SNIFF: Reverse Engineering of Neural Networks with Fault Attacks

Jakub Breier, Dirmanto Jap, Xiaolu Hou, Shivam Bhasin and Yang Liu

***Abstract*—Neural networks have been shown to be vulnerable against fault injection attacks. These attacks change the physical behavior of the device during the computation, resulting in a change of value that is currently being computed. They can be realized by various techniques, ranging from clock/voltage glitching to application of lasers to rowhammer. Previous works have mostly explored fault attacks for output misclassification, thus affecting the reliability of neural networks. In this paper we investigate the possibility to reverse engineer neural networks with fault attacks. Sign bit flip fault (SNIFF) attack enables the reverse engineering by changing the sign of intermediate values. We develop the first exact extraction method on deep-layer feature extractor networks that provably allows the recovery of proprietary model parameters. Our experiments with Keras library show that the precision error for the parameter recovery for the tested networks is less than 10^{-13} with the usage of 64-bit floats, which improves the current state of the art by 6 orders of magnitude.**

***Index Terms*—Neural networks, deep learning, reverse engineering, fault attacks**

I. Introduction

Neural networks form a basis for current artificial intelligence applications. They were shown to be effective in domains that can provide large amount of labeled data to be able to learn the classification model with sufficient level of accuracy. Because of this property, companies often protect their models as the cost of obtaining the data used to train them might be very high, while the availability of such data is limited. Thus, having a classification model whose internal parameters are secret gives companies a competitive advantage. It is therefore necessary to know the ways that enable reverse engineering of the models so that adequate protection could be applied.

Model stealing attacks (also called model extraction attacks) aim at retrieving the model parameters in a black-box settings [1]. In this setting, the attacker sends inputs to the network and observes the outputs. Based on this information, she tries to reconstruct the model that has accuracy close to the original one. In a similar fashion, it is possible to recover the hyperparameters of machine learning models in general [2].

There are certain similarities when it comes to comparing the model stealing attacks with the key recovery attacks on cryptography. Classical cryptanalysis works by querying the cryptosystem with inputs and observing the outputs. This helps in getting the information about the secret key. In the field of cryptography, researchers started observing the physical characteristics of the devices that perform the encryption to find the secret key more efficiently. Similarly, it was shown that by causing errors during the cryptographic computation, the attacker can learn secret information [3]. We call these implementation-level attacks *physical attacks* on cryptography.

Now, we can look into the emerging area concerned with physical attacks against neural networks. It was shown earlier that side-channel attacks can be applied to neural networks to recover certain model parameters [4]. It was also shown that neural networks are vulnerable to fault injection attacks that change the intermediate values of the model during the computation, enabling misbehavior of the activation functions in the model [5]. As the fault might also be introduced through external factors, the reliability of the neural network implementations is becoming a growing concern. In some cases, a single fault occurring in a GPU could reduce the reliability of a CNN performance [6]. Thus, this could also be exploited by the adversary. If we change the intermediate values, the model output will change, potentially revealing the information about the model parameters. We focus on utilizing this behavior to fully recover the values of the internal parameters of the neural network. More specifically, we utilize a fault that changes the sign of the intermediate values to get the information, hence the name *SNIFF – sign bit flip fault.*

Our contribution. In this paper, we present a way to reverse engineer neural networks with the help of fault injection attacks. More specifically, we target deep-layer feature extractor networks produced by transfer learning, to recover the parameters (weights and biases) of the last hidden layer. Our work mainly focuses on neural network classifiers based on the widely used softmax activation function in the output layer. On top of that, we also show application to other activation functions. Our method provably allows *exact extraction*, meaning that the exact values of parameters can be determined after the fault attack. Thus, in case of a deep-layer feature extractor, this allows to get the exact information on the entire network. We note that this is the first work using fault injection attack for the model extraction, and also the first work allowing

J. Breier is with Silicon Austria Labs, TU-Graz SAL DES Lab and Graz University of Technology, Graz, Austria. E-mail: jbreier@jbreier.com

D. Jap and S. Bhasin are with Temasek Laboratories, Nanyang Technological University, Singapore. E-mail: {djap, sbhasin}@ntu.edu.sg

X. Hou is with Faculty of Informatics and Information Technologies, Slovak University of Technology, Slovakia. E-mail: houxiaolu.email@gmail.com

Y. Liu is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. E-mail: yangliu@ntu.edu.sg

This work has been supported in parts by the "University SAL Labs" initiative of Silicon Austria Labs (SAL) and its Austrian partner universities for applied fundamental research for electronic based systems. The authors acknowledge the support from the Singapore National Research Foundation ("SOCure" grant NRF2018NCR-NCR002-0001 – www.green-ic.org/socure). This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under the Programme SASPRO 2 COFUND Marie Sklodowska-Curie grant agreement No. 945478.

exact extraction. In terms of *accuracy* and *fidelity* defined recently in [7], our work achieves perfect scores in both, as the extracted values are identical to the original network.

II. Preliminaries

This section recalls general concepts used in the rest of the paper. The target datasets and experimental setup are also discussed.

A. Fault Injection Methods

Fault injection can be performed with a variety of equipment based on the required precision, cost and impact.

Clock/voltage glitch methods offer limited precision and are normally used to alter the control flow of the program rather than disturbing the data directly. This is often referred to as *global* fault injection.

Electromagnetic (EM) emanation is more localized method, where the precision heavily depends on the resolution of the injection probe. It was shown that precise bit sets and resets in memory cells can be achieved [8].

Optical radiation includes methods with varying precision, using equipment ranging from camera flashes to lasers. The advantage is high reproducibility of faults and great precision – precise bit flips were shown to be possible with lasers.

Rowhammer [9] and *Voltpwn* [10] are fault injection methods that do not require a dedicated injection device. Such attacks exploit memory and microarchitectural properties for fault injection, and allow bit flips and software controlled faults.

Besides these, there are other less researched fault injection methods, such as X-rays/gamma rays [11], or hardware trojans [12].

B. Fault Injection on Neural Networks

The seminal work in the field of adversarial fault injection was published by Liu et al. in 2017 [13]. They introduced two types of attacks: *single bias attack* changes the bias value in either one of the hidden layers (in case of ReLU or similar activation function) or output layer of the network to achieve the misclassification; while *gradient descent attack* works in a similar way as Fast Gradient Sign Method [14], but changes the internal parameters instead of the input to the network.

Practical fault injection by using a laser technique was shown by Breier et al. in 2018 [5]. They were able to disturb the instruction execution inside the general-purpose microcontroller to achieve the change of the neuron output. In their paper, they focused on behavior of three activation functions: in case of sigmoid and tanh, the fault resulted in an inverted output, while in case of ReLU, the output was forced to be always zero. The work was further extended in [15] and [16] to show different attack strategies to improve the misclassification efficiency.

A comprehensive evaluation of bitwise corruptions on various deep learning models was presented by Hong et al. in 2019 [17]. They showed that most models have at least one parameter such that if there is a bit-flip introduced in its bitwise representation, it will cause an accuracy loss of over 90%.

Malicious bit-flips were further investigated for various misclassification/model degradation attacks in [18], [19], [20], [21].

When it comes to fault and error tolerance of neural networks, we would point interested reader to a survey written by Torres-Huitzil and Girau in 2017 [22], which provides exhaustive overview of this topic.

C. Transfer Learning

Transfer learning takes a pre-trained *teacher* model and transfers the knowledge (model architecture and weights) to a *student* model. The requirement is to have a similar task for the newly trained student model compared to the teacher model. Transfer learning is normally achieved by "freezing" the first $n-k$ layers of the teacher model out of the total number of n layers – by fixing the values of the weights. Then, the remaining k layers are removed and new layers are added to the end of the student model. These layers are then trained on the new data. There are 3 main approaches that are used in transfer learning [23]:

- *Deep-layer Feature Extractor:* in this approach, the first $n-1$ layers are frozen and only the last layer is updated, as can be seen in Figure 1. It is normally used when the student task is very similar to the teacher task. It allows very efficient training. In the rest of the paper, we will be focusing on the *secret parameter recovery* of this approach.
- *Mid-layer Feature Extractor:* this approach freezes the first $n-k$ layers, where $k < n-1$. It can be used in case the student task is less similar to the teacher task and there is enough data to train the Student.
- *Full Model Fine-tuning:* in this approach, all the layers are unfrozen and updated during the student training. It requires sufficient amount of data to fully train the student, and is normally used for cases where student task differs significantly from the teacher task.

Important observation when recovering the student model is that the layers copied from the teacher are publicly known, and therefore it is possible to derive the output values for all the frozen layers for any input. This way, we know the inputs to the layers trained by student, and the outputs from the model. Based on this information, we are able to design a weight recovery attack assisted by fault injection.

D. Model Extraction

If we consider $\mathcal{O}(\cdot)$ to be the original neural network model we want to extract, $\hat{\mathcal{O}}(\cdot)$ denotes the extracted model. Jagielski et al. [7] developed a taxonomy for model extraction attacks and differentiate four different extraction types:

- *Exact Extraction:* strongest type of extraction, where $\hat{\mathcal{O}} = \mathcal{O}$, that is, both the architecture and the weights of the extracted model have the same values as the original network. Exact extraction brings several advantages to the adversary over other extraction types. Firstly, it aides in computing perfect adversarial examples [24], which is considered one of the most powerful and stealthy exploits

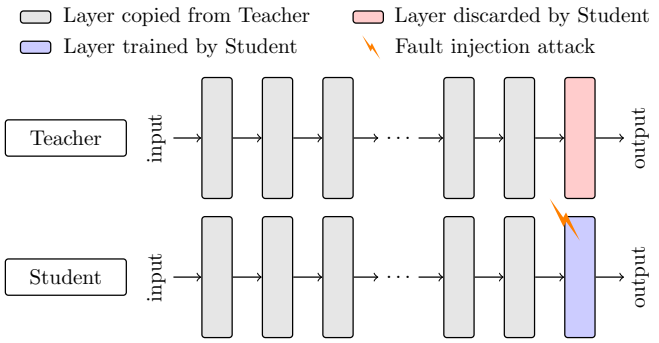


Fig. 1. Transfer learning using deep-layer feature extractor and fault injection into the student model for recovering the newly added layer.

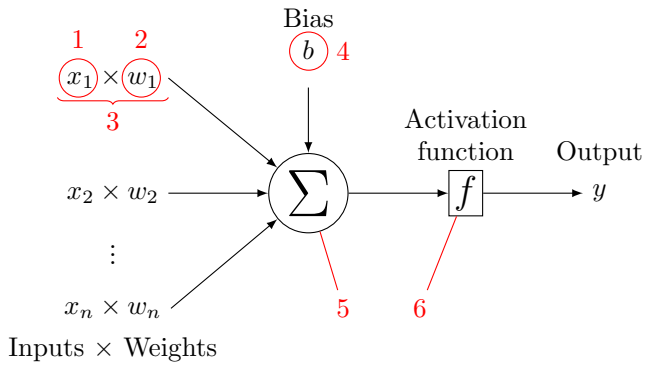


Fig. 2. Neuron computation of neural networks.

against neural networks. Secondly, the knowledge of exact model also reveals information on training data [25] which can be highly sensitive and proprietary. It was shown to be **impossible** to do such extraction for many types of neural networks in black-box *fault-free* scenario, and therefore [7] only focuses on the following three attacks.

- *Functionally Equivalent Extraction:* slightly weaker assumption is considered for functionally equivalent extraction, where the attacker is capable of constructing $\hat{\mathcal{O}}$ such that $\forall x \in \mathcal{X}, \hat{\mathcal{O}}(x) = \mathcal{O}(x)$. In such case, it is not necessary to match the two models exactly, only the output of both models has to be the same for all the elements from the domain $\mathcal{X}$ of the dataset $\mathcal{D}$.
- *Fidelity Extraction:* for a target distribution $\mathcal{D}_F$ over $\mathcal{X}$, and goal similarity function $S(p_1, p_2)$, fidelity extraction tries to construct $\hat{\mathcal{O}}$ that minimizes $\Pr_{x\sim\mathcal{D}_F}\left[S(\hat{\mathcal{O}}(x), \mathcal{O}(x))\right]$. The adversary normally wants to keep both the correct and incorrect classification between the two models. A functionally equivalent extraction achieves a fidelity of 1 on all distributions and all distance functions.
- *Task Accuracy Extraction:* for a true task distribution $\mathcal{D}_A$ over $\mathcal{X} \times \mathcal{Y}$, task accuracy extraction tries to construct an $\hat{\mathcal{O}}$ that maximizes $\Pr_{(x,y)\sim\mathcal{D}_A}\left[\arg\max(\hat{\mathcal{O}}(x)) = y\right]$. In this setting, the aim is to achieve the same or higher accuracy than the original model. Therefore, it is the easiest type of extraction attack to construct, as it does not care about the original model's mistakes.

III. Methods

To be able to reverse engineer a neural network with fault injection attack, we first need to know the erroneous behavior of its elementary components – neurons. To study this behavior, we first identify each part of a neuron that can be faulted.

A. Possibilities to Fault a Neuron

Figure 2 shows a typical neuron computation in a neural network. Inputs are multiplied with weights and then summed together, adding a bias. Resulting value is fed to the activation function, which produces the final output of the neuron. Below, we identify the points where a fault can be introduced (numbers correspond to those in Figure 2):

1. *Inputs:* there are two possibilities to fault the input – either at the output of a neuron from the previous layer or at the input of the multiplication of the current neuron. The first case affects the computation of all the neurons in the current layer, while the second case only affects the target neuron.

2.-3. *Weights, Product:* unlike faulting the input, weight or product change only affects the target neuron. As we explain later in this paper, attacks on these values can give the attacker knowledge of the weights.

4.-5. *Bias, Summation:* attacks on bias can slightly change the input to the activation function, while the attacks on summation can change this greatly. Therefore, the latter one can be considered as one of the means of misclassification by faults.

6. *Activation function:* Fault attacks on activation function were studied in [5] from instruction skipping perspective. If attacked with a sufficient precision, they can cause misclassification.

B. Experimental Setup

In this work, we consider different models which were pretrained using transfer learning [23] on ImageNet dataset, following deep-layer feature extractor approach. We use several models which are available in public libraries, such as Keras [26] and PyTorch [27], and for the experiments, the last fully connected layers are removed and substituted with single fully connected layer, and retrained. For the training data, the visual dataset for object recognition task, CIFAR-10 [28], is used. The CIFAR-10 dataset contains 50k training data, and 10k test data, each of which is a 32×32 pixels color image. First, the images are upscaled to be consistent with the dimension used in the pretrained model, followed by normalization. Next, we add a Dense layer with 10 neurons at the output, corresponding to 10 classes in the dataset. The activation function used for the output layer is softmax. Global Average Pooling or Flatten is used before the dense layer to reduce the number of neurons at the output of pretrained networks.

C. Adversary Model

We consider an adversary model, where the adversary aims at IP theft for overproduction and illegal cloning of ML proprietary models, running on edge/IoT devices. The proprietary ML models are carefully derived through transfer learning from popular and open ML models like AlexNet [29], VGG-16 [30], ResNet-50 [31], Inception V3 [32], etc. While the initial layers are publicly known, the adversary aims at recovering the parameters of the re-trained fully connected layers. To enable model recovery, adversary acquires few legal copies of the target. Being a legal user, the adversary can use the target devices with known data and inject faults into the device. Fault injection is followed by secret parameters recovery. This is a case of IP theft that allows adversary to overproduce/clone the ML model on huge number of devices without paying the legal licence fee.

D. SNIFF – Sign Bit Flip Fault

The attack model for our work is bit flip on the sign bit of the intermediate values. In particular, we consider attack on two intermediate values: SNIFF on the product of the weight and the input, and SNIFF on the bias value.

SNIFF attack on the product can be achieved in the real device by targeting either the input, the weight, or the final product value (targets 1, 2, and 3 in Figure 2). In Section IV, we use the bit flip fault on the weight to model this attack. In case of SNIFF attack on the bias value, the attacker has to target the bias itself (target 4 in Figure 2).

E. Finding the Correct Timing for Faults

Once the target step is identified, one needs to find precise timing locations corresponding to the sensitive computation. As already demonstrated in [4], it is possible to determine the timing by using side-channel information, coming either from the power consumption of the device or from electromagnetic emanation (EM).

It can be shown in the example of 4 fully connected layer with 50, 30, 20 and 50 neurons in each layer respectively from the input layer, on ARM Cortex-M3 microcontroller mounted on the Arduino Due. The electromagnetic emanation measured through a near field probe (RF-U 5-2 H-field probe from Langer) is shown in Fig. 3. In Fig 3 (a) each layer can be easily identified. Next, Fig 3 (b) shows a zoom on computation of the first neuron of the third layer. Given the (50, 30, 20, 50) architecture, 20 multiplications are expected followed by the activation function. Each multiplication can be easily identified in Fig 3 (b) and thus precisely targeted with faults.

The process of finding the correct timing can be automated by using pattern recognition techniques to locate the multiplication patterns within the neuron computations. Similarly, position on the chip which leaks the information in the form of an electromagnetic field can be automatically located. For example, [33] shows both processes.

IV. Recovery of Secret Parameters

In this section, we will explain the recovery of the weights and biases of the last layer of deep-layer feature extractor model, constructed by using transfer learning.

A. Attack Intuition

The intuition of the parameter recovery attack is as follows. As shown in Figure 1, the attack works on the last layer of the student network. The detail of this layer is illustrated in Figure 4. The attacker first executes the model computation on last layer input, denoted by $\boldsymbol{I} = (I_1, I_2, \ldots, I_n)$, without fault injection, and observes the outputs – classes and corresponding probabilities from the last softmax layer.

Then, she injects fault into the last layer by performing SNIFF on a single product of the weight and the input ($I_i \times w_{ij}$). Based on the original (non-faulty) output values and the faulty ones, she can recover the unknown weight w_{ij}. Similarly, by performing SNIFF on a single bias value ($b_{S,i}$), she can recover this value by comparing the faulty and original network outputs.

B. Formalization

In this section we formally describe the attack. Suppose there are k layers in the teacher neural net, and for an input $\boldsymbol{x}$, the output is given by $\mathcal{L}_k(\mathcal{L}_{k-1}(\ldots \mathcal{L}_1(\boldsymbol{x})))$, where $\mathcal{L}_i$ denotes the function at layer i, which takes the output of the previous layer and gives input for the next layer. For example, $k = 1$ and $\mathcal{L}(\boldsymbol{x}) = \text{sigmoid}(\boldsymbol{x}^T W + \boldsymbol{b})$ denotes a fully connected one layer network with weight matrix W, bias vector $\boldsymbol{b}$ and activation function sigmoid.

Let $\mathcal{O}_{\theta_{T,-1}}$ denote the part of the teacher neural network that was preserved by the student neural network, i.e.

$$\mathcal{O}_{\theta_{T,-1}}(\boldsymbol{x}) := \mathcal{L}_{k-1}(\ldots \mathcal{L}_1(\boldsymbol{x})).$$

Here $\theta_{T,-1}$ denotes the parameters of the first $k-1$ layers of the teacher neural network.

Let W_S and $\boldsymbol{b}_S$ denote the trained weight matrix and bias vector for the last layer of student neural network. Suppose the $(k-1)$th layer of teacher network has n neurons and the output layer of student network has m neurons. Then we have W_S is an $n \times m$ matrix and $\boldsymbol{b}_S$ is a vector of length m. For an input $\boldsymbol{x}$, the output of the student neural network is then given by

$$\mathcal{O}_\theta(\boldsymbol{x}) = \text{Softmax}(\mathcal{O}_{\theta_{T,-1}}(\boldsymbol{x})^T W_S + \boldsymbol{b}_S),$$

Let $\boldsymbol{y}(\boldsymbol{x}) := \mathcal{O}_{\theta_{T,-1}}(\boldsymbol{x})^T W_S + \boldsymbol{b}_S$, then we have for $i = 1, 2, \ldots, m$,

$$\mathcal{O}_{\theta,i}(\boldsymbol{x}) = \frac{\exp y_i(\boldsymbol{x})}{\sum_{j=1}^m \exp(y_j(\boldsymbol{x}))}.$$

By our assumption, the attacker knows the teacher neural network and she can also observe the Softmax output, in particular, she knows the number m and hence the dimensions of W_S and $\boldsymbol{b}_S$. Our goal of model extraction then consists of recovering θ, the parameters for the student neural network. Let $\theta_S := \{W_S, \boldsymbol{b}_S\}$, then $\theta = \theta_S \cup \theta_{T,-1}$. Note that $\theta_{T,-1}$ are the parameters from the teacher network, which are public information. Thus our goal is to recover θ_S, or equivalently, W_S and $\boldsymbol{b}_S$.

Definition 1. *An input $\boldsymbol{x}$ is called a non-vanishing input for i ($i = 1, 2, \ldots, n$) if $\mathcal{O}_{\theta_{T,-1},i}(\boldsymbol{x}) \neq 0$.*

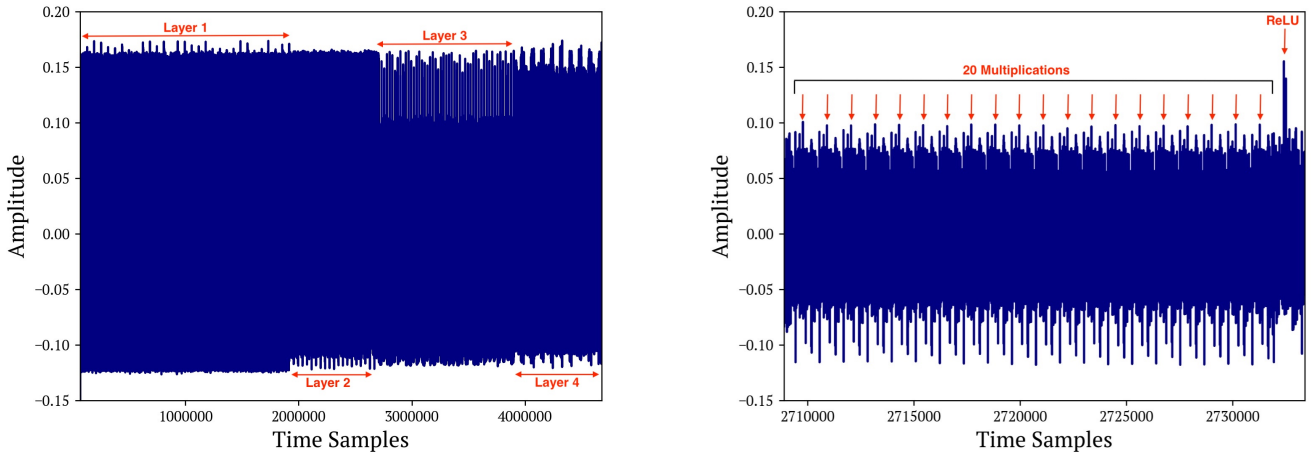


Fig. 3. Electromagnetic emanation measurement during the computation of 4 fully connected layers with 50, 30, 20, 50 neurons in each layer. In (a) each layer can be uniquely identified by the measurement trace, while (b) shows execution of one neuron in third layer showing timing of each of the 20 multiplications.

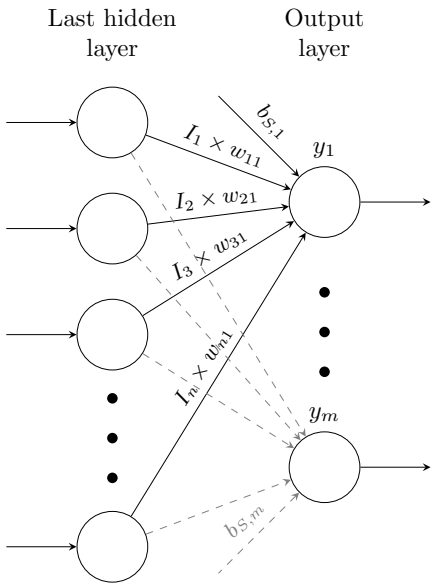


Fig. 4. Last two layers of the student model – nodes I_i are known, while the weights w_{ij} and biases $b_{S,j}$ are the target for the recovery.

For simplicity, let $\boldsymbol{I}(\boldsymbol{x})$ denote $\mathcal{O}_{\theta_{T,-1}}(\boldsymbol{x})$. As described in Section III-D, we consider SNIFF on the product $I_i w_{ij}$ and on the bias $b_{S,j}$.

We refer to the unknown weight w_{ij} as the *target weight parameter* and the unknown bias $b_{S,j}$ as the *target bias parameter*.

Theorem 1. *For any $j_0 \in \{1, 2, \ldots, m\}$ and any input $\boldsymbol{x}$. Suppose a SNIFF on target bias parameter b_{S,j_0} was carried out. Let z_{j_0} and $\tilde{z}_{j_0}$ denote the correct and faulted value of $\mathcal{O}_{\theta,j_0}(\boldsymbol{x})$. Then the target weight b_{S,j_0} can be recovered as:*

$$b_{S,j_0} = \frac{1}{2} \ln \left(\frac{\tilde{z}_{j_0}^{-1} - 1}{z_{j_0}^{-1} - 1} \right).$$

Proof. Let j_0 be given and let $\boldsymbol{x}$ be any input. For simplicity, we write $\boldsymbol{I}$ (resp. $\boldsymbol{y}$) instead of $\boldsymbol{I}(\boldsymbol{x})$ (resp. $\boldsymbol{y}(\boldsymbol{x})$). For any $j \in \{1, 2, \ldots, m\}$,

$$y_j = b_{S,j} + \sum_{i=1}^{n} I_i w_{ij}, \;\; z_j = \frac{\exp(y_j)}{\sum_{j'=1}^{m} \exp(y_{j'})}$$

In particular,

$$y_{j_0} = b_{S,j_0} + \sum_{i=1}^{n} I_i w_{ij_0}, \;\; z_{j_0} = \frac{\exp(y_{j_0})}{\sum_{j=1}^{m} \exp(y_j)}.$$

Let

$$\begin{aligned} A &:= \sum_{i=1}^{n} I_i w_{ij_0} = y_{j_0} - b_{S,j_0}, \\ B &:= \sum_{j=1, j \neq j_0}^{m} \exp(y_j). \end{aligned}$$

We have

$$\begin{aligned} z_{j_0} &= \frac{\exp(b_{S,j_0} + A)}{\exp(b_{S,j_0} + A) + B}, \\ \tilde{z}_{j_0} &= \frac{\exp(-b_{S,j_0} + A)}{\exp(-b_{S,j_0} + A) + B}. \end{aligned}$$

We note that by definition of Softmax, $z_{i_0} > 0$ and $\tilde{z}_{i_0} > 0$.

$$\begin{aligned} \frac{1}{z_{j_0}} - 1 &= \frac{\exp(b_{S,j_0}) \exp(A) + B}{\exp(b_{S,j_0}) \exp(A)} - 1 \\ &= \frac{B}{\exp(b_{S,j_0}) \exp(A)} = \exp(-b_{S,j_0}) \frac{B}{\exp(A)}. \end{aligned}$$

Similarly,

$$\begin{aligned} \frac{1}{\tilde{z}_{j_0}} - 1 &= \frac{\exp(-b_{S,j_0}) \exp(A) + B}{\exp(-b_{S,j_0}) \exp(A)} - 1 \\ &= \frac{B}{\exp(-b_{S,j_0}) \exp(A)} = \exp(b_{S,j_0}) \frac{B}{\exp(A)}. \end{aligned}$$

By definition of Softmax, $z_{j_0}^{-1} > 1$,

$$\frac{\tilde{z}_{j_0}^{-1} - 1}{z_{j_0}^{-1} - 1} = \exp(2b_{S,j_0}) \Longrightarrow b_{S,j_0} = \frac{1}{2} \ln \left(\frac{\tilde{z}_{j_0}^{-1} - 1}{z_{j_0}^{-1} - 1} \right).$$

□

Corollary 1. *The attacker can recover the bias vector $\boldsymbol{b}_S$ with m faults and $2m$ executions of the target neural network (the student neural network).*

Theorem 2. *For any $i_0 \in \{1, 2, \ldots, n\}, j_0 \in \{1, \ldots, m\}$ and any $\boldsymbol{x}$, a non-vanishing input for i_0. Suppose a SNIFF on target weight parameter $w_{i_0 j_0}$ was carried out. Let z_{j_0} and $\tilde{z}_{j_0}$ denote the correct and faulted value of $\mathcal{O}_{\theta,j_0}(\boldsymbol{x})$. Then the target weight $w_{i_0 j_0}$ can be recovered as:*

$$w_{i_0 j_0} = \frac{1}{2I_{i_0}} \ln \left(\frac{\tilde{z}_{j_0}^{-1} - 1}{z_{j_0}^{-1} - 1} \right).$$

Proof. Let i_0, j_0 be given, and let $\boldsymbol{x}$ be a non-vanishing input for i_0. For simplicity, we write $\boldsymbol{I}$ (resp. $\boldsymbol{y}$) instead of $\boldsymbol{I}(\boldsymbol{x})$ (resp. $\boldsymbol{y}(\boldsymbol{x})$). We let w_{ij} denote the (i,j)th entry of the weight matrix W_S. And let $b_{S,j}$ denote the jth entry of the bias vector $\boldsymbol{b}_S$. Then for any $j \in \{1, 2, \ldots, m\}$,

$$y_j = b_{S,j} + \sum_{i=1}^{n} I_i w_{ij}, \; z_j = \frac{\exp(y_j)}{\sum_{j'=1}^{m} \exp(y_{j'})}$$

In particular,

$$y_{j_0} = b_{S,j_0} + \sum_{i=1}^{n} I_i w_{ij_0}, \; z_{j_0} = \frac{\exp(y_{j_0})}{\sum_{j=1}^{m} \exp(y_j)}.$$

Let

$$\begin{aligned} A &:= b_{S,j_0} + \sum_{i=1, i \neq i_0}^{n} I_i w_{ij} = y_{j_0} - I_{i_0} w_{i_0 j_0}, \\ B &:= \sum_{j=1, j \neq j_0}^{m} \exp(y_j). \end{aligned}$$

We have

$$\begin{aligned} z_{j_0} &= \frac{\exp(I_{i_0} w_{i_0 j_0} + A)}{\exp(I_{i_0} w_{i_0 j_0} + A) + B}, \\ \tilde{z}_{j_0} &= \frac{\exp(-I_{i_0} w_{i_0 j_0} + A)}{\exp(-I_{i_0} w_{i_0 j_0} + A) + B}. \end{aligned}$$

We note that by definition of Softmax, $z_{j_0} > 0$ and $\tilde{z}_{j_0} > 0$.

$$\begin{aligned} \frac{1}{z_{j_0}} - 1 &= \frac{\exp(I_{i_0} w_{i_0 j_0}) \exp(A) + B}{\exp(I_{i_0} w_{i_0 j_0}) \exp(A)} - 1 \\ &= \frac{B}{\exp(I_{i_0} w_{i_0 j_0}) \exp(A)} = \exp(-I_{i_0} w_{i_0 j_0}) \frac{B}{\exp(A)}. \end{aligned}$$

Similarly,

$$\begin{aligned} \frac{1}{\tilde{z}_{j_0}} - 1 &= \frac{\exp(-I_{i_0} w_{i_0 j_0}) \exp(A) + B}{\exp(-I_{i_0} w_{i_0 j_0}) \exp(A)} - 1 \\ &= \frac{B}{\exp(-I_{i_0} w_{i_0 j_0}) \exp(A)} = \exp(I_{i_0} w_{i_0 j_0}) \frac{B}{\exp(A)}. \end{aligned}$$

Since $\boldsymbol{x}$ is a non-vanishing input for i_0, we have $I_{i_0} \neq 0$. Also by definition of Softmax, $z_{i_0}^{-1} > 1$. Together with the above equations,

$$\frac{\tilde{z}_{j_0}^{-1} - 1}{z_{j_0}^{-1} - 1} = \exp(2I_{i_0} w_{i_0 j_0}) \Longrightarrow w_{i_0 j_0} = \frac{1}{2I_{i_0}} \ln \left(\frac{\tilde{z}_{j_0}^{-1} - 1}{z_{j_0}^{-1} - 1} \right).$$

□

Thus the attacker can recover an i_0, j_0 entry of the weight matrix W_S, by first running an offline phase to find a non-vanishing input $\boldsymbol{x}$ for i_0, then with two executions of the student neural network - one without fault and one with fault.

Corollary 2. *The attacker can recover the weight matrix W_S with mn faults and $2mn$ executions of the targeted neural network (the student neural network).*

In practice, during the inference, Softmax might be omitted to save the computation time. We remark that in this case, Corollaries 1 and 2 still hold and the computations needed will be even easier. Keeping notations in Theorem 1 and the proof, we have

$$z_{j_0} = y_{j_0} = b_{S,j_0} + A, \tilde{z}_{j_0} = \tilde{y}_{j_0} = -b_{S,j_0} + A,$$

then, the target bias can be recovered using

$$b_{S,j_0} = \frac{1}{2}(z_{j_0} - \tilde{z}_{j_0}).$$

The target weight can be recovered in a similar manner.

V. Results and Discussion

Bit-flip attacks have been shown to be practical on embedded devices [34]. Similar results can be obtained by using a Rowhammer in DRAM memories [9]. In this part we first simulate the bit-flip attack in the code and then use the formulas from the previous section to reverse engineer the model parameters. Then, we compare our results to previous works. Finally, we discuss selection of the model extraction method based on the attack purpose.

A. Experimental Results

Experimental results for reverse engineering with bit-flips are stated in Table I. We targeted deep-layer feature extractor networks that were based on publicly available networks, being able to reverse engineer the weights in the last layer. When it comes to recovery of weights, the weight precision for all except 3 networks was 10^{-14}, for the remaining cases it was 10^{-13}. In case of bias recovery, the precision was always 10^{-14}.

We would like to highlight that the method from Section III allows the recovery of the exact weight value if we have arbitrary precision of floating point numbers. In practice, this depends on the used library, computer architecture, and settings. For our experiments we used Python with Keras library (version 2.3.1) for deep learning. This library uses numpy for floating point number representation, offering different precision ranging from 16 to 64 bits[1]. In our setting we set

[1]Numpy supports up to 128-bit floats, but those are not compatible with Keras.

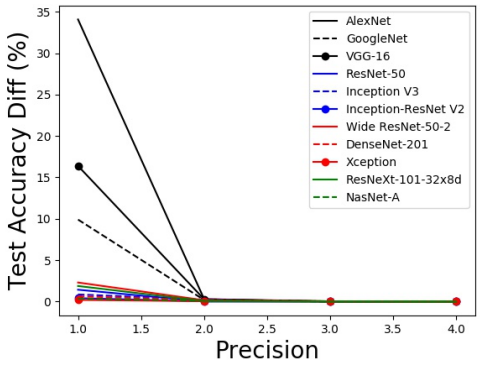


Fig. 5. Functionally equivalent model extraction: The difference in test accuracy between the actual model and recovered model against the parameter precision up to certain floating point digit. If the parameter values are the same up to the second decimal point, the test accuracy of the recovered model is the same as the original one for all the evaluated networks.

the `float64` to be the default representation to get the most precise results.

B. Comparison to Prior Work

The seminal work of Lowd and Meek [42] enabled full model functionally equivalent extraction for linear models. Further, full model functionally equivalent extraction for a 2-layer non-linear neural network was proposed by Milli et al [43] in a theoretical setting. When considering extraction of fully implemented neural networks, only two works have come to light. Batina et al. [4] relied on side-channel leakage on electromagnetic measurements to extract the functionally equivalent model in a known input setting. They reported an error on recovered weight of 2.5×10^{-3}, and full network recovery. Later, Jagielski et al. [7] proposed two attacks. One of the two attacks enabled full model functionally equivalent extraction for a 2-layer neural network with a weight error of only 9×10^{-7}, which is current state-of-the-art. This method required access to logit values, which is a stronger assumption compared to outputs of the softmax function used in our approach. The other method they developed enabled full model extraction preserving task accuracy and fidelity.

Compared to these prior works, the goal of our work is exact extraction. When experimentally testing our method with Keras and Pytorch, the recovered weight error of our fault assisted approach was at most 10^{-13}. *It must be noted that the stated error is the precision error of the Python libraries used in our experiments. Otherwise, our proposed method can provably recover the exact weights.* The comparison is summarized in Table II.

C. Selecting the Model Extraction Method

It is important to understand the purpose of the model extraction attack – after that, it is possible to determine what type of attack should the attacker choose, ultimately deciding the difficulty of the extraction.

If the main goal is to have a task accurate extraction or functionally equivalent extraction, the attacker can achieve this by querying the network with a set of inputs and observing the outputs [7], [43]. In this case, the extracted network might have a different architecture than the original one, but will perform well on the same or similar task. As can be seen in Figure 5, for functionally equivalent extraction, it is enough to be able to recover the parameters with the precision of two floating point digits for all the considered networks. However, if the task changes, the extracted network might give different output than the original one, as it was not trained the same way. For example, some attackers might be interested in robustness of a certain network to a set adversarial examples, but are not able to query the original network with the entire set. In such case, task accurate extraction will not help as it will not reveal the vulnerability of the original network by testing the extracted network. As the adversarial examples are often very close to decision boundaries [24], precision of the parameters is crucial to assess the vulnerability. For such scenarios, it is necessary to have extracted network that is as close to the original network as possible. That is a task of exact extraction.

VI. Protection Techniques

In this section we will outline different techniques that can help protect neural network implementations against fault injection attacks.

A. Overview

In general, the protection techniques against fault injection can work either on device level, or implementation level.

Device level techniques focus on preventing the attacker to reach the chip, by various forms of packaging, light sensors, etc. [44]. The goal is to increase the equipment and expertise requirement to access the chip in a way that the possible reward for the attacker for doing so will be lower than the effort she has to put in. Device level techniques can also have a different working principle – to detect potential tampering with the chip. In this case, a hardware sensor that checks environmental conditions can be deployed [45], [46], [47].

Implementation level techniques aim at detecting changes in the intermediate data. Detection can be achieved by using various encoding techniques, ranging from simple ones such as parity [48], to sophisticated codes that can be customized to protect against specific fault models [49]. Another approach is performing the computation several times and comparing the result. A different way to use redundancy is to perform it at the instruction level, either by generating instruction sequences that replace the original vulnerable instructions [50], or by re-arranging the data within the instructions to make it hard to tamper with without detection [51]. However, there is no straightforward way of using these two techniques for protecting DNNs. It is important to mention that unlike device level techniques, the implementation level countermeasures normally incur significant overheads, either in time, circuit area, or power consumption.

Protecting the learning phase. Additionally, there is a line of work that focuses on protecting the learning phase of the deep learning method [52]. Such protection technique might be useful in case the learning does not happen in a protected

TABLE I
EXPERIMENTAL RESULTS FOR REVERSE ENGINEERING WITH FAULTS. WE TARGETED DEEP-LAYER FEATURE EXTRACTOR NETWORKS BASED ON PUBLICLY AVAILABLE NETWORKS FOR IMAGE CLASSIFICATION.

		Reverse Engineering	
Model	No. of Features To Recover	Weight Precision	Bias Precision
AlexNet [29]	9216	10^{-13}	10^{-14}
GoogleNet (Inception V1) [35]	1024	10^{-14}	10^{-14}
VGG-16 [30]	25088	10^{-13}	10^{-14}
ResNet-50 [31]	2048	10^{-14}	10^{-14}
Inception V3 [32]	2048	10^{-13}	10^{-14}
Inception ResNet V2 [36]	1536	10^{-14}	10^{-14}
Wide-ResNet-50-2 [37]	2048	10^{-14}	10^{-14}
DenseNet-201 [38]	1920	10^{-14}	10^{-14}
Xception [39]	2048	10^{-14}	10^{-14}
ResNeXt-101 32x8d [40]	2048	10^{-14}	10^{-14}
NasNet-A (6 @ 4032) [41]	4032	10^{-14}	10^{-14}

TABLE II
COMPARISON WITH PRIOR WORK TARGETING DIRECT MODEL EXTRACTION. * DENOTES THAT TECHNIQUE HAS NULL PRECISION ERROR. IN OUR EXPERIMENTS THE ERROR REPORTED WAS AT MOST 10^{-13}, WHICH IS THE PRECISION LIMITATION OF THE USED PYTHON LIBRARIES.

Attack	Leakage Source	Weight Error	Target Network	Goal
[42]	Labels	N/A	Linear models	Functionally equivalent
[43]	Gradients/logits	N/A	2-layer neural network	Functionally equivalent
[4]	EM Side-Channel	2.5×10^{-3}	Full network	Functionally equivalent
[7]	Probabilities/logits	9×10^{-7}	2-layer neural network	Functionally equivalent
This Work	Faults/Probabilities	$0\ (10^{-13})^*$	2-layer neural network	Exact extraction

environment and there is a significant risk of faults coming either from the environment or from the attacker. In our work we consider the model is already learned and therefore, the attacker is trying to tamper with the classification phase.

B. Analysis

Analysis of overheads and coverage of each countermeasure that can be used against instruction skips presented in earlier sections is stated in Table III. Here, we provide more details on each technique and its applicability to DNN.

Spatial/temporal redundancy. This is the most straightforward way to protect a circuit. Implementer can choose the number of redundant executions depending on what attacker model is expected. In case of redundancy, there is always an integrity check or a majority voting that decides whether the output is valid or not. When used as a countermeasure in cryptography, circuit is either deployed 2-3$\times$ on the chip (spatial redundancy), or the computation is repeated 2-3$\times$ one after another (temporal redundancy) [53]. Execution times can be randomized so that it is hard to reproduce the same fault in all the redundant executions.

Software encoding. As the software encoding countermeasures are realized by table look-up operations, they are not directly applicable to neural networks which operate on real values. However, it is possible to apply this countermeasure for fixed-point arithmetic networks [55]. As it was shown, fixed-point arithmetic can provide good results when used on bigger networks [56]. The timing overhead in this case is around 75% – for example, let us consider a multiplication operation on AVR architecture: for the unprotected implementation, there is operand loading into the registers (2×1 clk cycle), followed by a multiplication (2 clk cycles), resulting into 4 clock cycles. For the protected implementation, there is a register precharge (see e.g. Section 5.1 of [49]) of both input registers and the output register (3×1 clk cycle), followed by the operand loading (2×1 clk cycle) and table look-up (2 clk cycles), resulting into 7 clock cycles. Regarding the area overhead, as stated in [49], in case the codeword size is ≤ 8 bits, there is a fixed table size of 65 kB per binary operation (e.g. multiplication). That is why the area (memory) overhead is huge for this case.

Hardware sensor. Application of a hardware sensor to protect DNN circuit is depicted in Figure 6. The main advantage of hardware sensor is that there is no need to change the underlying implementation of the neural network. The sensor resides on the front side of the chip, protecting all the underlying circuits from fault injection. In case there is a sudden parasitic voltage detected by such sensor, it raises an alarm. While front side deployment might be vulnerable to back (substrate) side injection, [45] reported successful detection of backside injection. Recently, circuit level techniques were also proposed to enhance backside detection capabilities [57]. Afterwards, security measures, such as discarding the output, can be applied. Recently, a way to automate the deployment of such circuit was proposed [58].

To summarize, selection of countermeasures depends heavily on the type of application that relies on DNN outputs. For security critical application, it would be recommended to combine several techniques together to minimize the possible attack vectors and make cost of the attack as high as possible.

VII. CONCLUSION

In this paper, we developed a method for provable exact extraction of neural network parameters with the help of fault

TABLE III
OVERVIEW OF COUNTERMEASURES EFFECTIVE AGAINST SKIPPING INSTRUCTIONS.

Countermeasure	Overhead Time	Overhead Area	Coverage
Spatial redundancy ($\times N$)	–	$N \times 100\%$	Covers up to $N-1$ faults. To break the countermeasure, faults need to be injected at the same instruction in all the redundant circuits – which normally requires multiple fault injection devices.
Temporal redundancy ($\times N$)	$N \times 100\%$	–	Covers up to $N-1$ faults. To break the countermeasure, faults need to be injected at the same instruction in all the redundant executions.
Software encoding [49]	75%	$\approx 65,000\%$	Protects against instruction skips that target one instruction at a time. Although it does not protect against consecutive instruction skips, during one execution it can protect arbitrary number of non-consecutive skips with 100% detection rate.
Hardware sensor [54]	–	$1.1\%^2$	As the sensor is based on detecting voltage variations on the chip surface, the detection rate depends on the fault injection device parameters. The most recent work shows high detection rates for both laser and EM fault injection techniques, 97% and 100% detected injections, respectively.

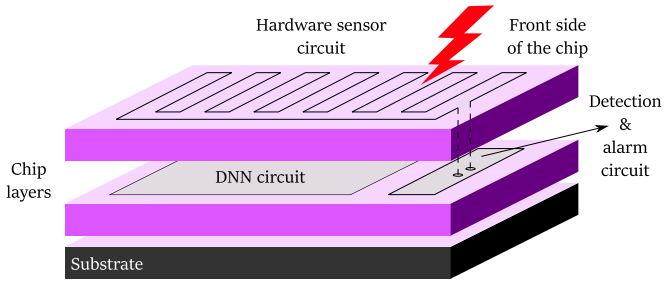


Fig. 6. Hardware sensor protecting the DNN circuit.

injection. Our method aims at recovering the student layer of deep-layer feature extractor networks that were constructed by transfer learning. This is done by changing the sign of intermediate values to obtain the information about the parameters with a method called SNIFF – sign bit flip fault. Our practical experiments show that the exact recovery ultimately depends on computer architecture and the precision of the library used. For 64-bit floats used in Keras, the parameter recovery error was at most 10^{-13}.

For the future work, it would be interesting to look at methods that would allow extraction of parameters from deeper layers of a network. It would be also worth exploring whether a combination of multiple faults during a single execution can improve the efficiency of the attack.

REFERENCES

[1] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *USENIX Security 16*, 2016, pp. 601–618.

[2] B. Wang and N. Z. Gong, "Stealing hyperparameters in machine learning," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 36–52.

[3] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Annual international cryptology conference*. Springer, 1997, pp. 513–525.

[4] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel," in *USENIX Security 19*, 2019, pp. 515–532.

[5] J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Practical fault attack on deep neural networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2204–2206.

[6] F. F. dos Santos, P. F. Pimenta, C. B. Lunardi, L. Draghetti, L. Carro, D. R. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Trans. Reliab.*, vol. 68, no. 2, pp. 663–677, 2019. [Online]. Available: https://doi.org/10.1109/TR.2018.2878387

[7] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot, "High accuracy and high fidelity extraction of neural networks," in *USENIX Security 20*, 2020, pp. 1345–1362.

[8] S. Ordas, L. Guillaume-Sage, K. Tobich, J.-M. Dutertre, and P. Maurine, "Evidence of a larger em-induced fault model," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2014, pp. 245–259.

[9] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

[10] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0ltpwn: Attacking x86 processor integrity from software," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[11] S. Anceau, P. Bleuet, J. Clédière, L. Maingault, J.-l. Rainard, and R. Tucoulou, "Nanofocused x-ray beam to reprogram secure circuits," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 175–188.

[12] J. Breier and W. He, "Multiple fault attack on present with a hardware trojan implementation in fpga," in *2015 international workshop on secure internet of things (SIot)*. IEEE, 2015, pp. 58–64.

[13] Y. Liu, L. Wei, B. Luo, and Q. Xu, "Fault injection attack on deep neural network," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 131–138.

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

[15] X. Hou, J. Breier, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Security evaluation of deep neural network resistance against laser fault injection," in *2020 IEEE International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA)*. IEEE, 2020, pp. 1–6.

[16] ——, "Physical security of deep learning on edge devices: Comprehensive evaluation of fault injection attack vectors," *Microelectronics Reliability*, vol. 120, p. 114116, 2021.

[17] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraş, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," *arXiv:1906.01017*, 2019.

[18] F. Yao, A. S. Rakin, and D. Fan, "Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1463–1480.

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

[20] ——, "Tbt: Targeted neural network attack with bit trojan," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 13 198–13 207.

[21] J. Bai, B. Wu, Y. Zhang, Y. Li, Z. Li, and S.-T. Xia, "Targeted attack against deep neural networks via flipping limited weight bits," *arXiv preprint arXiv:2102.10496*, 2021.

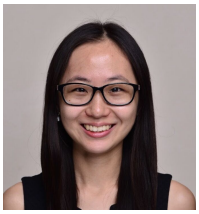
[22] C. Torres-Huitzil and B. Girau, "Fault and error tolerance in neural networks: A review," *IEEE Access*, vol. 5, pp. 17 322–17 341, 2017.

[23] B. Wang, Y. Yao, B. Viswanath, H. Zheng, and B. Y. Zhao, "With great training comes great vulnerability: Practical attacks against transfer learning," in *USENIX Security 18*, 2018, pp. 1281–1297.
[24] A. Shamir, I. Safran, E. Ronen, and O. Dunkelman, "A simple explanation for the existence of adversarial examples with small hamming distance," *arXiv:1901.10861*, 2019.
[25] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 3–18.
[26] F. Chollet *et al.*, "Keras," 2015.
[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
[28] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.
[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, 2014.
[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 2016, pp. 770–778. [Online]. Available: https://doi.org/10.1109/CVPR.2016.90
[32] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, USA, 2016*. IEEE Computer Society, 2016, pp. 2818–2826. [Online]. Available: https://doi.org/10.1109/CVPR.2016.308
[33] J. Danial, D. Das, S. Ghosh, A. Raychowdhury, and S. Sen, "Scniffer: Low-cost, automated, efficient electromagnetic side-channel sniffing," 2020.
[34] M. Agoyan, J.-M. Dutertre, A.-P. Mirbaha, D. Naccache, A.-L. Ribotta, and A. Tria, "How to flip a bit?" in *2010 IEEE 16th International On-Line Testing Symposium*. IEEE, 2010, pp. 235–239.
[35] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR*, 2015, pp. 1–9.
[36] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, S. P. Singh and S. Markovitch, Eds. AAAI Press, 2017, pp. 4278–4284. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14806
[37] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *BMVC*, 2016.
[38] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, 2017, pp. 2261–2269. [Online]. Available: https://doi.org/10.1109/CVPR.2017.243
[39] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 1800–1807. [Online]. Available: https://doi.org/10.1109/CVPR.2017.195
[40] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," *arXiv:1611.05431*, 2016.
[41] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 2018, pp. 8697–8710. [Online]. Available: https://ieeexplore.ieee.org/xpl/conhome/8576498/proceeding
[42] D. Lowd and C. Meek, "Adversarial learning," in *ACM SIGKDD*. ACM, 2005, pp. 641–647.
[43] S. Milli, L. Schmidt, A. D. Dragan, and M. Hardt, "Model reconstruction from model explanations," in *Proceedings of the Conference on Fairness, Accountability, and Transparency*. ACM, 2019, pp. 1–9.
[44] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.
[45] W. He, J. Breier, S. Bhasin, N. Miura, and M. Nagata, "Ring oscillator under laser: Potential of pll-based countermeasure against laser fault injection," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2016 Workshop on*. IEEE, 2016, pp. 102–113.
[46] L. Zussa, A. Dehbaoui, K. Tobich, J.-M. Dutertre, P. Maurine, L. Guillaume-Sage, J. Clediere, and A. Tria, "Efficiency of a glitch detector against electromagnetic fault injection," in *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 203.
[47] P. Ravi, S. Bhasin, J. Breier, and A. Chattopadhyay, "Ppap and ippap: Pll-based protection against physical attacks," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2018, pp. 620–625.
[48] R. Karri, G. Kuznetsov, and M. Goessel, "Parity-based concurrent error detection of substitution-permutation network block ciphers," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2003, pp. 113–124.
[49] J. Breier, X. Hou, and Y. Liu, "On evaluating fault resilient encoding schemes in software," *IEEE Transactions on Dependable and Secure Computing*, 2019.
[50] S. Patranabis, A. Chakraborty, and D. Mukhopadhyay, "Fault tolerant infective countermeasure for aes," *Journal of Hardware and Systems Security*, vol. 1, no. 1, pp. 3–17, 2017.
[51] C. Patrick, B. Yuce, N. F. Ghalaty, and P. Schaumont, "Lightweight fault attack resistance in software using intra-instruction redundancy," in *International Conference on Selected Areas in Cryptography*. Springer, 2016, pp. 231–244.
[52] Y. Taniguchi, N. Kamiura, Y. Hata, and N. Matsui, "Activation function manipulation for fault tolerant feedforward neural networks," in *Proceedings Eighth Asian Test Symposium (ATS'99)*. IEEE, 1999, pp. 203–208.
[53] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures against fault attacks on software implemented aes: effectiveness and cost," in *Proceedings of the 5th Workshop on Embedded Systems Security*. ACM, 2010, p. 7.
[54] M. Khairallah, J. Breier, S. Bhasin, and A. Chattopadhyay, "Differential fault attack resistant hardware design automation," in *Automated Methods in Cryptographic Fault Analysis*. Springer, 2019, pp. 209–219.
[55] K. Hwang and W. Sung, "Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1," in *2014 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2014, pp. 1–6.
[56] W. Sung, S. Shin, and K. Hwang, "Resiliency of deep neural networks under quantization," *arXiv preprint arXiv:1511.06488*, 2015.
[57] K. Matsuda, T. Fujii, N. Shoji, T. Sugawara, K. Sakiyama, Y.-i. Hayashi, M. Nagata, and N. Miura, "A 286 f 2/cell distributed bulk-current sensor and secure flush code eraser against laser fault injection attack on cryptographic processor," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 11, pp. 3174–3182, 2018.
[58] J. Breier, X. Hou, and S. Bhasin, Eds., *Automated Methods in Cryptographic Fault Analysis*, 1st ed. Springer, Mar 2019.

Jakub Breier is currently a Senior Researcher in Embedded Security at Silicon Austria Labs, Graz, Austria. Before that, he worked at Nanyang Technological University, Singapore on hardware security and at Underwriters Laboratories, Singapore on security evaluation of embedded devices. He received his PhD in Applied Informatics from Slovak University of Technology (STU), Slovakia in 2013, Master's in Information Technology Security from Masaryk University, Czech Republic in 2010, and Bachelor's in Informatics from STU, Slovakia in 2008. His research topics include fault and side-channel analysis methods and countermeasures, advanced fault injection techniques, and deep learning security.

[2]Sensor requires power during the operation, therefore there is a power overhead of $\approx 5.3\%$ per 16-bit multiplier.

Dirmanto Jap is currently a Research Scientist at PACE Lab, Temasek Laboratories, Nanyang Technological University (NTU), Singapore. He previously received his Ph.D in Mathematics from NTU in 2016. His main research topics include physical attacks (side-channel and fault attacks) and countermeasures, practical laser/EM fault injection, application of machine learning and deep learning for side-channel attacks and hardware Trojan detection, as well as security of deep learning.



Xiaolu Hou is currently an Assistant Professor at Faculty of Informatics and Information Technologies, Slovak University of Technology, Slovakia. She received her Ph.D. degree in mathematics from Nanyang Technological University (NTU), Singapore, in 2017. Her research focus is on fault injection and side-channel attacks. She also has research experience in security of neural networks, location privacy, multiparty computation and differential privacy. With a wide range of research interests, she has published her work at top venues within various fields, ranging from mathematics to computer security.

Shivam Bhasin is a Senior Research Scientist and Programme Manager (Cryptographic Engineering) at Centre for Hardware Assurance, Temasek Laboratories, Nanyang Technological University Singapore. He received his PhD in Electronics & Communication from Telecom Paristech in 2011, Advanced Master (Mastère Spécialisé) in security of integrated systems & applications from Mines Saint-Etienne, France in 2008. Before NTU, Shivam held position of Research Engineer in Institut Mines-Telecom, France. He was also a visiting researcher at UCL, Belgium (2011) and Kobe University (2013). His research interests include embedded security, trusted computing and secure designs. He has co-authored several publications at recognized journals and conferences. Some of his research now also forms a part of ISO/IEC 17825 standard.

Yang Liu graduated in 2005 with a Bachelor of Computing (Honours) in the National University of Singapore (NUS). In 2010, he obtained his PhD and started his post doctoral work in NUS, MIT and SUTD. In 2012 fall, he joined Nanyang Technological University (NTU) as a Nanyang Assistant Professor. He is currently an associate professor and Director of the cybersecurity lab in NTU. Dr. Liu specializes in software verification, security and software engineering. His research has bridged the gap between the theory and practical usage of formal methods and program analysis to evaluate the design and implementation of software for high assurance and security. By now, he has more than 200 publications in top tier conferences and journals. He has received a number of prestigious awards including MSRA Fellowship, TRF Fellowship, Nanyang Assistant Professor, Tan Chin Tuan Fellowship, and 8 best paper awards in top conferences like ASE, FSE and ICSE. He is leading a large research team working on the state-of-the-art software engineering and cybersecurity problems.

APPENDIX

A. Other Activation Functions

In this section we consider the case when the activation function of the output layer is not softmax. Following notations from Section IV-B, suppose there are k layers in the teacher neural net, and let $\mathcal{O}_{\theta_{T,-1}}$ denote the part of the teacher neural network that was preserved by the student neural network. Let W_S and $\boldsymbol{b}_S$ denote the trained weight matrix and bias vector for the last layer of student neural network. Suppose the $(k-1)$th layer of teacher network has n neurons and the output layer of student network has m neurons.

Our attack goal is to recover W_S and $\boldsymbol{b}_S$. We refer to the unknown weight w_{ij} as the *target weight parameter* and the unknown bias $b_{S,j}$ as the *target bias parameter*.

For an input $\boldsymbol{x}$, let $\boldsymbol{I}(\boldsymbol{x})$ denote $\mathcal{O}_{\theta_{T,-1}}(\boldsymbol{x})$. Let $\boldsymbol{y}(\boldsymbol{x}) := \mathcal{O}_{\theta_{T,-1}}(\boldsymbol{x})^T W_S + \boldsymbol{b}_S$. For simplicity, we write $\boldsymbol{I}$ (resp. $\boldsymbol{y}$) instead of $\boldsymbol{I}(\boldsymbol{x})$ (resp. $\boldsymbol{y}(\boldsymbol{x})$). As described in Section III-D, we consider SNIFF on the product $I_i w_{ij}$ and on the bias $b_{S,j}$.

1) Sigmoid: In case the activation function for the last layer is sigmoid, for an input $\boldsymbol{x}$, the output of the student neural network is given by

$$\mathcal{O}_\theta(\boldsymbol{x}) = \text{sigmoid}(\mathcal{O}_{\theta_{T,-1}}(\boldsymbol{x})^T W_S + \boldsymbol{b}_S),$$

for $i = 1, 2, \ldots, m$,

$$\mathcal{O}_{\theta,i}(\boldsymbol{x}) = \frac{1}{1+\exp(-y_i(\boldsymbol{x}))}.$$

Theorem 3. *For any $j_0 \in \{1, 2, \ldots, m\}$ and any input $\boldsymbol{x}$. Suppose a SNIFF on target bias parameter b_{S,j_0} was carried out. Let z_{j_0} and $\tilde{z}_{j_0}$ denote the correct and faulted value of $\mathcal{O}_{\theta,j_0}(\boldsymbol{x})$. Then the target weight b_{S,j_0} can be recovered as:*

$$b_{S,j_0} = \frac{1}{2}\ln\left(\frac{\tilde{z}_{j_0}^{-1}-1}{z_{j_0}^{-1}-1}\right).$$

Proof. Let j_0 be given and let $\boldsymbol{x}$ be any input. For any $j \in \{1, 2, \ldots, m\}$,

$$y_j = b_{S,j} + \sum_{i=1}^{n} I_i w_{ij},\ z_j = \frac{1}{1+\exp(-y_j)}.$$

In particular,

$$y_{j_0} = b_{S,j_0} + \sum_{i=1}^{n} I_i w_{ij_0},\ z_{j_0} = \frac{1}{1+\exp(-y_{j_0})}.$$

Let

$$A \ := \ \sum_{i=1}^{n} I_i w_{ij_0} = y_{j_0} - b_{S,j_0}.$$

We have

$$\begin{aligned} z_{j_0} &= \frac{1}{1+\exp(-A-b_{S,j_0})}, \\ \tilde{z}_{j_0} &= \frac{1}{1+\exp(-A+b_{S,j_0})}. \end{aligned}$$

We note that by definition of sigmoid, $z_{i_0} > 0$ and $\tilde{z}_{i_0} > 0$.

$$\begin{aligned} \frac{1}{z_{j_0}} - 1 &= \exp(-A-b_{S,j_0}) \\ \frac{1}{\tilde{z}_{j_0}} - 1 &= \exp(-A+b_{S,j_0}) \end{aligned}$$

By definition of sigmoid, $z_{j_0}^{-1} > 1$,

$$\frac{\tilde{z}_{j_0}^{-1} - 1}{z_{j_0}^{-1} - 1} = \exp(2b_{S,j_0}) \Longrightarrow b_{S,j_0} = \frac{1}{2} \ln \left(\frac{\tilde{z}_{j_0}^{-1} - 1}{z_{j_0}^{-1} - 1} \right).$$

□

Corollary 3. *The attacker can recover the bias vector $\boldsymbol{b}_S$ with m faults and $2m$ executions of the target neural network (the student neural network).*

Theorem 4. *For any $i_0 \in \{1, 2, \ldots, n\}, j_0 \in \{1, \ldots, m\}$ and any $\boldsymbol{x}$, a non-vanishing input for i_0. Suppose a SNIFF on target weight parameter $w_{i_0 j_0}$ was carried out. Let z_{j_0} and $\tilde{z}_{j_0}$ denote the correct and faulted value of $\mathcal{O}_{\theta,j_0}(\boldsymbol{x})$. Then the target weight $w_{i_0 j_0}$ can be recovered as:*

$$w_{i_0 j_0} = \frac{1}{2I_{i_0}} \ln \left(\frac{\tilde{z}_{j_0}^{-1} - 1}{z_{j_0}^{-1} - 1} \right).$$

Proof. Let i_0, j_0 be given, and let $\boldsymbol{x}$ be a non-vanishing input for i_0. We let w_{ij} denote the (i,j)th entry of the weight matrix W_S. And let $b_{S,j}$ denote the jth entry of the bias vector $\boldsymbol{b}_S$. Then for any $j \in \{1, 2, \ldots, m\}$,

$$y_j = b_{S,j} + \sum_{i=1}^{n} I_i w_{ij}, \; z_j = \frac{1}{1 + \exp(-y_j)}.$$

In particular,

$$y_{j_0} = b_{S,j_0} + \sum_{i=1}^{n} I_i w_{ij_0}, \; z_{j_0} = \frac{1}{1 + \exp(-y_{j_0})}.$$

Let

$$A \;:=\; b_{S,j_0} + \sum_{i=1, i \neq i_0}^{n} I_i w_{ij_0} = y_{j_0} - I_{i_0} w_{i_0 j_0}.$$

We have

$$\begin{aligned} z_{j_0} &= \frac{1}{1 + \exp(-A - I_{i_0} w_{i_0 j_0})}, \\ \tilde{z}_{j_0} &= \frac{1}{1 + \exp(-A + I_{i_0} w_{i_0 j_0})}. \end{aligned}$$

We note that by definition of sigmoid, $z_{i_0} > 0$ and $\tilde{z}_{i_0} > 0$.

$$\begin{aligned} \frac{1}{z_{j_0}} - 1 &= \exp(-A - I_{i_0} w_{i_0 j_0}) \\ \frac{1}{\tilde{z}_{j_0}} - 1 &= \exp(-A + I_{i_0} w_{i_0 j_0}) \end{aligned}$$

Since $\boldsymbol{x}$ is a non-vanishing input for i_0, we have $I_{i_0} \neq 0$. Also by definition of sigmoid, $z_{i_0}^{-1} > 1$. Together with the above equations,

$$\frac{\tilde{z}_{j_0}^{-1} - 1}{z_{j_0}^{-1} - 1} = \exp(2I_{i_0} w_{i_0 j_0}) \Longrightarrow w_{i_0 j_0} = \frac{1}{2I_{i_0}} \ln \left(\frac{\tilde{z}_{j_0}^{-1} - 1}{z_{j_0}^{-1} - 1} \right).$$

□

Corollary 4. *The attacker can recover the weight matrix W_S with mn faults and $2mn$ executions of the targeted neural network (the student neural network).*

2) Tanh: In case the activation function for the last layer is tanh, for an input $\boldsymbol{x}$, the output of the student neural network is given by

$$\mathcal{O}_\theta(\boldsymbol{x}) = \tanh(\mathcal{O}_{\theta_{T,-1}}(\boldsymbol{x})^T W_S + \boldsymbol{b}_S),$$

for $i = 1, 2, \ldots, m$,

$$\mathcal{O}_{\theta,i}(\boldsymbol{x}) = \frac{\exp(y_i(\boldsymbol{x})) - \exp(-y_i(\boldsymbol{x}))}{\exp(y_i(\boldsymbol{x})) + \exp(-y_i(\boldsymbol{x}))}.$$

Theorem 5. *For any $j_0 \in \{1, 2, \ldots, m\}$ and any input $\boldsymbol{x}$. Suppose a SNIFF on target bias parameter b_{S,j_0} was carried out. Let z_{j_0} and $\tilde{z}_{j_0}$ denote the correct and faulted value of $\mathcal{O}_{\theta,j_0}(\boldsymbol{x})$. Then the target weight b_{S,j_0} can be recovered as:*

$$b_{S,j_0} = \frac{1}{4} \ln \frac{(1 + z_{j_0})(1 - \tilde{z}_{j_0})}{(1 - z_{j_0})(1 + \tilde{z}_{j_0})}.$$

Proof. Let j_0 be given and let $\boldsymbol{x}$ be any input. For any $j \in \{1, 2, \ldots, m\}$,

$$\begin{aligned} y_j &= b_{S,j} + \sum_{i=1}^{n} I_i w_{ij}, \\ z_j &= \frac{\exp(y_j) - \exp(-y_j)}{\exp(y_j) + \exp(-y_j)} = 1 - \frac{2}{\exp(2y_j) + 1}. \end{aligned}$$

In particular,

$$y_{j_0} = b_{S,j_0} + \sum_{i=1}^{n} I_i w_{ij_0}, \; z_{j_0} = 1 - \frac{2}{\exp(2y_{j_0}) + 1}.$$

Let

$$A \;:=\; \sum_{i=1}^{n} I_i w_{ij_0} = y_{j_0} - b_{S,j_0}.$$

We have

$$\begin{aligned} z_{j_0} &= 1 - \frac{2}{\exp(2A + 2b_{S,j_0}) + 1}, \\ \tilde{z}_{j_0} &= 1 - \frac{2}{\exp(2A - 2b_{S,j_0}) + 1}. \end{aligned}$$

We note that by definition of tanh, $z_{i_0} < 1$ and $\tilde{z}_{i_0} < 1$.

$$\begin{aligned} \frac{1 + z_{j_0}}{1 - z_{j_0}} &= \exp(2A + 2b_{S,j_0}) \\ \frac{1 + \tilde{z}_{j_0}}{1 - \tilde{z}_{j_0}} &= \exp(2A - 2b_{S,j_0}), \end{aligned}$$

which gives

$$\frac{(1 + z_{j_0})(1 - \tilde{z}_{j_0})}{(1 - z_{j_0})(1 + \tilde{z}_{j_0})} = \exp(4b_{S,j_0}) \Longrightarrow b_{S,j_0} = \frac{1}{4} \ln \frac{(1 + z_{j_0})(1 - \tilde{z}_{j_0})}{(1 - z_{j_0})(1 + \tilde{z}_{j_0})}.$$

□

Corollary 5. *The attacker can recover the bias vector $\boldsymbol{b}_S$ with m faults and $2m$ executions of the target neural network (the student neural network).*

Theorem 6. *For any $i_0 \in \{1, 2, \ldots, n\}, j_0 \in \{1, \ldots, m\}$ and any $\boldsymbol{x}$, a non-vanishing input for i_0. Suppose a SNIFF on target weight parameter $w_{i_0 j_0}$ was carried out. Let z_{j_0} and*

$\tilde{z}_{j_0}$ denote the correct and faulted value of $\mathcal{O}_{\theta,j_0}(\boldsymbol{x})$. Then the target weight $w_{i_0j_0}$ can be recovered as:

$$w_{i_0j_0} = \frac{1}{4I_{i_0}} \ln \frac{(1+z_{j_0})(1-\tilde{z}_{j_0})}{(1-z_{j_0})(1+\tilde{z}_{j_0})}.$$

Proof. Let j_0 be given and let $\boldsymbol{x}$ be any input. For any $j \in \{1, 2, \ldots, m\}$,

$$y_j = b_{S,j} + \sum_{i=1}^{n} I_i w_{ij}, \; z_j = 1 - \frac{2}{\exp(2y_j)+1}.$$

In particular,

$$y_{j_0} = b_{S,j_0} + \sum_{i=1}^{n} I_i w_{ij_0}, \; z_{j_0} = 1 - \frac{2}{\exp(2y_{j_0})+1}.$$

Let

$$A \;:=\; b_{S,j_0} + \sum_{i=1, i\neq i_0}^{n} I_i w_{ij_0} = y_{j_0} - I_{i_0} w_{i_0j_0}.$$

We have

$$\begin{aligned} z_{j_0} &= 1 - \frac{2}{\exp(2A + 2I_{i_0}w_{i_0j_0})+1}, \\ \tilde{z}_{j_0} &= 1 - \frac{2}{\exp(2A - 2I_{i_0}w_{i_0j_0})+1}. \end{aligned}$$

We note that by definition of tanh, $z_{i_0} < 1$ and $\tilde{z}_{i_0} < 1$.

$$\begin{aligned} \frac{1+z_{j_0}}{1-z_{j_0}} &= \exp(2A + 2I_{i_0}w_{i_0j_0}) \\ \frac{1+\tilde{z}_{j_0}}{1-\tilde{z}_{j_0}} &= \exp(2A - 2I_{i_0}w_{i_0j_0}), \end{aligned}$$

which gives

$$\begin{aligned} &\frac{(1+z_{j_0})(1-\tilde{z}_{j_0})}{(1-z_{j_0})(1+\tilde{z}_{j_0})} = \exp(4I_{i_0}w_{i_0j_0}) \\ \Longrightarrow\; &w_{i_0j_0} = \frac{1}{4I_{i_0}} \ln \frac{(1+z_{j_0})(1-\tilde{z}_{j_0})}{(1-z_{j_0})(1+\tilde{z}_{j_0})}. \end{aligned}$$

□

Corollary 6. *The attacker can recover the weight matrix W_S with mn faults and $2mn$ executions of the targeted neural network (the student neural network).*

3) Relu: In case the activation function for the last layer is relu, for an input $\boldsymbol{x}$, the output of the student neural network is given by

$$\mathcal{O}_\theta(\boldsymbol{x}) = \max\{0, \mathcal{O}_{\theta_{T,-1}}(\boldsymbol{x})^T W_S + \boldsymbol{b}_S\},$$

for $j = 1, 2, \ldots, m$,

$$\mathcal{O}_{\theta,j}(\boldsymbol{x}) = \max\{0, y_j(\boldsymbol{x})\}.$$

When the output of the activation function is 0, we cannot get much information. Thus, for relu, we need to consider an additional faulting position, position 5 in Figure 2. The effect of the fault is to flip the sign of the result of the summation, i.e. y_j with notation above. Thus in case the output of relu was original zero, after fault, the output would be absolute value of the summation, i.e. $-y_j$.

Given any $i_0 \in \{1, 2, \ldots, n\}, j_0 \in \{1, \ldots, m\}$ and any input $\boldsymbol{x}$. Then there are two steps for the attack:

Step 1:

1) If $\mathcal{O}_{\theta,j_0}(\boldsymbol{x}) \neq 0$. Let z_{j_0} denote $\mathcal{O}_{\theta,j_0}(\boldsymbol{x})$.
2) If $\mathcal{O}_{\theta,j_0}(\boldsymbol{x}) = 0$. The attacker executes the inference with the same input and inject SNIFF on the summation y_i. Let z_{j_0} denote the negative of the faulted value of $\mathcal{O}_{\theta,j_0}(\boldsymbol{x})$.

Step 2: The attacker executes the inference with the same input and inject SNIFF on target parameter - bias b_{S,j_0} or weight w_{i_0,j_0}.

1) If the faulted value of $\mathcal{O}_{\theta,j_0}(\boldsymbol{x}) \neq 0$. Let $\tilde{z}_{j_0}$ denote the faulted value of $\mathcal{O}_{\theta,j_0}(\boldsymbol{x})$.
2) Otherwise, the attacker executes the inference with the same input and inject SNIFF on both the target parameter and the summation y_i. Let $\tilde{z}_{j_0}$ denote the negative of the faulted value of $\mathcal{O}_{\theta,j_0}(\boldsymbol{x})$.

Theorem 7. *For any $j_0 \in \{1, 2, \ldots, m\}$ and any input $\boldsymbol{x}$. Following the attack steps described above, the target bias can be recovered as*

$$b_{S,j_0} = \frac{1}{2}(z_{j_0} - \tilde{z}_{j_0})$$

Proof. We note that with the above attack, we have

$$z_{j_0} = y_{j_0} = b_{S,j_0} + \sum_{i=1}^{n} I_i w_{ij_0}, \tilde{z}_{j_0} = \tilde{y_{j_0}} = -b_{S,j_0} + \sum_{i=1}^{n} I_i w_{ij_0}.$$

Thus

$$b_{S,j_0} = \frac{1}{2}(z_{j_0} - \tilde{z}_{j_0})$$

□

Corollary 7. *The attacker can recover the bias vector $\boldsymbol{b}_S$ with at most $3m$ faults and at most $4m$ executions of the target neural network (the student neural network).*

Theorem 8. *For any $i_0 \in \{1, 2, \ldots, n\}, j_0 \in \{1, \ldots, m\}$ and any $\boldsymbol{x}$, a non-vanishing input for i_0. Following the attack steps described above, the target weight $w_{i_0j_0}$ can be recovered as:*

$$w_{i_0j_0} = \frac{1}{2I_{i_0}}(z_{j_0} - \tilde{z}_{j_0}).$$

Proof. We note that with the above attack, we have

$$\begin{aligned} z_{j_0} &= y_{j_0} = b_{S,j_0} + \sum_{i=1}^{n} I_i w_{ij_0}, \\ \tilde{z}_{j_0} &= \tilde{y_{j_0}} = b_{S,j_0} - I_{i_0} w_{i_0j_0} + \sum_{i=1, i\neq i_0}^{n} I_i w_{ij_0}. \end{aligned}$$

Since $\boldsymbol{x}$ is a non-vanishing input for i_0, we have $I_{i_0} \neq 0$. We have

$$w_{i_0j_0} = \frac{1}{2I_{i_0}}(z_{j_0} - \tilde{z}_{j_0})$$

□

Corollary 8. *The attacker can recover the weight matrix W_S with at most $3mn$ faults and at most $4mn$ executions of the targeted neural network (the student neural network).*