# Physical Security of Deep Learning on Edge Devices: Comprehensive Evaluation of Fault Injection Attack Vectors

Xiaolu Hou[a], Jakub Breier[b,c], Dirmanto Jap[d], Lei Ma[e,f], Shivam Bhasin[d], Yang Liu[d]

[a] *Slovak University of Technology in Bratislava, Slovakia*
[b] *Silicon Austria Labs, TU-Graz SAL DES Lab, Austria*
[c] *Graz University of Technology, Austria*
[d] *Nanyang Technological University, Singapore*
[e] *University of Alberta*
[f] *Alberta Machine Intelligence Institute (Amii)*

**Abstract**

Decision making tasks carried out by the usage of deep neural networks are successfully taking over in many areas, including those that are security critical, such as healthcare, transportation, smart grids, where intentional and unintentional failures can be disastrous. Edge computing systems are becoming ubiquitous nowadays, often serving deep learning tasks that do not need to be sent over to servers. Therefore, there is a necessity to evaluate the potential attacks that can target deep learning in the edge.

In this work, we present evaluation of deep neural networks (DNNs) reliability against fault injection attacks. We first experimentally evaluate DNNs implemented in an embedded device by using laser fault injection to get the insight on possible attack vectors. We show practical results on four activation functions, ReLu, softmax, sigmoid, and tanh. We then perform a deep study on DNNs based on derived fault models by using several different attack strategies based on random faults. We also investigate a powerful attacker who can find effective fault location based on genetic algorithm, to show the most efficient attacks in terms of misclassification success rates. Finally, we show how a state of the art countermeasure against model extraction attack can be bypassed with a fault attack. Our results can serve as a basis to outline the susceptibility of DNNs to physical attacks which can be considered a viable attack vector whenever a device is deployed in hostile environment.

## 1. Introduction

With the success of deep learning (DL) and machine learning (ML) across domains, their combination with edge based computing is driving new innovation. This has also

*Email addresses:* xiaolu.hou@stuba.sk (Xiaolu Hou), jbreier@jbreier.com (Jakub Breier), djap@ntu.edu.sg (Dirmanto Jap), ma.lei@acm.org (Lei Ma), sbhasin@ntu.edu.sg (Shivam Bhasin), yangliu@ntu.edu.sg (Yang Liu)

motivated the rise of so called TinyML[1] TinyML refers to ML and DL models which are optimised and shrunk to fit and bring artificial intelligence into small devices and tiny hardware like embedded systems. These devices are being designed to be used in many areas of the industry and in everyday life, including tasks which require high level of security. Therefore, it is important to assess the available threat vectors that could compromise the integrity of the results.

In this work, we focus on a class of physical attacks known as fault attacks, which have become a reality owing to decreasing price and expertise required to mount such attack [22]. Fault attacks are active attacks on a given implementation which try to perturb the internal software/hardware computations by external means. Embedded systems deployed on the edge become a prime target owing to easy physical access and must be tested against such vulnerabilities. The adversary uses methods like voltage glitches, electromagnetic pulses, or laser injection to introduce perturbations for various purposes, ranging from erroneous computation, denial of service etc. Such attacks are commonly used for mounting secret key recovery attacks in cryptography or for violating/bypassing security checks [25]. In this paper, we analyze deep learning under fault attacks.

Deep learning is a family of neural networks composed of an input layer, three or more hidden layers and an output layer. Based on the internal structure, several candidates exist like multi-layer perceptron (MLP), convolutional neural networks (CNNs), recurrent neural networks (RNNs) etc. These are popularly known as deep neural networks (DNN). While each of these architectures has unique functions, we focus on activation functions which remain common across architectures and are an important part of the algorithm to obtain non-linear behaviors [20]. These commonly used activation functions are: softmax, ReLu, sigmoid and tanh. Studying these functions under fault attacks allows deriving general conclusions on susceptibility of deep learning to fault attacks.

We implemented the most common activation functions used across DNNs on a low-cost microcontroller (often used in IoT). Next, we performed *practical laser fault injection* using a near-infrared diode pulse laser to inject faults during the processing of activation function. With fault models, derived from practical fault injection, we analyze in detail the susceptibility of DNN against such attacks. The primary goal of the performed attacks is to achieve miss-classification during the testing phase, thus forcing the network to report an incorrect label for a given input. In the hindsight, the achieved miss-classification can jeopardize the functioning of DNN-based paradigms like smart city, affecting the reliability of the whole application.

Extensive studies have been performed on adversarial attacks, that crafts the input data with little perturbation to fool deep learning systems [21, 23, 39, 13, 12, 11, 41] by miss-classification. In our study we explore practical (physical) fault injection on deep neural network, where we focus on attacking the DNNs itself instead of creating input data to fool DNNs like adversarial attack does. We evaluate different ways of selecting neurons to fault, from random selection to optimized method using a genetic algorithm. Our results indicate that in some cases, a relatively small number of faulted neurons ($\approx$

[1] tinyML Foundation. tinyml summit, 2019. URL https: //www.tinymlsummit.org/.

10%) can already present a high risk of misclassification ($\approx$ 70%). Our attack requires a whitebox access to the neural network. This means that the attacker knows the internal structure of the neural network and also the model parameters. This is a relatively practical scenario as many optimized models are made public (e.g. Resnet [42], GoogLeNet [38], ...). However, there are so-called *model extraction* attacks which can be applied in blackbox setting to get the model parameters and therefore allow whitebox access for the attacker [40, 35]. While there are several countermeasures against model extraction, we additionally show how a fault attack can be applied to bypass one type of such countermeasure.

**Organization.** The rest of the paper is organized as follows. Section 2 provides the necessary background on laser fault injection and activation functions of neural networks. Section 3 presents the details of the experiment, with the explanations of the effect of faults on activation functions. These findings are applied on full DNN in Section 4. An attack strategy based on findings from genetic algorithm is presented in Section 5. Section 6 discusses how fault attack can be applied to bypass *PRADA* [26], a countermeasure against model extraction attacks. Section 8 concludes this paper and provides a motivation for follow up work.

## 2. Background

In this section, we recall basic concepts of deep neural networks, activation functions and fault injection attacks.

### *2.1. Deep Neural Networks*

Artificial neural networks (ANNs) are computing units designed on basis of biological neural networks. ANN is a network of interconnected nodes or neurons where a signal is transmitted from input neurons towards output neurons. Arranged in layers, each neuron computes an output based on sum of (weighted) inputs from other neurons, followed by a non-linear function. The weights are determined during the training process. The non-linear layer function, also known as the activation function, is what gives an ANN its power to learn and classify difficult problems. A simple ANN can be composed of an input layer, one hidden layer and an output layer. To train the network, the backpropagation algorithm is used, which is a generalization of the least mean squares algorithm in the linear perceptron. Backpropagation is used by the gradient descent optimization algorithm to adjust the weight of neurons by calculating the gradient of the loss function [32].

Deep neural networks (DNNs) are fairly new variants of ANNs with three or more hidden layers. DNNs have become realistic with the latest advances in computing power, thanks to high performance graphical processing units (GPU). Several variants of DNN exist, including multi-layer perceptron (MLP), convolutional neural networks (CNNs), recurrent neural networks (RNNs), etc. Owing to the deep architecture, they have shown great success across domains – the most prominent being image classification, with the biggest ones composed of as many as 152 layers (Resnet [42]).

As it was pointed out in [34], in case of large neural networks, there are many nodes that do not contribute to the neural network function. However, there are some nodes

which are crucial for correct functionality and if these are faulted, it can result in a failure.

### 2.2. Activation Functions

The activation functions we consider are the following: `softmax`, `ReLu`, `sigmoid` and `tanh` [20].

Softmax is normally used as the activation function for output layer. It takes a vector $\mathbf{x}$ as input, $i$th entry of the output gives the probability of a given input belonging to class $i$:

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}, \tag{1}$$

where exp is the exponentiation function with base $e$.

In modern neural networks, the default recommendation for activation function is the rectified linear unit or ReLu defined as follows:

$$\text{ReLu}(x) = \max\{0, x\}. \tag{2}$$

It is a piecewise linear function which preserves properties that make the optimization of linear model easy.

Before the introduction of ReLu, commonly used activation functions are logistic sigmoid activation function

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}, \tag{3}$$

and hyperbolic tangent function

$$\tanh(x) = \frac{2}{1 + \exp(-2x)} - 1. \tag{4}$$

The sigmoid function is normally used to introduce non-linearity in the model. A reason for its popularity comes from the simple equation between its derivative and itself

$$\text{sigmoid}'(x) = \text{sigmoid}(x)(1 - \text{sigmoid}(x)).$$

However, sigmoid functions become insensitive to inputs with large absolute values. In such cases, the hyperbolic tangent activation function is used as an alternative.

### 2.3. Fault Injection Attacks

Fault injection attacks are a popular physical attack vector used against cryptographic circuits [5]. By changing intermediate values during the cryptographic algorithm execution, they can efficiently provide information on secret values, helping to recover the secret key in just a few encryptions [6, 7, 10]. Normally, the secret key recovery would require infeasible amount of computing time. Similarly, these attacks can be used against verification circuits, such as `PIN` verification on a smartcard, where a comparison function can be skipped and grant access to a malicious user [19].

When it comes to fault injection techniques, there are several options one can use, mostly depending on the adversary budget and expertise [4]. The most basic methods include variations in voltage or clock signal, allowing disturbance of instruction sequences in microcontrollers [3]. Electromagnetic fault injection allows more precise location targeting, enabling faults in memories [27, 33]. Laser fault injection is the most precise from commonly used techniques, being capable of flipping single bits [2].

Up to date, to the best of our knowledge, only [30] describes fault injection attack on neural networks. In their paper, they only provide a white box attack on deep neural network through software simulation, while observing the changes in the output after introducing faults in the network's values. However, they do not provide insight on practicality of such attack. Whether such attacks could also be applied physically remained an open problem. Therefore, in our paper, we experimentally show what types of faults are achievable in practice and we further use this information to develop a realistic attack on DNNs.

### *2.4. Difference from Adversarial Learning*

A huge amount of research is undergoing towards adversarial learning [31]. It basically involves constructing special inputs which are capable of confusing the machine learning models, often leading to output misclassification. In this work, we explore an alternate avenue to arrive at the same but by different means. The proposed fault attacks target the implementation of the DNN, particularly the critical activation function to achieve misclassification without any perturbation of the input. Depending on the application scenario and adversary model, one attack might be more suited than the other.

## 3. Practical DNN attack feasibility analysis

In this part we first show the practical laser fault attack setup in Section 3.1. In Section 3.2 we show the possible fault attacks on activation functions that we have discovered with practical experiments. In Section 4, those attacks will be used for simulating missclassification attacks on MNIST DNNs.

### *3.1. Attack Equipment Setup*

The main component of the experimental laser fault injection station is the diode pulse laser. It has a wavelength of 1064 nm and pulse power of 20 W. This power is further reduced to 8 W by a 20× objective lens which reduces the spot size to 15×3.5 $\mu m^2$.

As the device under test (DUT), we used `ATmega328P` microcontroller, mounted on `Arduino UNO` development board. The package of this chip was opened so that there is a direct visibility on a back-side silicon die with a laser. The board was placed on an `XYZ` positioning table with the step precision of 0.05 $\mu m$ in each direction. A trigger signal was sent from the device at the beginning of the computation so that the injection time could be precisely determined. After the trigger signal was captured by the trigger and control device, a specified delay was inserted before laser activation. Laser activation timing was also checked by a digital oscilloscope for a greater precision. Our setup is depicted in Figure 1.

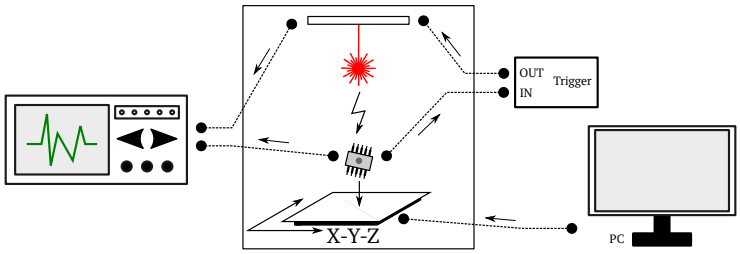


Figure 1: Experimental laser fault injection setup.

### *3.2. DNN Activation Function Fault Analysis*

To evaluate different activation functions, we implemented three simple 3-layer neural networks with sigmoid, ReLu and tanh as the activation function for the second layer respectively. The activation function for the last layer was set to be softmax. The neural networks were implemented in C programming language, which were further compiled to AVR assembly and uploaded to the DUT.

We surrounded the activation functions in the second layer with a trigger signal that raised a voltage on a selected Arduino board pin to 5 V, helping us to determine the proper laser timing.

As instruction skip/change is one of the most basic attacks on microcontrollers, with high repeatability rates [10], we aimed at this fault model in our experiments. The microcontroller clock is 16 MHz, one instruction takes 62.5 ns. Some of the activation functions took over 2000 instructions to execute. To check what are the vulnerabilities of the implementations, we have carefully varied the timing of the laser glitch from the beginning until the end of the function execution so that every instruction would be eventually targeted.

Please note that we used a single fault adversarial model, meaning that exactly one fault was injected during one activation function execution. We consider an attack is successful for a given input data if the output classification is different from the classification obtained by the original network. And we refer to such a successful attack as misclassification.

After we observed a successful missclassification, we determined the vulnerable instructions by visual inspection of the compiled assembly code and by checking the timing of the laser in that particular fault injection instance. Area of the chip vulnerable to these disturbances is depicted in Figure 2. The chip area is 3×3 $mm^2$, while the area sensitive to laser is $\approx$ 70×100 $\mu m^2$. With a laser power of 4.5% we were able to disturb the algorithm execution, when tested with reference codes. More details on the behavior on this particular microcontroller under laser fault injection can be found in [10] while the sample preparation and guidance on the laser experiments is provided in [8].

In this exploratory study, we implemented a random neural network, consisting of 3 layers, with 19, 12, and 10 neurons in input layer, hidden layer, and output layer, respectively. Our fault attack was always targeting the computation of one of the activation functions in hidden layer. In the following, we will explain the experimental

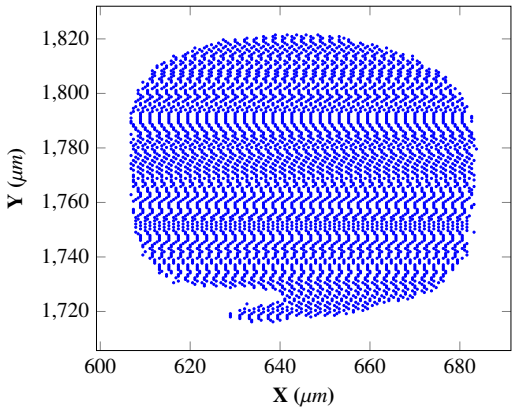


Figure 2: Area plot depicting successful instruction skip experiments.

results on different activation functions in detail.

**ReLu.** This function is implemented by a following code in C:

```
if (Accum > 0) {
   HiddenLayerOutput[i] = Accum;}
else {
   HiddenLayerOutput[i] = 0;}
```

where *i* loops from 1 to 12 so that each loop gives one output of the hidden layer. `Accum` is an intermediate variable that stores the input of activation function for each neuron.

The assembly code inspection showed that the result of successful attack was executing the statement after else such that the output would always be 0. The corresponding assembly code is as follows:

```
1        ldi r1, 0       ;load 0 to r1
2        cp r1, r15      ;compare MSB of Accum to r1
3        brge else       ;jump to else if 0 >= Accum
4        movw r10, r15   ;HiddenLayerOutput[i] = Accum
5        movw r12, r17   ;HiddenLayerOutput[i] = Accum
6        jmp end         ;jump after the else statement
7  else: clr r10         ;HiddenLayerOutput[i]= 0
8        clr r11         ;HiddenLayerOutput[i]= 0
9        clr r12         ;HiddenLayerOutput[i]= 0
10       clr r13         ;HiddenLayerOutput[i]= 0
11 end:  ...             ;continue the execution
```

where each float number is stored in 4 registers. For example, `Accum` is stored in registers `r15,r16,r17,r18` and `HiddenLayerOutput[i]` is stored in `r10,r11,r12,r13`. Line 4,5 executes the equation `HiddenLayerOutput[i] = Accum`.

The attack was skipping the "`jmp end`" instruction that would normally avoid the part of code setting `HiddenLayerOutput[i]` to 0 in case `Accum` > 0. Therefore, such change in control flow renders the neuron inactive no matter what is the input value.

**Sigmoid.** This function is implemented by a following code in C:

| Target activation function | Relation between $y$ and $y'$ |
|---|---|
| ReLu | $y' = 0$ |
| sigmoid | $y' = 1 - y$ |
| tanh | $y' = -y$ |

Table 1: Relation between correct output $y$ and faulted output $y'$ when a single fault is injected in target activation function

```
HiddenLayerOutput[i] = 1.0/(1.0 + exp(-Accum));
```

After the assembly code inspection, we observed that the successful attack was taking advantage of skipping the negation in the exponent of `exp()` function, which compiles into one of the two following codes, depending on the compiler version:

```
A)  neg r16          ;compute negation r16
B)  ldi r15, 0x80    ;load 0x80 into r15
    eor r16, r15     ;xor r16 with r15
```

Laser experiments showed that both `neg` and `eor` could be skipped, and therefore, significant change to the function output was achieved.

**Hyperbolic tangent.** This function is implemented by a following code in C:

```
HiddenLayerOutput[i] = 2.0/(1.0 + exp(-2*Accum)) - 1;
```

Similarly to sigmoid, the experiments showed that the successful attack was exploiting the negation in the exponential function, leading to an impact similar to sigmoid.

**Softmax.** In case of softmax function, we were unable to obtain any successful misclassification. There were only two different outputs as a result of the fault injection: either there was no output at all, or the output contained invalid values. This lack of valid output prevented us to do further fault analysis to derive the actual fault model that happened in the device. Therefore, a thorough analysis of softmax behavior under faults would be an interesting topic for the future work. Another line of future work would be to analyze bit flip attacks [2]. The first application of such attack would be to target IEEE 754 floating point representation that is used for storing the weights. The representation follows 32-bit pattern ($b_{31}...b_0$): 1 sign bit ($b_{31}$), 8 exponent bits ($b_{30}...b_{23}$) and 23 mantissa (fractional) bits ($b_{22}...b_0$). The represented number is given by $(-1)^{b_{31}} \times 2^{(b_{30}...b_{23})_2 - 127} \times (1.b_{22}...b_0)_2$. A bit flip attack on the sign bit or on the exponent bits would make significant influence on the weight. Another application of bit flip attack would be to fault interconnecting weights, resulting to incorrect input to the next layer. We leave both directions for future investigation as they are out of scope for the current work.

If we let $y$ and $y'$ denote the correct and faulted output of the target activation function, the relation between $y$ and $y'$ is summarized in Table 1. For further illustration, the graph of original and faulted activation functions is depicted in Figure 3.

## 4. Application to DNN

The results from previous section aiming at single functions can be directly used to alter the behavior of a neural network. In this section we extend the attack to a

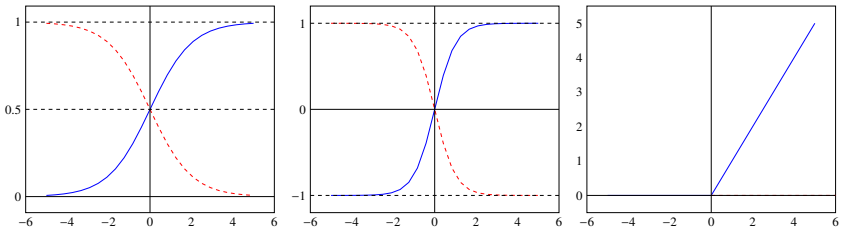


Figure 3: (a) Sigmoid, (b) Hyperbolic tangent, and (c) ReLu functions. Blue lines indicate original function, red lines indicate faulted ones.

full network, while targeting several function computations at once with a multi-fault injection model. When it comes to deep neural networks, there are three possible places to introduce a fault:

- Input layer – such fault would be identical to introducing a change at the input data. Therefore, it is of little interest, since it would be normally easier for the attacker to directly alter the input data rather than injecting precise faults with an expensive equipment.
- Hidden layer(s) – since the structure of the hidden layer is normally unknown to the attacker, she cannot easily predict the outcome of the fault injection. However, she can still achieve the missclassification, although not necessarily to the class she decides. Therefore, such attack might be interesting in case the attacker does not care about the outcome class as long as it is different from the correct outcome.
- Output layer – normally, softmax is the function of choice for the output layer. According to our results, introducing a meaningful fault into softmax is harder compared to other functions. However, as we discussed, in case the attacker can alter registers storing the floating point data, she can easily missclasify the outcome to a chosen class, making it a very powerful attack model.

Deciding on what layer to attack, it makes sense to inject the fault as close to the output layer as possible to make the impact highest. Therefore, for our case, the attacker injects faults into the last hidden layer of the network, targeting multiple activation function computations.

In the following we consider DNNs severed for classification purposes and the activation function of the output layer is sofmax. We further assume the output layer is dense and the goal of the attacker is to misclassify an input. In Section 4.1 we discuss the possible strategies of an attacker. In Section 4.2 we present the evaluation results using the strategies on a sample DNN.

### *4.1. Algorithms for attacking the last hidden layer*

We model the last two layers of a DNN as follows: let $\mathbf{x}$ denote the output of the last hidden layer and let $W$ and $B$ denote the matrix of weights and the vector of bias weights for output layer. Let $\mathbf{z}$ denote the input of softmax function. Suppose

there are $m$ neurons in the last hidden layer and $n$ neurons in the output layer. Let $W_k$, $k = 1, 2, \ldots, n$ be the columns of $W$. Then the output is given by

$$\text{output}_i = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} = \frac{\exp(\mathbf{x}W_i + B_i)}{\sum_{j=1}^{n} \mathbf{x}W_j + B_j}, \quad i = 1, 2, \ldots, n.$$

The final classification is given by $\ell$ such that $\max_i \text{output}_i = \text{output}_\ell$. For any sequence of $z_j$, $j = 1, 2, \ldots, n$, we have

$$\max_i \text{output}_i = \max_i \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} = \frac{\max_i \exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} = \frac{\exp\left(\max_i z_i\right)}{\sum_{j=1}^{n} \exp(z_j)}.$$

Hence the output classification is equal to $\ell$ such that $\max_i z_i = z_\ell$.

The attacker injects faults in the computation of the activation functions for neurons in the last hidden layer and gets a faulted $\mathbf{x}'$. Correspondingly we have a faulted vector $\mathbf{z}'$. Thus, for a given input with correct classification $\ell$, the goal of misclassification is equivalent to: achieve $\mathbf{z}'$ such that there exists $j$ with $z'_j > z'_\ell$ or $z'_j - z'_\ell > 0$. Consequently, an input can be misclassified if and only if

$$\begin{aligned}
(\mathbf{x}'W_j + B_j) - (\mathbf{x}'W_\ell + B_\ell) &> 0 \\
(\mathbf{x}W_j + B_j + (\mathbf{x}' - \mathbf{x})W_{jk}) - (\mathbf{x}W_\ell + B_\ell + (\mathbf{x} - \mathbf{x})W_{\ell k}) &> 0 \\
\mathbf{x}W_j + B_j - \mathbf{x}W_\ell - B_\ell + (\mathbf{x}' - \mathbf{x})(W_{jk} - W_{\ell k}) &> 0 \\
z_j - z_\ell + (\mathbf{x}' - \mathbf{x})(W_{jk} - W_{\ell k}) &> 0 \\
z_j - z_\ell + \sum_{x'_k \neq x_k} (x'_k - x_k)(W_{jk} - W_{\ell k}) &> 0 \qquad (5)
\end{aligned}$$

Algorithm 1 gives matrix $A$ such that $A[k][j] = (x'_k - x_k)(W_{jk} - W_{\ell k})$ and diagonal matrix $D$ whose diagonal is given by $\mathbf{x}' - \mathbf{x}$.

**Single fault strategy.** When a single fault model is considered, $\mathbf{x}$ and $\mathbf{x}'$ only differs in one entry, say $x_k$. Equation (5) becomes

$$z_j - z_\ell + (x'_k - x_k)(W_{jk} - W_{\ell k}) > 0 \qquad (6)$$

For a given DNN and target input, Algorithm 2 outputs $k$, the neuron to attack so that a misclassification can be achieved. Line 2 calculates the matrix $A$ with column $i$ given by $W_i - W_\ell$. Depending on the activation function, $\mathbf{x}'$ is related to $\mathbf{x}$ as described in Table 1. After line 13, the $(k, j)-$entry of matrix $A$ is given by $(x'_k - x_k)(W_{jk} - W_{\ell k})$. Line 15 checks if Equation (6) is satisfied for any $j, k$. If it can be satisfied for some $k$, $j$, the target input can be misclassified with a fault attack on neuron $k$.

For multiple fault model, a natural strategy is **random faults**, i.e. random number of neurons in the last hidden layers are faulted. Here we provide another strategy which utilizes the information of weights and bias of the last layer.

**Multiple faults strategy.** For a target input with correct class $\ell$, we aim to find a list of neurons to attack so that the probability of class $\ell$ in the output will be reduced. Details are given in Algorithm 3.

**Algorithm 1:** Calculation of matrix $A$

**Input :** $W$: matrix of weights for the last layer with columns $W_1, W_2, \ldots, W_n$; $B$ vector of bias weights for the last layer; $\ell$: the correct class of target input; $\mathbf{x}$: output of the last hidden layer for target input; activation function: ReLu, sigmoid or Tanh.

**Output:** Matrices $A, D$.

```
for i = 1, 2, ..., n do
    A[i] = W_i − W_ℓ;
if activation function is ReLu then
    for k = 1, 2, ..., m do
        x'[i] = 0;
if activation function is sigmoid then
    for k = 1, 2, ..., m do
        x'[i] = 1 − x[i];
if activation function is Tanh then
    for k = 1, 2, ..., m do
        x'[i] = −x[i];
D = diagonal matrix with diagonal x' − x;
A = DA;
return A, D;
```

### *4.2. Evaluation of a sample DNN*

To test how our attack can influence a real-world DNN, we trained and evaluated different DNNs with the attack strategies described above. The attack vectors considered are as described in Section 3.2. We have selected a popular MNIST dataset [28]. The training of DNNs was accomplished using Keras (ver.2.1.6) [14] and Tensorflow libraries (ver.1.8.0) [1]. The structures of the DNNs are detailed in Table 2. For each target function (ReLu, sigmoid and tanh), 10 DNNs with different number of neurons ($n = 50, 100, 150, 200, 250, 300, 350, 400, 450, 500$) in hidden layer 4 were evaluated. We used a partially fixed structure of DNN in order to study the effects of fault attacks on different activation functions. The prediction accuracy we obtained is summarized in Table 3. The accuracy shows that although the DNNs we choose are relatively simple, their accuracy is comparable with the state of the art. Success rates are calculated for 800 random inputs.

For multiple fault model, we evaluated the DNNs with number of faults equal to $10, 20, 30, 40, 50$ percent of the number of neurons in hidden layer 4. The simulation results for targeting activation function being ReLu, Sigmoid and tanh are presented in Figures 4, 5 and 6 respectively.

Overall, it can be concluded that in case of sigmoid and tanh, if the attacker wants to have a reasonable success rate (>50%), she should inject faults in at least 40% of the neurons using multiple faults strategy in the chosen layer. But for ReLu, when the number of neurons is big, the DNN becomes more resistant to fault attacks.

The results also show that sigmoid and tanh functions follow the same trend, which

**Algorithm 2:** Single fault strategy

**Input :** $A$:obtained from Algorithm 1; $\mathbf{z}$: input of softmax function.
**Output:** `True/False` indicating if an attack exists or not; $k$ s.t. the input can be misclassified with fault attack on neuron $k$.

```
for k = 1, 2, ..., m do
    for j = 1, 2, ..., n, j ≠ ℓ do
        if z_j − z_ℓ + A[k][j] > 0 then
            output k;
            return True;
return False;
```

**Algorithm 3:** Multiple faults strategy

**Input :** $D$: obtained from Algorithm 1; $W_\ell$: the $\ell$th column of $W$; $M$: number of faults.
**Output:** `indices`: a list of neurons to attack.

```
indices= [];
B = DW_ℓ;
for k = 1, 2, ..., m do
    if B[k][j] < 0 then
        add k to indices;
        if length of indices== M then
            return indices;
return indices;
```

is caused by the same type of fault as explained in the previous section – skipping the negation in the exponentiation function.

## 5. Genetic algorithm for attacking the whole DNN

A natural question to ask is "what if we assume the attacker can target any neurons in the whole DNN?", and "how many neurons does she need to attack to achieve a certain percentage of misclassification?"

To find answer these questions, we analyzed three different DNNs with structures given in Table 4, where the target activation functions are ReLu, Sigmoid, tanh, respectively. Similarly to Section 4.2, the DNNs were trained using Keras (ver 2.1.6.) on MNIST dataset. The training and testing accuracies are summarized in Table 5. The aim of the experiment was to check the effect on the DNN when a certain percentage of neurons is attacked. For this purpose, we have adopted the genetic algorithm to help in searching for the vulnerable collections of neurons in a given DNN.

Genetic Algorithm (GA) is a computational intelligence heuristic algorithm normally used for optimization problems, based on the concept of natural selection. For optimization problems with large search space, it is often a preferable choice compared to brute-force search, since it can help to reduce the search time for finding the solution.

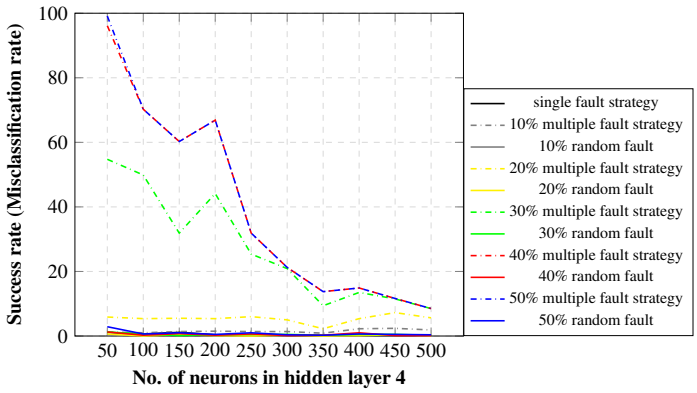


Figure 4: Target activation function – ReLu.

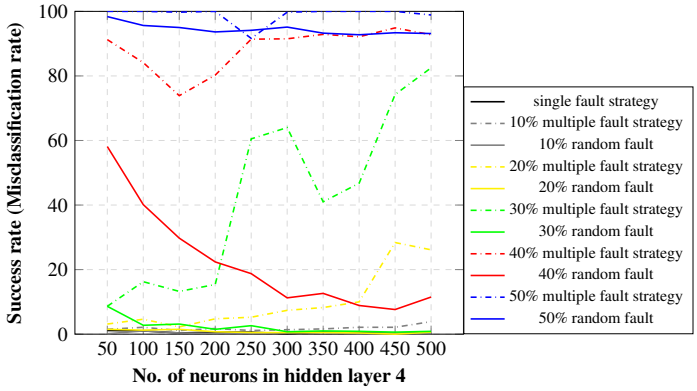


Figure 5: Target activation function – Sigmoid.

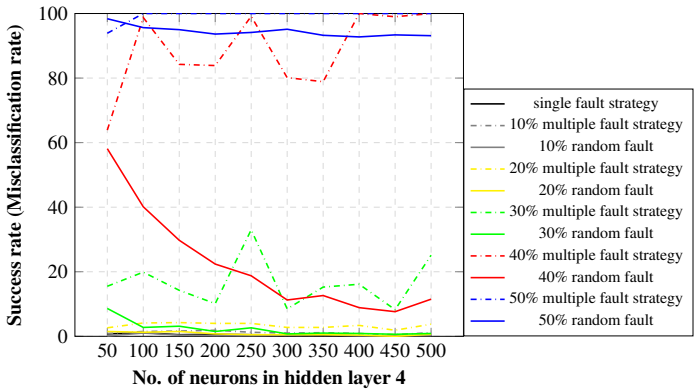


Figure 6: Target activation function – tanh.

| Layer | No. of neurons | Activation function |
|---|---|---|
| Input layer | 784 | - |
| Hidden layer 1 | 500 | ReLu |
| Hidden layer 2 | 500 | ReLu |
| Hidden layer 3 | 500 | ReLu |
| Hidden layer 4 | $n$ | target activation function |
| Output layer | 10 | Softmax |

Table 2: Structure of the DNN used in evaluations.

| Target | ReLu | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| n | 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 | 500 |
| Train. Acc. | 99.2 | 99.2 | 99.4 | 98.8 | 99.1 | 99.0 | 99.2 | 98.4 | 98.9 | 99.1 |
| Test. Acc. | 97.4 | 97.9 | 98.0 | 97.4 | 97.7 | 97.5 | 97.8 | 97.3 | 97.5 | 98.0 |
| Target | sigmoid | | | | | | | | | |
| n | 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 | 500 |
| Train. Acc. | 99.2 | 99.0 | 99.3 | 99.0 | 99.3 | 99.3 | 99.4 | 99.1 | 99.3 | 99.4 |
| Test. Acc. | 98.0 | 97.7 | 98.0 | 97.6 | 98.1 | 98.0 | 98.0 | 97.7 | 98.1 | 98.0 |
| Target | tanh | | | | | | | | | |
| n | 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 | 500 |
| Train. Acc. | 99.0 | 99.0 | 98.2 | 99.1 | 99.1 | 99.3 | 99.0 | 98.9 | 99.2 | 98.9 |
| Test. Acc. | 98.0 | 97.5 | 97.8 | 97.8 | 97.8 | 98.0 | 97.6 | 97.7 | 98.1 | 97.4 |

Table 3: Training/testing accuracy of DNNs used in evaluation.

Though it does not guarantee finding a perfect solution, it is an alternative approach that finds a good enough solution, while saving the computational resources significantly. GA itself has been applied as well for fault attacks problems, for example, to search for optimal experiment parameters for fault injection [36].

Typically, the standard GA method is to assign fitness values for each individuals within the search space. A population of these individuals is initialized randomly according to the specification for the population. For each generation (or iteration), the algorithm selects better individuals and removes the worse ones, while combining different individuals using crossover algorithm to generate new ones. The evaluation is performed according to the fitness function defined, and the aim is to find an individual which could optimize the fitness value in the search space. Normally, to avoid converging to local optima, a mutation function is introduced by randomly changing parts of the new individuals.

In our experiment, we use DEAP [17] for the GA implementation. DEAP is an evolutionary algorithm library in Python. Since we are using Keras for our DNN implementation, DEAP can be easily adopted and integrated for the experiments. Our GA follows a standard structure as shown in Algorithm 4. Here we explain how each component of GA was implemented:

**Algorithm 4:** Genetic Algorithm (GA) for attacking the whole DNN

**Input :** DNN structure, `noOfFaults`: number of faults, `noGen`: number of generation
**Output:** `indices`: a list of neurons to attack.

```
P = Generate Population(noOfFaults);
Evaluate(P);
for i in range(noGen) do
    Crossover(P);
    Mutation(P);
    Evaluate(P);
    Selection(P);
return the best individual in P;
```

- `Individual`: Each individual is generated as a binary vector whose length is the number of neurons in the hidden layers of the neural network. For DNNs we evaluated (see Table 4), each individual has length 800. As we consider faults to be inserted randomly in the hidden layers, we do not differentiate to which layer the faulted neuron belongs, that is why the individual is of vector shape. A 0 in index $i$ would indicate the $i$th neuron is not attacked and a 1 in index $j$ would indicate the $j$th neuron will be attacked. Naturally, The number of 1s is equal to the number of faults allowed.

- `Fitness function:` The fitness of an individual is the corresponding misclassification rate – more precisely, we calculate the percentage of misclassified image by faulting the network according to the fault model represented by the individual.

- `Population`: In our experiments, we set size of population to be 200 and number of generations to be 120. These numbers were selected for practical reasons, as higher values would yield impractical computation times.

- `Selection` The selection of next generation follows tournament selection with tournament size 3.

Regarding the crossover and mutation, we followed the selection guidelines stated in [16]. In general, it is advised to select lower values for these parameters in case of binary values.

- `Crossover`: For each pair of individuals in the population, the crossover rate is set to be 0.78. This value is relatively high because of the size of the search space in our problem – crossover handles the *exploration* part of the GA, which means searching through the available space [43]. The offsprings are obtained by performing two-point crossover.

- `Mutation`: Mutation is performed in order to avoid falling for local minima in the search space. In this experiment, flip bits are used for mutation. The mutation rate was chosen to be 0.05. We chose a relatively low mutation rate to

| Layer | No. of neurons | Activation function |
|---|---|---|
| Input layer | 784 | - |
| Hidden layer 1 | 200 | target activation function |
| Hidden layer 2 | 200 | target activation function |
| Hidden layer 3 | 200 | target activation function |
| Hidden layer 4 | 200 | target activation function |
| Output layer | 10 | Softmax |

Table 4: Structure of the DNN used in evaluation for attacking the whole network.

| Activation function | Training Accuracy | Test Accuracy |
|---|---|---|
| ReLu | 99.9 | 98.7 |
| Sigmoid | 99.3 | 97.6 |
| tanh | 99.9 | 98.1 |

Table 5: Training/test accuracy of DNNs used in evaluation for attacking the whole network.

avoid reducing the algorithm to a random search, but significant enough to get a good convergence.

In each generation, new individuals have to be checked to ensure that they satisfy the constraint in the original problem, namely, the number of 1s is equal to number of faults allowed. We include this constraint in the evaluation step – we penalize the outliers by assigning zero score, to exclude them from the next generation.

Figure 7 shows the success rate of misclassification when the neurons are selected by using GA, compared to random selection. It shows that especially in case of Sigmoid and ReLu, careful choice of which neurons to fault can increase the success rate significantly. To summarize, the result can be improved up to 62% in case of ReLu, 31% in case of Sigmoid, and 20% in case of tanh.

## 6. Attack on *PRADA* Countermeasure

In many cases, the model parameters are kept confidential as training models require computing resources and training data. The trained model is thus considered an intellectual property of the organization that has developed it. As a result, several attacks that recover these parameters, called *model extraction* attacks [40, 35] were proposed. Naturally, another branch of research that provides protection against the model extraction has emerged. In this section we show how fault attack can be used to bypass a countermeasure, entitled *PRADA* [26], so that existing attacks can be carried out to extract the model. *PRADA* is among the state of the art countermeasures preventing most commonly known model extraction attacks.

*PRADA* [26] is a model extraction attack detection method proposed in 2019. This detection method analyzes a distribution of consecutive queries and it considers the

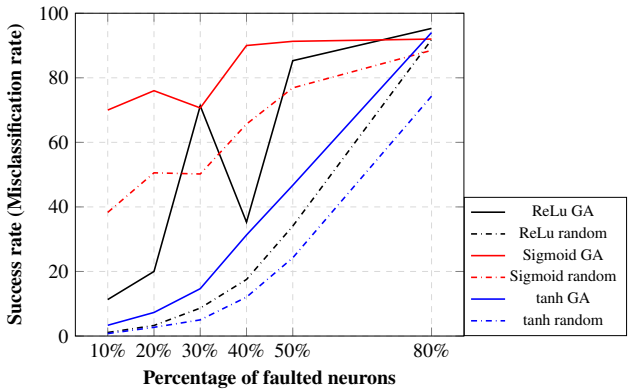


Figure 7: Evaluation results using genetic algorithm (GA) to select neurons versus random selection.

queries as malicious if the distribution deviates from a normal distribution. Once malicious queries are detected, the authors propose to return wrong predictions instead of the correct ones. For our attack, we follow their public implementation[2], where once malicious queries are detected, the neural network model randomly returns one of the first three classes with the highest probabilities. The random selection was implemented by shuffling the top three classes. To show the behavior of the countermeasure on embedded devices, we implemented *PRADA* in C code suitable for running on Arduino platform.

The idea of the defense mechanism is realized by the following Python code:

```
res = shuffle_max_logits(logits, 3) if attacker_present else logits
```

where `logits` contains the correct output and `shuffle_max_logits(logits: np.ndarray, n: int)` is a function that shuffles the $n$ largest entries of `logits` and returns the shuffled array. The corresponding C code in our implementation is as follows

```
1  if (attacker_present) {
2        shuffle_max_logits(res,3);
3  }
```

Below we state the AVR assembly for the previous statement in C. Please note that register `r16` contains the value of `attacker_present`:

```
1  ldi r17, 0x01              ;load 1 to r17
2  cp r16, r17                ;compare r16 and r17
3  breq shuffle_max_logits    ;jump to shuffle_max_logits if they are equal
```

We note that an instruction skip on line 3 can skip the function call to the function `shuffle_max_logits`. In this case even if attacker is detected, the logit values will not be shuffled and we can bypass the countermeasure completely. Given the execution

[2] `https://github.com/SSGAalto/prada-protecting-against-dnn-model-stealing-attacks`

| Strategy | Attack target | No. of faults | Precision |
|---|---|---|---|
| Single fault | Last layer | 1 | exact |
| Random fault | Last layer | multiple | random |
| Multiple fault | Last layer | multiple | exact |
| Genetic algorithm | Full network | multiple | exact |

Table 6: Similarities and differences of proposed attack strategies.

delay of target neural network is orders of magnitude higher than the triggering rate of the laser injection system, the fault can target each execution individually with near perfect repetition rate.

## 7. Discussion, Analysis and Limitations

The presented study shows possibilities of causing misclassifications on neural networks with three major activation functions. It has to be noted that these attacks assume a relatively strong adversary model – the attacker has to be able to physically tamper with the device to flip the bits either in memory or during the activation function execution. Compared to traditional adversarial attacks [39], which focus on input perturbations, fault attack is relatively harder to perform. On the other hand, standard protection techniques would check the integrity of the input and therefore, would miss the perturbations that happen during the computation. Also, in case the input is stored somewhere for further checking in case the model fails, it is easy to identify the standard adversarial attacks additionally, but in case the model fails due to a fault attack, there will not be any evidence.

We have presented four attack strategies: *single fault*, *random fault*, *multiple fault* and *selection of faulted neurons with the genetic algorithm*. The similarities and differences of those strategies are summarized in Table 6.

In terms of the attack target, the first three strategies focus on attacking the last layer of a neural network and the last strategy can be applied to the whole network. Thus, the attack with the genetic algorithm explores more vulnerable spots in the neural network and provides better security analysis of the network.
The single fault strategy only requires one fault, which makes it easier to perform compared to the other strategies.
Based on the analyzed attack strategies, we can further differentiate the attacker's abilities into two classes: (1) capable of randomly targeting neurons and (2) capable of selectively targeting neurons in the network. *Random fault* strategy falls in the first class and the other three strategies fall within the second class. While the first class can be achieved relatively easily once the device is in possession of the adversary, for example by underpowering the device or by electromagnetic emanations, the second class requires internal knowledge of the network structure and the timing information on when the individual neurons are executed.

In the area of open problems and suggestions for future work, we believe there are two promising directions: (1) focusing on optimizing the number of faults to achieve the misclassification, and (2) finding a stealthy and possibly even remote way to cause

faults. The optimization of the number of faults has already gained interest in the AI security community [24]. In the paper, they study the degradation of the network caused by fault attacks and try to find a single bit flip which causes the biggest drop in the accuracy. However, as the number of parameters is normally huge and the number of bits which form these parameters is even larger, we believe these results can be still improved.

Regarding the stealthy and remote way to cause faults, the rowhammer-based approaches seem like the best option at the moment, although their practicality can be argued. Memory manufacturers react quickly to remove the erroneous behavior of their chips, so every time there is a modification to the chips, a slightly different technique has to be developed. To the best of our knowledge, the current state-of-the-art rowhammer attack can target DDR4 memory chips [18].

## 8. Conclusion and Future Work

In this paper, we evaluated reliability of activation functions of deep neural networks implemented in embedded devices against laser fault injection. With these results, we simulated various scenarios on how the entire network can be affected in case multiple neurons are targeted during one execution. With the help of genetic algorithm, we managed to achieve high misclassification rates ($\approx 70\%$) with a relatively low number of faulted neurons (10%). Additionally, we show that with a relatively simple instruction skip attack, one can skip a countermeasure against model extraction attack.

In the future, it would be interesting to look at fault injection countermeasures that would fit implementations of neural networks on embedded devices. There are already techniques available that correct non-malicious alterations of the processed values in DNN (due to environmental conditions) [29]. However, countermeasures against malicious adversaries still need to be developed, similarly to the area of applied cryptography [9, 37, 15]. As cryptography normally works on byte values and neural networks require floating point values, it is necessary to find adjustments to existing fault protection techniques.

## Acknowledgement

This research is supported in parts by the National Research Foundation, Singapore, under its National Cybersecurity Research & Development Programme / Cyber-Hardware Forensic & Assurance Evaluation R&D Programme (Award: NRF2018NCR-NCR009-0001).
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.

## References


[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] Michel Agoyan, Jean-Max Dutertre, Amir-Pasha Mirbaha, David Naccache, Anne-Lise Ribotta, and Assia Tria. How to flip a bit? In *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*, pages 235–239. IEEE, 2010.

[3] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, pages 105–114. IEEE, 2011.

[4] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[5] Gilles Barthe, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Jean-Christophe Zapalowicz. Synthesis of fault attacks on cryptographic implementations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1016–1027, New York, NY, USA, 2014. ACM.

[6] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual international cryptology conference*, pages 513–525. Springer, 1997.

[7] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

[8] Jakub Breier and Chien-Ning Chen. On determining optimal parameters for testing devices against laser fault attacks. In *2016 International Symposium on Integrated Circuits (ISIC)*, pages 1–4. IEEE, 2016.

[9] Jakub Breier and Xiaolu Hou. Feeding two cats with one bowl: On designing a fault and side-channel resistant software encoding scheme. In *Cryptographers' Track at the RSA Conference*, pages 77–94. Springer, 2017.

[10] Jakub Breier, Dirmanto Jap, and Chien-Ning Chen. Laser profiling for the backside fault attacks: with a practical laser skip instruction attack on aes. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, pages 99–103. ACM, 2015.

[11] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. In *ICLR*, 2018.

[12] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 3–14. ACM, 2017.

[13] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 39–57. IEEE, 2017.

[14] François Chollet et al. Keras, 2015.

[15] Mathieu Ciet and Marc Joye. Practical fault countermeasures for chinese remaindering based rsa. In *Workshop on Fault Diagnosis and Tolerance in Cryptography–FDTC*, volume 5, pages 124–132, 2005.

[16] Agoston E Eiben and Selmar K Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

[17] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[18] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Trrespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762. IEEE, 2020.

[19] Christophe Giraud and Hugues Thiebeauld. A survey on fault attacks. In *Smart Card Research and Advanced Applications VI*, pages 159–176. Springer, 2004.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[21] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *ICLR*, 2015.

[22] Oscar M Guillen, Michael Gruber, and Fabrizio De Santis. Low-cost setup for localized semi-invasive optical fault injection attacks. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 207–222. Springer, 2017.

[23] Warren He, Bo Li, and Dawn Song. Decision boundary analysis of adversarial examples. In *ICLR*, 2018.

[24] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraș. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 497–514, 2019.

[25] Marc Joye and Michael Tunstall. *Fault analysis in cryptography*, volume 40. Springer, 2012.

[26] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N Asokan. Prada: protecting against dnn model stealing attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 512–527. IEEE, 2019.

[27] Chong Hee Kim and Jean-Jacques Quisquater. Faults, injection methods, and fault attacks. *IEEE Design & Test of Computers*, 24(6):544–545, 2007.

[28] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[29] Minjae Lee, Kyuyeon Hwang, and Wonyong Sung. Fault tolerance analysis of digital feed-forward deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 5031–5035. IEEE, 2014.

[30] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. Fault injection attack on deep neural network. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 131–138. IEEE Press, 2017.

[31] Daniel Lowd and Christopher Meek. Adversarial learning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 641–647. ACM, 2005.

[32] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[33] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 77–88. IEEE, 2013.

[34] Amir Moosavie Nia and Karim Mohammadi. A generalized abft technique using a fault tolerant neural network. *Journal of Circuits, Systems, and Computers*, 16(03):337–356, 2007.

[35] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4954–4963, 2019.

[36] Stjepan Picek, Lejla Batina, Pieter Buzing, and Domagoj Jakobovic. Fault injection with a new flavor: Memetic algorithms make a difference. In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*, volume 9064 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2015.

[37] Victor Servant, Nicolas Debande, Houssem Maghrebi, and Julien Bringer. Study of a novel software constant weight implementation. In *International Conference on Smart Card Research and Advanced Applications*, pages 35–48. Springer, 2014.

[38] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[39] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *ICLR*, 2014.

[40] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 601–618, 2016.

[41] Chaowei Xiao, Jun-Yan Zhu, Bo Li, Warren He, Mingyan Liu, and Dawn Song. Spatially transformed adversarial examples. In *ICLR*, 2018.

[42] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *arXiv preprint arXiv:1611.05431*, 2016.

[43] Xinjie Yu and Mitsuo Gen. *Introduction to evolutionary algorithms*. Springer Science & Business Media, 2010.