Experimental Evaluation of Deep Neural Network Resistance Against Fault Injection Attacks

Xiaolu Hou[1]*, Jakub Breier[2], Dirmanto Jap[2], Lei Ma[3], Shivam Bhasin[2] and Yang Liu[2]

[1]Acronis, Singapore
[2]Nanyang Technological University, Singapore
[3] Kyushu University, Japan
ho0001lu@e.ntu.edu.sg,jbreier@jbreier.com,djap@ntu.edu.sg,malei@ait.kyushu-u.ac.jp,sbhasin@ntu.edu.sg,yangliu@ntu.edu.sg

Abstract. Deep learning is becoming a basis of decision making systems in many application domains, such as autonomous vehicles, health systems, etc., where the risk of misclassification can lead to serious consequences. It is necessary to know to which extent are Deep Neural Networks (DNNs) robust against various types of adversarial conditions.

In this paper, we experimentally evaluate DNNs implemented in embedded device by using laser fault injection, a physical attack technique that is mostly used in security and reliability communities to test robustness of various systems. We show practical results on four activation functions, ReLu, softmax, sigmoid, and tanh. Our results point out the misclassification possibilities for DNNs achieved by injecting faults into the hidden layers of the network. We evaluate DNNs by using several different attack strategies to show which are the most efficient in terms of misclassification success rates. Protection techniques against these attacks are also presented. Outcomes of this work should be taken into account when deploying devices running DNNs in environments where malicious attacker could tamper with the environmental parameters that would bring the device into unstable conditions, resulting into faults.

1 Introduction

Connected technologies have became ubiquitous in everyday life. Small, single-purpose devices with sensing and responding capabilities have emerged into what has become known as Internet of things (IoT). Components of IoT are designed to be placed everywhere, allowing easy physical access to potential threats. At the same time, the developments in the area of artificial intelligence (AI) have widened the capabilities of automation in various domains, spreading into all aspects of modern digital society. Out of AI, one of the most promising technologies is deep learning, which tries to simulate the behavior of neurons in a human brain. Deep learning enables to analyze

* This research was done while the author was with NTU, Singapore

This is an extended version of paper *"J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu. Practical Fault Attack on Deep Neural Networks. In 2018 ACM SIGSAC Conference on Computer & Communications Security (CCS), pages 2204–2206. ACM, Oct 2018."*

speech, text, and images to decide about complex tasks [19]. Deployment of deep learning has reached security-critical areas, such as autonomous driving, medical systems, etc., making misjudgement a risk that has to be taken into account.

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 [25]. Fault attacks are active attacks on a given implementation which try to perturb the internal software/hardware computations by external means. 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 [29]. 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 [23]. These commonly used activation functions are: `softmax`, `ReLu`, `sigmoid` and `tanh`. Studying these functions under fault attacks allows to derive 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. The use of laser facilitates a strong attacker model with extensive fault injection capabilities. With the models, derived from practical fault injection, we analyze the susceptibility of DNN against such attacks. The primary goal of the performed attacks is to achieve miss-classification during the testing phase. In the hindsight, the achieved miss-classification can jeopardize the functioning of DNN-based paradigms like smart city.

Extensive studies have been performed on adversarial attacks, that crafts the input data with little perturbation to fool deep learning systems [24,26,46,16,15,14,48]. 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 ($\approx 10\%$) can already present a high risk of misclassification ($\approx 62\%$). Moreover, we discuss potential protection techniques that can be applied to neural network implementations and devices that process DNNs to prevent the successful application of a fault injection attack.

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 overviews potential protection techniques against fault attacks, followed by Section 7 which 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 [37].

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 [49]).

As it was pointed out in [39], 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`[23].

Softmax is normally used as the activation function for output layer. It takes a vector $\mathbf{x}$ as input, ith 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 becomes 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 [6]. 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 [7,8,13]. 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 [22].

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 [32,38]. 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 [35] 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 [36]. 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 μm^2.

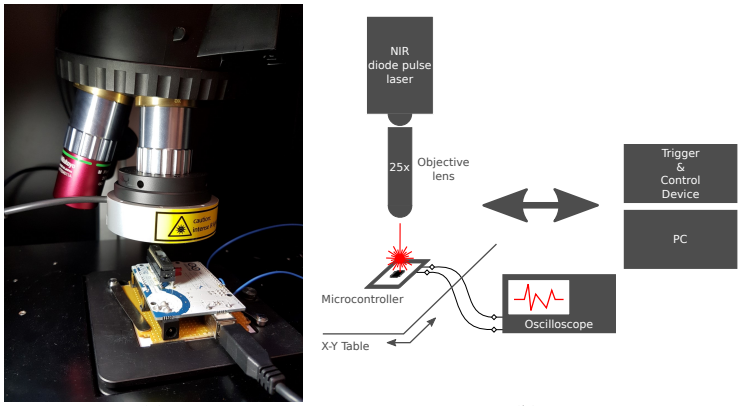
As the device under test (DUT), we used `ATmega328P` microcontroller, mounted on `Arduino UNO` development board. The package of this chip was opened so that there is a direct visibility on a back-side silicon die with a laser. The board was placed on an `XYZ` positioning table with the step precision of 0.05 μm 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.

The chip area is 3×3 mm^2, while the area sensitive to laser is $\approx$ 50×70 μm^2. With a laser power of 4.5% we were able to disturb the algorithm execution, when tested with reference codes.

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 fuction 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.



(a) (b)

Fig. 1: Experimental laser fault injection setup – (a) device under test, (b) setup components.

As instruction skip/change are one of the most basic attacks on microcontrollers, with high repeatability rates [13], 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 $\approx 70\times100\ \mu m^2$. 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 [13] while the sample preparation and guidance on the laser experiments is provided in [9].

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 results on different activation functions in detail.

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

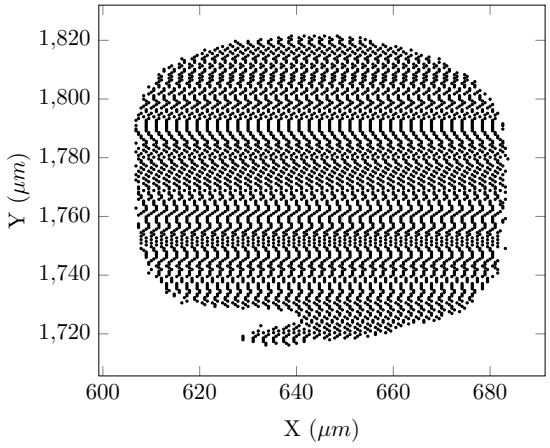


Fig. 2: Area plot depicting successful instruction skip experiments.

```
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:

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

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

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] on 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 full

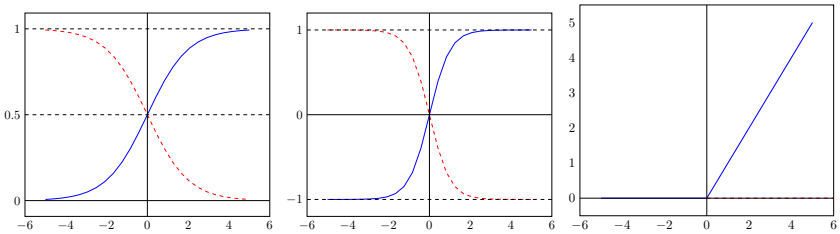


Fig. 3: (a) Sigmoid, (b) Hyperbolic tangent, and (c) ReLu functions. Blue lines indicate original function, red lines indicate faulted ones.

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 ℓ 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 ℓ 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 ℓ, 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 given DNN and a 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 misclassfied 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 ℓ, we aim to find a list of neurons to attack so that the probability of class ℓ 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; ℓ: 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 [33]. The training of DNNs was accomplished using Keras (ver.2.1.6) [17] 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.

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_ℓ: the ℓth 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;
```

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

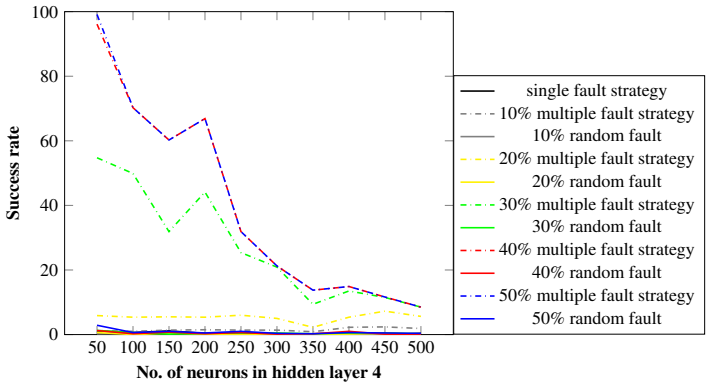


Fig. 4: Target activation function – ReLu.

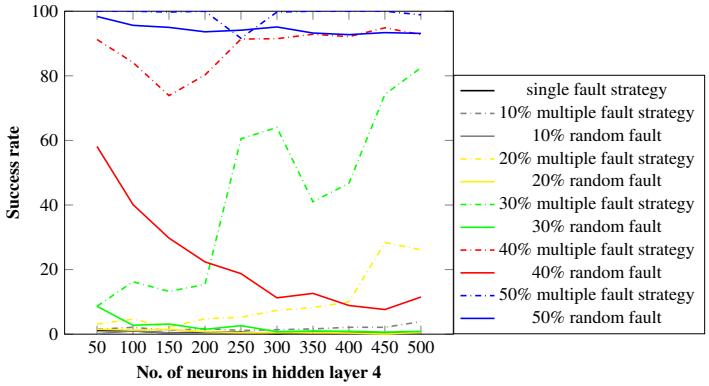


Fig. 5: Target activation function – Sigmoid.

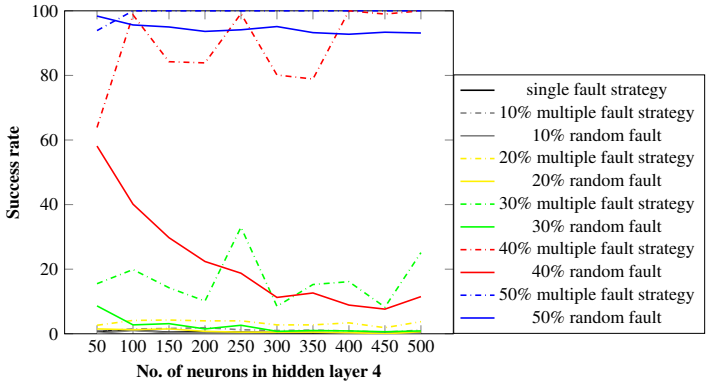


Fig. 6: Target activation function – tanh.

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.

it can help to reduce the search time for finding the solution. While 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 [42].

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 [21] 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

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;
```

follows a standard structure as shown in Algorithm 4. Here we explain how each component of GA was implemented:

- `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 ith neuron is not attacked and a 1 in index j would indicate the jth 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 [20]. 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 [50]. 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 avoid reducing the algorithm to a random search, but significant enough to get a good convergence.

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.

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 Protection Techniques

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

6.1 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. [4]. 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

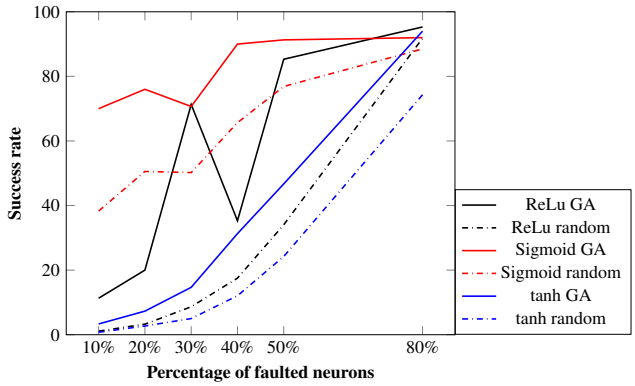


Fig. 7: Evaluation results using genetic algorithm (GA) to select neurons versus random selection.

the chip. In this case, a hardware sensor that checks environmental conditions can be deployed [27,51,43].

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 [30], to sophisticated codes that can be customized to protect against specific fault models [12]. 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 [40], or by re-arranging the data within the instructions to make it hard to tamper with without detection [41]. 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 [47]. such protection technique might be useful in case the learning does not happen in a protected 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 .

6.2 Analysis

Analysis of overheads and coverage of each countermeasure that can be used against instruction skips presented in earlier sections is stated in Table 6. 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

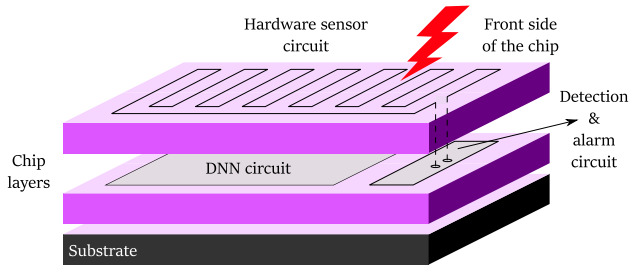


Fig. 8: Hardware sensor protecting the DNN circuit.

countermeasure in cryptography, circuit is either deployed 2-3× on the chip (spatial redundancy), or the computation is repeated 2-3× one after another (temporal redundancy) [5]. 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 [28]. As it was shown, fixed-point arithmetic can provide good results when used on bigger networks [45]. 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 [12]) 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 [12], 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 8. 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. Afterwards, security measures, such as discarding the output, can be applied. Recently, a way to automate the deployment of such circuit was proposed [11].

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.

Countermeasure	Overhead		Coverage
	Time	Area	
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 [12]	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 [31]	–	1.1%[1]	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.

Table 6: Overview of countermeasures effective against skipping instructions.

7 Conclusion and Future Work

In this paper, we have proposed the first physical fault injection attack technique on the major activation functions of deep neural networks. We stated implications how such attack can alter the behavior of targeted network, together with simulations. Our results demonstrate practicality of the attack on ReLu, sigmoid, and tanh.

It will also be interesting to look at possible countermeasures. While there are already techniques available that correct non-malicious alterations of the processed values in DNN (due to environmental conditions) [34], the fault tolerance techniques against malicious entities have to be developed in the same way as in the area of applied cryptography [10,44,18].

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

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: OSDI. vol. 16, pp. 265–283 (2016)
2. Agoyan, M., Dutertre, J.M., Mirbaha, A.P., Naccache, D., Ribotta, A.L., Tria, A.: How to flip a bit? In: On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International. pp. 235–239. IEEE (2010)
3. Balasch, J., Gierlichs, B., Verbauwhede, I.: 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. pp. 105–114. IEEE (2011)
4. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE 94(2), 370–382 (2006)
5. Barenghi, A., Breveglieri, L., Koren, I., Pelosi, G., Regazzoni, F.: Countermeasures against fault attacks on software implemented aes: effectiveness and cost. In: Proceedings of the 5th Workshop on Embedded Systems Security. p. 7. ACM (2010)
6. Barthe, G., Dupressoir, F., Fouque, P.A., Grégoire, B., Zapalowicz, J.C.: Synthesis of fault attacks on cryptographic implementations. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1016–1027. CCS '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2660267.2660304
7. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Annual international cryptology conference. pp. 513–525. Springer (1997)
8. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: International conference on the theory and applications of cryptographic techniques. pp. 37–51. Springer (1997)
9. Breier, J., Chen, C.N.: On determining optimal parameters for testing devices against laser fault attacks. In: 2016 International Symposium on Integrated Circuits (ISIC). pp. 1–4. IEEE (2016)
10. Breier, J., Hou, X.: Feeding two cats with one bowl: On designing a fault and side-channel resistant software encoding scheme. In: Cryptographers' Track at the RSA Conference. pp. 77–94. Springer (2017)
11. Breier, J., Hou, X., Bhasin, S. (eds.): Automated Methods in Cryptographic Fault Analysis. Springer, 1st edn. (Mar 2019)
12. Breier, J., Hou, X., Liu, Y.: On evaluating fault resilient encoding schemes in software. IEEE Transactions on Dependable and Secure Computing (2019)
13. Breier, J., Jap, D., Chen, C.N.: Laser profiling for the back-side fault attacks: with a practical laser skip instruction attack on aes. In: Proceedings of the 1st ACM Workshop on Cyber-Physical System Security. pp. 99–103. ACM (2015)
14. Brendel, W., Rauber, J., Bethge, M.: Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. In: ICLR (2018)
15. Carlini, N., Wagner, D.: Adversarial examples are not easily detected: Bypassing ten detection methods. In: Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security. pp. 3–14. ACM (2017)
16. Carlini, N., Wagner, D.: Towards evaluating the robustness of neural networks. In: Security and Privacy (SP), 2017 IEEE Symposium on. pp. 39–57. IEEE (2017)
17. Chollet, F., et al.: Keras (2015)
18. Ciet, M., Joye, M.: Practical fault countermeasures for chinese remaindering based rsa. In: Workshop on Fault Diagnosis and Tolerance in Cryptography–FDTC. vol. 5, pp. 124–132 (2005)

19. Deshpande, A.: The last 5 years in deep learning. https://adeshpande3.github.io/The-Last-5-Years-in-Deep-Learning (2017), accessed: 2018-11-11
20. Eiben, A.E., Smit, S.K.: Parameter tuning for configuring and analyzing evolutionary algorithms. Swarm and Evolutionary Computation 1(1), 19–31 (2011)
21. Fortin, F.A., De Rainville, F.M., Gardner, M.A., Parizeau, M., Gagné, C.: DEAP: Evolutionary algorithms made easy. Journal of Machine Learning Research 13, 2171–2175 (jul 2012)
22. Giraud, C., Thiebeauld, H.: A survey on fault attacks. In: Smart Card Research and Advanced Applications VI, pp. 159–176. Springer (2004)
23. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), http://www.deeplearningbook.org
24. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. ICLR (2015)
25. Guillen, O.M., Gruber, M., De Santis, F.: Low-cost setup for localized semi-invasive optical fault injection attacks. In: International Workshop on Constructive Side-Channel Analysis and Secure Design. pp. 207–222. Springer (2017)
26. He, W., Li, B., Song, D.: Decision boundary analysis of adversarial examples. In: ICLR (2018)
27. He, W., Breier, J., Bhasin, S., Miura, N., Nagata, M.: Ring oscillator under laser: Potential of pll-based countermeasure against laser fault injection. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2016 Workshop on. pp. 102–113. IEEE (2016)
28. Hwang, K., Sung, W.: Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In: 2014 IEEE Workshop on Signal Processing Systems (SiPS). pp. 1–6. IEEE (2014)
29. Joye, M., Tunstall, M.: Fault analysis in cryptography, vol. 40. Springer (2012)
30. Karri, R., Kuznetsov, G., Goessel, M.: Parity-based concurrent error detection of substitution-permutation network block ciphers. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 113–124. Springer (2003)
31. Khairallah, M., Breier, J., Bhasin, S., Chattopadhyay, A.: Differential fault attack resistant hardware design automation. In: Automated Methods in Cryptographic Fault Analysis, pp. 209–219. Springer (2019)
32. Kim, C.H., Quisquater, J.J.: Faults, injection methods, and fault attacks. IEEE Design & Test of Computers 24(6), 544–545 (2007)
33. LeCun, Y.: The mnist database of handwritten digits. http://yann. lecun. com/exdb/mnist/ (1998)
34. Lee, M., Hwang, K., Sung, W.: Fault tolerance analysis of digital feed-forward deep neural networks. In: Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on. pp. 5031–5035. IEEE (2014)
35. Liu, Y., Wei, L., Luo, B., Xu, Q.: Fault injection attack on deep neural network. In: Proceedings of the 36th International Conference on Computer-Aided Design. pp. 131–138. IEEE Press (2017)
36. Lowd, D., Meek, C.: Adversarial learning. In: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining. pp. 641–647. ACM (2005)
37. Mitchell, T.M.: Machine Learning. McGraw-Hill, Inc., New York, NY, USA, 1 edn. (1997)
38. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on. pp. 77–88. IEEE (2013)
39. Nia, A.M., Mohammadi, K.: A generalized abft technique using a fault tolerant neural network. Journal of Circuits, Systems, and Computers 16(03), 337–356 (2007)
40. Patranabis, S., Chakraborty, A., Mukhopadhyay, D.: Fault tolerant infective countermeasure for aes. Journal of Hardware and Systems Security 1(1), 3–17 (2017)

41. Patrick, C., Yuce, B., Ghalaty, N.F., Schaumont, P.: Lightweight fault attack resistance in software using intra-instruction redundancy. In: International Conference on Selected Areas in Cryptography. pp. 231–244. Springer (2016)
42. Picek, S., Batina, L., Buzing, P., Jakobovic, D.: Fault injection with a new flavor: Memetic algorithms make a difference. In: Mangard, S., Poschmann, A.Y. (eds.) Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers. Lecture Notes in Computer Science, vol. 9064, pp. 159–173. Springer (2015), https://doi.org/10.1007/978-3-319-21476-4_11
43. Ravi, P., Bhasin, S., Breier, J., Chattopadhyay, A.: Ppap and ippap: Pll-based protection against physical attacks. In: 2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). pp. 620–625. IEEE (2018)
44. Servant, V., Debande, N., Maghrebi, H., Bringer, J.: Study of a novel software constant weight implementation. In: International Conference on Smart Card Research and Advanced Applications. pp. 35–48. Springer (2014)
45. Sung, W., Shin, S., Hwang, K.: Resiliency of deep neural networks under quantization. arXiv preprint arXiv:1511.06488 (2015)
46. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. In: ICLR (2014)
47. Taniguchi, Y., Kamiura, N., Hata, Y., Matsui, N.: Activation function manipulation for fault tolerant feedforward neural networks. In: Proceedings Eighth Asian Test Symposium (ATS'99). pp. 203–208. IEEE (1999)
48. Xiao, C., Zhu, J.Y., Li, B., He, W., Liu, M., Song, D.: Spatially transformed adversarial examples. In: ICLR (2018)
49. Xie, S., Girshick, R., Dollár, P., Tu, Z., He, K.: Aggregated residual transformations for deep neural networks. arXiv preprint arXiv:1611.05431 (2016)
50. Yu, X., Gen, M.: Introduction to evolutionary algorithms. Springer Science & Business Media (2010)
51. Zussa, L., Dehbaoui, A., Tobich, K., Dutertre, J.M., Maurine, P., Guillaume-Sage, L., Clediere, J., Tria, A.: Efficiency of a glitch detector against electromagnetic fault injection. In: Proceedings of the conference on Design, Automation & Test in Europe. p. 203. European Design and Automation Association (2014)