instead of finding the modification global-wise, we manually restrict the θ in Eq. 4 to be the set of all parameters within one layer. Intuitively, a layer with more parameters allows us to adjust the mapping F at finer granularity and optimize the objective function better, i.e., introducing smaller perturbation. While the naive global-wise searching in fact focuses on adjusting the smallest layer, we can conduct layer-wise searching on the layer with the most parameters to search the solution at the finest granularity. In Section VI-C, we demonstrate conducting layer-wise searching on any layer, except the smallest layer, can achieve higher stealthiness than the global-wise searching.

B. Modification Compression

Because gradient search is a greedy method, the obtained perturbation list E is not optimal even with our layer-wise searching technique, which implies some perturbations in E are redundant. Motivated by this, modification compression aims at removing the insignificant part of E to further reduce the overall perturbation, that is replacing some non-zero elements in E with zero.

Considering changing E too much may destroy the required adversarial mapping, we achieve the replacement by iteration as shown in Figure 3(b). At each iteration step, we only replace the element with smallest absolute value in E to be zero and check whether the adversarial mapping is still preserved. If it is preserved, we continue

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 23,2022 at 18:29:28 UTC from IEEE Xplore. Restrictions apply.

TABLE I. The architectures of the MNIST model and the CIFAR model.

Layer	MNIST	CIFAR
1	28*28 image	32*32 RGB image
2	3*3 conv. 20 $tanh$	3*3 conv. 96 ReLU
3	3*3 conv. 40 $tanh$ 2*2 max pooling	3*3 conv. 96 ReLU 2*2 max pooling
4	3*3 conv. 60 $tanh$	3*3 conv. 192 ReLU
5	3*3 conv. 80 $tanh$ 2*2 max pooling	3*3 conv. 192 ReLU 2*2 max polling
6	3*3 conv. 120 $tanh$	3*3 conv. 192 ReLU
7	fully connect. 150 $tanh$	1*1 conv. 192 ReLU
8	fully connect. 10 identity	1*1 conv. 10 identity global average pooling
softmax function		

to explore further replacement. Otherwise, only previously found replacements are finally used.

With modification compression, we can not only reduce the number of modified parameters but also improve the stealthiness, because replacing the elements in E with zero reduces the amount of perturbation imposed.

VI. Experimental Results

A. Experimental Setup

Dataset. All the experiments are performed on two machine learning datasets: MNIST and CIFAR10. The MNIST dataset is a collection of 70000 black-white images of handwritten digit, where each pixel is encoded as a real number in $[0, 1]$. The classification goal is to determine the digit written in the given image. So the candidate classes range from 0 to 9. The CIFAR10 dataset is a collection of 60000 nature color images in RGB format, where the value of each pixel on each channel is encoded as an integer in $[0, 255]$. The images are to be classified in one of the 10 mutually exclusive classes, e.g., airplane, bird, and cat. We also label these 10 classes from 0 to 9 for short.

DNN Model. For each dataset, we implement one DNN model, detailed in Table I. The MNIST model from [18] is an 8 layer DNN for MINST dataset, which uses $tanh$ as activation function. The CIFAR model from [19] is an 8 layer DNN for CIFAR10 dataset, which uses ReLU as activation function. In each model, the `Layer1` and `Layer8` are the input layer and output layer respectively. The MNIST model and CIFAR model can achieve 99.06% and 84.01% classification accuracy with benign parameter respectively, which are comparable to the state-of-the-art results.

Platform. All the experiments are conducted on a machine equipped with an Intel Xeon E5 CPU and a NVIDIA TitanX GPU. We implement the DNN models with Theano [8] library, and Theano uses default float32 precision to store the parameters and intermediate results.

B. Evaluation for SBA

In this section, we evaluate SBA on the CIFAR model which is a ReLU-like DNN. We first demonstrate biases having sink class widely exist in DNN. Then we show how DNN's output converges as we increase a bias. At last, we evaluate the magnitude of ϵ_{sink} required to saturate the DNN and the stealthiness of the attack.

At first, we found every bias in the CIFAR model had a sink class, and the distribution of sink class within each layer is shown in Figure 5(a). We observe that every layer in the CIFAR model can provide all the ten kinds of sink classes. Hence, once we can inject fault into one layer, we can achieve any specific maliciously mapping.

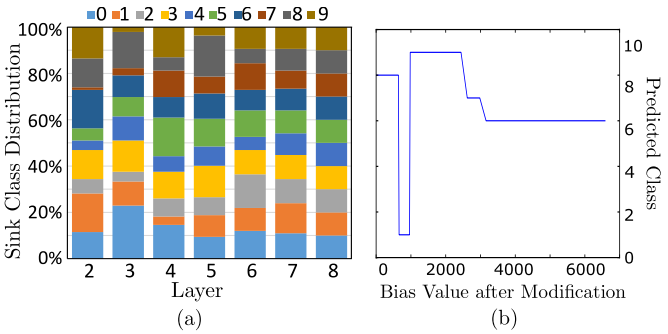

Fig. 5. (a) Sink class distribution in the CIFAR model, where each color represents one kind of sink class, and (b) the convergence of CIFAR model's output as a bias increases.

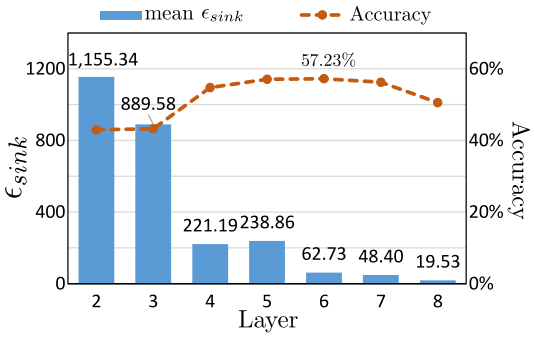

Fig. 6. The mean ϵ_{sink} in each layer and the corresponding classification accuracy after attack.

Next, to verify CIFAR model's output would converge at a bias's sink class as we increase the bias, we test every bias in the CIFAR model with about 100 randomly selected input patterns. In all the cases, the CIFAR model's output successfully converges at the expected sink class. Figure 5(b) shows the convergence procedure of CIFAR's output in a random selected case, which can be divided into two stages. In the early stage, the model's output oscillates when the bias increases, because the classification function intended to be learned by the DNN is non-linear. Then, as we increase the bias beyond the ϵ_{sink} value, the output converges at the sink class of the bias.

At last, to show the increment on bias required by SBA is practical, we examine the biases for each layer individually. For each layer, we randomly select about 1000 input patterns for misclassification, and we record the minimal ϵ_{sink} for each pattern when using biases in this layer. In Figure 6, the bar chart shows the mean of such minimal ϵ_{sink} values for each layer, and the dashed curve plots the average classification accuracy after attack where we modify the bias slightly larger than the ϵ_{sink}. Given Figure 6, we have two observations. First, the average ϵ_{sink} for each layer is definitely within the valid range of the *float32* precision, which can represent number as large as $3e38$. Especially for `Layer6` $\sim$ `Layer8`, the average ϵ_{sink}s are smaller than 70, which are exploitable on platforms using simpler arithmetic format, e.g., 16-bit fixed-point number with 8-bit fraction. Second, we find the highest accuracy 57.23% is achieved by attacking `Lyaer6`. However, attacking `Layer6` in fact imposes larger perturbation on the bias compared to attacking `Layer8`, given the magnitude of every bias's benign value in our CIFAR model is smaller than one. One possible reason is that, the gradient of the output w.r.t. a bias in `Layer6` is smaller than that for `Layer8`, similar to the case indicated by Figure 4. Hence, although attacking `Layer6` imposes

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 23,2022 at 18:29:28 UTC from IEEE Xplore. Restrictions apply.

TABLE II. The classification accuracy(CA) after attack and the number of modified parameters(# of MP) during attack, where 'LW+digit' represents layer-wise searching on corresponding layer, 'MC' is short for modification compression, and '§' indicates the results for naive gradient descent method.

	MNIST				CIFAR			
	CA		# of MP		CA		# of MP	
	w/o MC	MC	w/o MC	MC	w/o MC	MC	w/o MC	MC
LW 2	46.38%	59.89%	200	19	12.98%	25.06%	2334	283
LW 3	56.22%	68.62%	7240	221	12.98%	54.54%	57009	1354
LW 4	58.80%	84.93%	21660	1077	25.34%	76.45%	129759	697
LW 5	46.07%	90.44%	43280	1215	23.39%	73.73%	195502	2321
LW 6	65.23%	95.20%	86520	2345	11.68%	81.66%	115127	198
LW 7	89.88%	97.01%	72150	5734	13.87%	80.57%	19109	43
LW 8	95.12%	96.86%	1439	125	13.02%	80.32%	1147	2
Global-wise	26.68%(§)	63.70%	232559(§)	1170	10.00%(§)	50.97%	519691(§)	425

larger perturbation on a bias, it changes DNN's output less.

C. Evaluation for GDA

In this section, we evaluate GDA on both MNIST model and CIFAR model. We examine conducting layer-wise searching on each layer, while modification compression is present or absent. Table II shows the number of modified parameters and the classification accuracy after attack in all the cases.

At first, to evaluate the effectiveness of layer-wise searching, we compare the data within each column in Table II. First, by comparing the data in third column and seventh column respectively, we observe layer-wise searching on any layer except `Layer2` can achieve higher accuracy than the global-wise searching. According to Table I, in both models, any layer in `Layer3`∼`Layer8` has more parameters than `Layer2`. Hence, attacking a layer with abundant parameters is more stealthy than simply using global-wise searching. Second, by comparing the data in fifth column and ninth column respectively, we observe layer-wise searching on layers close to input layer or output layer modifies fewer parameters than layers in the middle of DNN. This is because the layers close to the two ends have much fewer parameters than layers in the middle according to Table I, hence fewer parameters are modified during gradient descent.

Next, to evaluate the effectiveness of modification compression, we examine the data within each row in Table II. On the one hand, we observe, no matter layer-wise searching or global-wise searching is used, conducting modification compression can always significantly reduce the number of modified parameters. Without modification compression, almost all tunable parameters are modified in each case. On the other hand, we observe modification compression can always improve the classification accuracy, which also demonstrates minimizing perturbation on parameters indeed helps improve the stealthiness.

Finally, by conducting layer-wise searching on `Layer6`, the largest layer containing the most parameters in both models, together with modification compression, GDA can achieve classification accuracy as high as 95.20% and 81.66% for each model, which only degrades the benign accuracy by 3.86 percent and 2.35 percent respectively. Compared to the naive method discussed in Section III-B, whose results are indicated by '§' in Table II, GDA can averagely reduce the number of modified parameters by 99.50%, and improve the classification accuracy by 67.97 percent.

D. SBA VS. GDA

We compare SBA and GDA based on the results for CIFAR model in Figure 6 and Table II. First, we observe SBA is more efficient than GDA, because SBA only modifies one parameter and GDA modifies 198 parameters when attacking `Layer6`. Second, by examining the accuracy curve in Figure 6, we observe GDA is more stealthy than SBA, given GDA can achieve 81.66% accuracy while SBA's average accuracy is below 60%. One possible reason is that, GDA imposes small perturbations on multiple parameters instead of imposing large perturbation on a single parameter like SBA, and the total perturbation on parameters imposed by GDA is smaller than SBA.

VII. Discussion and Future Works

In this section, we discuss potential countermeasures to the proposed attacks and possible future work.

We can detect whether DNN suffers fault injection attack by checking whether the stored parameters are modified. To achieve this, we can apply error detection code (EDC) on parameters, e.g., parity check. Because DNNs generally contain a very large number of parameters, however, the cost of checking all parameters' status via EDC could be prohibitively high. As a direction of future study, one can investigate the vulnerabilities of different parameters under attack and only protect those more vulnerable ones with EDC, which can achieve a good balance between fault detection overhead and fault detection capability.

We can mitigate SBA's threat on ReLU-like DNN by optimizing DNN's architecture. SBA relies on the one side linear property of ReLU-like function to propagate the large increment on internal bias to DNN's final output, so that the output neuron with the largest slope would finally become the largest one. However, such large value propagation is not possible for activation function like *tanh*, because *tanh*'s output is always between -1 and 1. Hence, as a direction of future work, one can study how to allocate ReLU-like and other types of activation functions in the same DNN, so that we can not only preserve ReLU-like function's advantage in simplifying training procedure but also mitigate SBA.

We may reduce the required increment on a bias in SBA by modifying more biases instead of one, which may also improve the stealthiness of SBA. In Section VI-D, we found GDA can achieve higher stealthiness than SBA by imposing smaller perturbations on multiple parameters. Similarly, we may enhance SBA by imposing small perturbations on multiple biases instead of one large perturbation on single bias. As a direction of future work, one can study how to select the set of multiple biases and allocate perturbations on these biases to efficiently saturate DNN's output.

It is also possible to attack DNN with low-precision fault injection techniques. With low-precision fault injection, attacker cannot modify the parameter precisely. For instance, the bits maybe flipped with uncertainty within the fault-affected area. Because SBA doesn't need to modify the bias exactly and increasing the bias beyond its ϵ_{sink} is sufficient, SBA may still succeed in this case if we can guarantee

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 23,2022 at 18:29:28 UTC from IEEE Xplore. Restrictions apply.

the bias is increased during attack, e.g., injecting faults into a zero byte. As a direction of future study, one can investigate how to attack DNN with low-precision fault injection techniques in practice.

VIII. Related Works

In this section, we briefly discuss the related works on adversarial example and techniques for improving DNN's fault-tolerance capability against random faults.

The state-of-the-art neural networks are vulnerable to adversarial examples [4]. That is, small but intentionally perturbation on input forces neural network to provide adversary output, and such perturbation for an input pattern can be crafted by methods like L-BFGS [4] and fast-gradient-sign [5]. Given adversarial example and fault injection attack proposed in this work target on DNN's input and internal parameters respectively, adversarial example is orthogonal to our proposed attacks. To defeat adversarial example problem, recent countermeasures aim at improving the generalization capability of neural network by manipulating DNN's training procedure. For instance, Papernot *et al.* [6] proposed to train the same neural network twice and the soft labels used in the second train process are able to improve the generalization capability. Gu and Rigazion [7], and Goodfellow *et al.* [5] proposed to regularize the objective function in training procedure to manually restrict the predicted class for data points around training samples.

Though no previous works have studied fault injection attack on DNN, there are many works improving DNN's fault-tolerance capability against random faults on the weights. These works can be mainly divided into three groups, i.e., training with artificial fault [24], [25], neuron duplication [26], [27], and weight restriction [28], [29]. Training with artificial fault intentionally injects faults into DNN during training process, so that obtained DNN can perform correctly even in faulty cases [24], [25]. However, the cost of enumerating all faulty cases is prohibitive when the number of faulty cases is exponentially larger, like enumerating multi-bit faults. Hence, such method is inefficient for defeating SBA and GDA as well, where the number of exploitable faulty cases is large. Neuron duplication improves the redundancy of DNN by duplicating internal neurons and scaling down corresponding weights [26], [27]. Such method is specially designed for stuck@0 fault. Because the magnitude of each weight is scaled down, the perturbation imposed by stuck@0 fault is reduced and expected to be tolerated. However, adversaries can intentionally change a weight's value and control the amount of perturbation introduced, so such method is inefficient for malicious faults. At last, weight restriction determines a range to which weights should belong during training, and any weight being outside the range is forced to be its upper limit or lower limit [28], [29]. In this way, weight restriction can mitigate the interference from fault imposing large perturbation on a weight. Intuitively, weight restriction can mitigate SBA, but it is less effective for defeating GDA. This is because SBA need to impose large perturbation on one parameter, while GDA imposes small perturbations on multiple parameters.

IX. Conclusion

In this paper, we present two kinds of fault injection attacks on DNN. SBA is able to achieve misclassification by modifying only one bias in the network. By combining DNN's fault tolerance capability with the proposed layer-wise searching and modification compression techniques, GDA is able to achieve high stealthiness and efficiency at the same time. Experimental results demonstrate the effectiveness of the proposed attacks on DNN.

Acknowledgement

This work is supported in part by Project No. 61432017 and Project No. 61532017 by National Natural Science Foundation of China (NSFC), in part by Huawei Technologies Co. Ltd., and in part by Shanghai Inbestech Co. Ltd.

References

[1] G. E. Dahl, *et al*. Large-scale malware classification using random projections and neural networks. In *Proc. of International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[2] D. C. Ciresan, *et al*. Multi-column deep neural network for traffic sign classification. In *Neural Networks*, 2012.

[3] Y. Taigman, *et al*. Deepface: Closing the gap to human-level performance in face verification. In *Proc. of Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[4] C. Szegedy, *et al*. Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199, 2013.

[5] I. J. Goodfellow, *et al*. Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572, 2014.

[6] N. Papernot, *et al*. Distillation as a defense to adversarial perturbations against deep neural networks. In *Proc. of Symposium on Security and Privacy (SP)*, 2016.

[7] S. Gu and L. Rigazio. Towards deep neural network architectures robust to adversarial examples. arXiv preprint arXiv:1412.5068, 2014.

[8] J. Bergstra, *et al*. Theano: A CPU and GPU math compiler in python. In *Proc. of Python in Science Conf*, 2010.

[9] T. Chen, *et al*. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[10] A. Barenghi, *et al*. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. In *Proc. of the IEEE*, 2012.

[11] B. Selmke, *et al*. Precise laser fault injections into 90 nm and 45 nm sram-cells. In *Proc. of Smart Card Research and Advanced Applications (CARDIS)*, 2015.

[12] M. Agoyan, *et al*. How to flip a bit?. In *Proc. of International On-Line Testing Symposium (IOLTS)*, 2010.

[13] Y. Kim, *et al*. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proc. of International Symposium on Computer Architecture (ISCA)*, 2014.

[14] K. Razavi, *et al*. Flip feng shui: Hammering a needle in the software stack. In *Proc. of USENIX Security Symposium*, 2016.

[15] I. Goodfellow, *et al*. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] K. Jarrett, *et al*. What is the best multi-stage architecture for object recognition? In *Proc. of International Conference on Computer Vision (ICCV),* 2009.

[17] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proc. of International Conference on Machine Learning (ICML)*, 2010.

[18] D. C. Ciresan, *et al*. Flexible, high performance convolutional neural networks for image classification. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

[19] J. T. Springenberg. *et al*. Striving for simplicity: The all convolutional net. arXiv preprint arXiv:1412.6806, 2014.

[20] A. L. Maas, *et al*. Rectifier nonlinearities improve neural network acoustic models. In *Proc. of International Conference on Machine Learning (ICML)*, 2013.

[21] C. Giraud. DFA on AES. In *Advanced Encryption Standard - AES*, 2004.

[22] Y. Xiao, *et al*. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *Proc. of USENIX Security Symposium*, 2016.

[23] V. van der Veen, *et al*. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proc. of Conference on Computer and Communications Security (CCS)*, 2016.

[24] T. Ito and I. Takanami. On fault injection approaches for fault tolerance of feedforward neural networks. In *Proc. of Asian Test Symposium (ATS)*, 1997.

[25] I. Takanami, *et al*. A fault-value injection approach for multiple-weight-fault tolerance of mnns. In *Proc. of International Joint Conference on Neural Networks*, 2000.

[26] D. Phatak and I. Koren. Fault tolerance of feedforward neural nets for classification tasks. In *Proc. of International Joint Conference on Neural Networks*, 1992.

[27] D. S. Phatak and I. Koren. Complete and partial fault tolerance of feedforward neural nets. In *IEEE Transactions on Neural Networks*, 1995.

[28] N. Kamiura, *et al*. An improvement in weight-fault tolerance of feedforward neural networks. In *Proc. of Asian Test Symposium (ATS)*, 2001.

[29] N. Kamiura, *et al*. Learning based on fault injection and weight restriction for fault-tolerant hopfield neural networks. In *Proc. of International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, 2004.

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 23,2022 at 18:29:28 UTC from IEEE Xplore. Restrictions apply.