NeuroAttack: Undermining Spiking Neural Networks Security through Externally Triggered Bit-Flips

Valerio Venceslai[1,2,*], Alberto Marchisio[1,*], Ihsen Alouani[3], Maurizio Martina[2], Muhammad Shafique[1]
[1]*Technische Universität Wien, Vienna, Austria*
[2]*Politecnico di Torino, Turin, Italy*
[3]*Université Polytechnique Hauts-De-France, Valenciennes, France*
Email: s254591@studenti.polito.it, {alberto.marchisio, muhammad.shafique}@tuwien.ac.at, ihsen.alouani@uphf.fr, maurizio.martina@polito.it

***Abstract*—Due to their proven efficiency, machine-learning systems are deployed in a wide range of complex real-life problems. More specifically, Spiking Neural Networks (SNNs) emerged as a promising solution to the accuracy, resource-utilization, and energy-efficiency challenges in machine-learning systems. While these systems are going mainstream, they have inherent security and reliability issues. In this paper, we propose NeuroAttack, a cross-layer attack that threatens the SNNs integrity by exploiting low-level reliability issues through a high-level attack. Particularly, we trigger a fault-injection based sneaky hardware backdoor through a carefully crafted adversarial input noise. Our results on Deep Neural Networks (DNNs) and SNNs show a serious integrity threat to state-of-the art machine-learning techniques.**

***Index Terms*—Machine Learning, Spiking Neural Networks, Reliability, Adversarial Attacks, Fault-Injection Attacks, Deep Neural Networks, DNN, SNN, Security, Resilience, Cross-Layer.**

I. Introduction

Deep Neural Networks (DNNs) are known to be resilient to numerical perturbations and architectural imprecision [12][27][40][44]. This is demonstrated through an established performance even after aggressive pruning [26], quantization [30], and other compression techniques [10][14], which significantly reduce the number of parameters in the network. However, recent works [11][16][33][34] have shown that these networks are vulnerable to surgical bit-flips in specific locations. Moreover, system-level threats called adversarial attacks [9] have shown effective ability to induce behavioral anomalies in DNNs. In fact, DNNs are vulnerable to malicious inputs modified to yield erroneous labels, while being undetectable to human observers [13][28]. In safety-critical applications such as transportation systems, adversarial examples could be a non-negligible threat to public safety. For this reason, attacks and defenses on adversarial examples have drawn great attention in the scientific community. On the other hand, due to the ubiquity of machine-learning, attacks from the supply chain such as hardware Trojans emerged as a threat to DNNs security. In [24], the authors use fault-injection techniques on SRAM or DRAM to alter the single bit value or few bit values in memory thereby leading to misclassification.

Spiking Neural Networks (SNNs) provide a biologically plausible alternative to DNNs, because the neuron model as well as the event-based communication model between neurons resemble to the current understanding of the human brain's functioning. Compared to DNNs, SNNs show a different response to the adversarial attacks [29]. Moreover, due to their asynchronous and spike-based propagation, the SNNs are naturally more energy-efficient than DNNs when deployed in the hardware, as shown by neuromorphic chips like Intel Loihi [7] and IBM TrueNorth [31].

Towards this, the focus of our paper is to show a new attack vector that threatens the integrity of both the DNNs and SNNs. We propose a cross-layer attack against neural networks that transforms a circuit-level vulnerability to a system-level security flaw. We exploit memory bit-flips in neural networks synapses' weights through a hardware Trojan triggered using a surgical adversarial attack.

To the best of our knowledge, this is first end-to-end attack against SNNs that exploits circuit-level backdoor through a high-level input pattern.

In summary, the contributions of our paper are as follows:

- We analyze the resilience of SNNs to errors.
- We propose a methodology for triggering a bit-flip attack remotely through an adversarial input pattern.
- We introduce ***NeuroAttack***, a hardware Trojan triggered by an input noise. We design and compare different versions of the noise pattern that triggers the Trojan.
- We show the practicality of NeuroAttack on DNNs and SNNs, by converting pre-trained DNNs into the spike domain.

II. Background and Related Work

A. Spiking Neural Networks

Spiking Neural Networks (SNNs) are considered as the 3rd generation neural networks. The previous generations employed continuous values for the output signals of the neurons, whereas SNNs use spike trains to encode the information. Therefore SNNs, for their binary (spiking or no spiking) operation, lend themselves well to fast and energy-efficient implementation on hardware devices [15]. Each incoming signal from an input neuron, which is encoded in the SNN technology as a spike train, is multiplied by the weight of the synapses, and all the results are added together to produce the so-called *membrane potential* V_m, expressed as:

$$V_m = \sum_{i=1}^{N} w_i \cdot s_i,$$

where *N* is the number of input synapses. When the membrane potential reaches a particular value, called *threshold*, the output neuron "spikes", or "fires".

*These authors contributed equally to this work.

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

There are different ways in which the continuous values can be coded as spikes in time domain. The most commonly used are *rate coding* and *time coding*. In the first case, the information is encoded by the number of spikes per second, i.e., an higher number of spikes per second refers to an higher analog value. In this case, the spike rate is determined by the mean rate of a Poisson process [41]. Moreover, the pixels of the images are converted to a constant current entering in the input neurons, so that they will spike at constant rates depending on the input pixel intensity. The *time coding* can be implemented in different ways, for example the *latency coding*, in which the analog value is inversely proportional to the spiking delay of the neuron.

Many different models for the spiking neurons have been studied. These models must be at the same time (1) biologically accurate and capable of producing rich patterns, and (2) computationally simple. The Hodgkin-Huxley biologically-accurate model [2] is computationally expensive, whereas, on the other hand the *Leaky Integrate and Fire* (LIF) model [42] gives the opportunity, for its simplicity, to process lots of neurons in real-time but its biological plausibility is very low compared to the Hodgkin-Huxley's model. Other models have been developed to make a compromise between the two extremes. An example of such a tradeoff is the Izhikevich model [17]. However, we take advantage of the simple LIF model (shown in Figure 1) to explain in the details the working principles of a SNN, as it has been deployed in real-world neuromorphic processors.

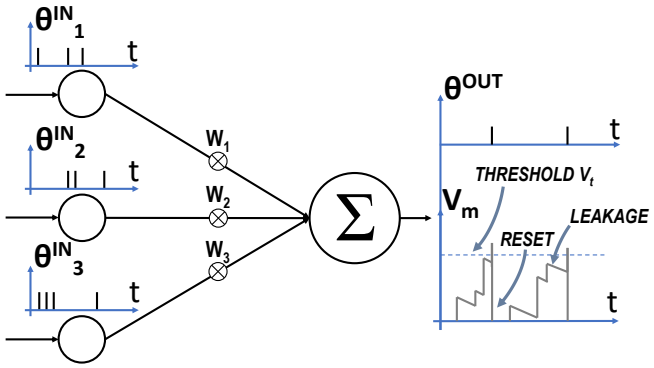

Fig. 1: Input and output spikes, referred to the membrane potential for a simple LIF model.

When a spike inputs the neuron, the associated synaptic weight w_i will be integrated on the membrane. When the membrane potential V_m overcomes a threshold V_t, the neuron fires and resets its membrane potential to a value V_R, which is considered to be zero in Figure 1. In addition, due to leakage, the membrane potential decreases continuously at the leak rate between two input spikes [4]. The sub-threshold dynamics of LIF spiking neuron can be formulated as follows:

$$\tau_m \frac{dV_m}{dt} = -V_m + I(t),$$

where V_m is the membrane potential and τ_m is the time constant for the membrane potential leakage [21]. Local learning rules for unsupervised learning can be used to train the network, as can be done also in the recent Loihi neuromorphic processor [7]. The *Spiking Time Dependent Plasticity* (STDP) local learning rule can be applied. The goal of such a rule is to strengthen the synaptic weight of two neurons whose spiking activity happens in a highly-correlated causal dependency order, and to weaken it otherwise [5]. However, learning through the unsupervised learning rules is found to be effective just for shallow networks [22]. On the contrary, the backpropagation mechanism used to train DNNs cannot be applied as-is, due to the non-differentiabile nature of the spiking function [21]. To overcome this problem two solutions are typically employed: (1) take advantage of an approximate derivative method, or (2) convert offline trained DNNs to SNNs. The first solution has been extensively studied in many works [3][21][23][32]. The second solution is exploited in the following discussions. The neural networks are described as Keras models, trained as DNN and then converted to SNN by means of the SNNtoolbox [39], and implemented by means of spiking neuron's simulators through *rate encoding*. A built-in simulator based on Keras, i.e., INIsim, is used, which features the simple LIF neuron model. The duration of the simulation is set to 50 milliseconds, one millisecond for each time step while the other parameters are left with the default values.

B. Adversarial Attacks

An adversary, using information learnt about the structure of the classifier, tries to craft the perturbations added to the input to cause its misclassification, i.e., its incorrect classification. For explanation purposes, we consider a generic DNN for image classification. Given an original input image x and a target classification model $C(.)$, the problem of generating an adversarial example x^{adv} can be formulated as a constrained optimization [43]:

$$x^{adv} = \arg\min_{x^{adv}} \mathcal{D}(x, x^{adv}), s.t.$$
$$C(x) = l, C(x^{adv}) = l^{adv}, l \neq l^{adv}$$

Where $\mathcal{D}$ is a distance metric used to quantify the similarity between two images, and the goal of the optimization is to minimize the added noise, typically to avoid the detection of the adversarial perturbations. l and l^{adv} are the two labels of x and x^{adv}, respectively. Here, x^{adv} is considered as an adversarial example if and only if the label of the two images are different ($C(x) \neq C(x^{adv})$) and the added noise is bounded ($\mathcal{D}(x, x^{adv}) < \epsilon$ where $\epsilon \geqslant 0$).

C. Fault-Injection

The outputs of a DNN depend on both the input images and its internal parameters. By inserting errors in the internal parameters of a network, it is possible to misclassify a given input image. Since the parameters of the network, when implemented in hardware, are stored in memory units as SRAM or DRAM, with the development of precise memory fault-injection techniques, such as laser beam fault-injection [38] and row hammer attack [18], it is possible to launch effective fault-injection attacks on DNNs [24]. Shattering the accuracy of a DNN in a significant way, with a low amount of faults, is a challenging task. This is due to the high resilience of neural networks which will be analyzed in section III. Towards

this, an efficient fault-injection technique will be used in Section III-B, and it will be shown that few tens of faults (bit-flips), associated to network's internal parameters, are sufficient to cause a considerable reduction of performances. The results of this analysis will be used to build up an efficient attack methodology through the hypothesis of an hardware Trojan insertion in the supply chain plus a well-crafted input Trojan trigger pattern, which can threaten the security properties of both the DNNs and the SNNs. Unlike previous works, our NeuroAttack is a **cross-layer** attack that exploits a hardware backdoor through a carefully crafted adversarial input noise.

III. Bit-flip Resilience Analysis of SNNs

A. Statistical Analysis of Random Bit-Flip

In this section, we analyze the resilience of SNNs to random bit-flips in its internal parameters. Two different networks, whose structures are reported in Table I and Table II, have been chosen.

TABLE I: Structure of the Multilayer Perceptron network.

Layer	Output shape
Input	784
Dense	1200
Dense	1200
Dense	10

TABLE II: Structure of the LeNet network [20].

Layer	Output shape	Output maps	Kernel size	Strides
Input	(28, 28, 1)	-	-	-
Conv2D	(28, 28, 32)	32	(5,5)	(1,1)
MaxPool2D	(14, 14, 32)	-	-	(2,2)
Conv2D	(10, 10, 48)	48	(5,5)	(1,1)
MaxPool2D	(5, 5, 48)	-	-	(2,2)
Dense	256	-	-	-
Dense	84	-	-	-
Dense	10	-	-	-

The first one is the so called *Multilayer Perceptron* (MLP). The perceptron is a basic neuron, which receives as input the signals multiplied by the synaptic weights. These signals are summed together with a bias θ, and a non-linear function is applied [8], as expressed by the following formula:

$$f\left(\sum_{i=1}^{N} x_i \cdot w_i + \theta\right).$$

These neurons are connected in a dense (or fully-connected) fashion, so that each neuron in layer l receives as inputs the outputs of each neuron in the previous layer *l-1*. The amount of synapses and related weights connecting one layer to the previous one is given by $n_{l-1} \cdot n_l$, where n_l is the amount of neurons in a given layer l. For instance, for a simple 4 layer MLP, like the one in Table I, the number of parameters is about 2 millions. This huge amount of parameters is related to an inherent resilience of DNNs to errors or approximations, as it has been studied in prior works [12][36][37].

With *Convolutional Neural Networks* (CNNs), additional types of layers are introduced, i.e., the *convolutional layers* to extract features from the input image and the *pooling layers* to reduce the size of the data. The so-called *feature maps*

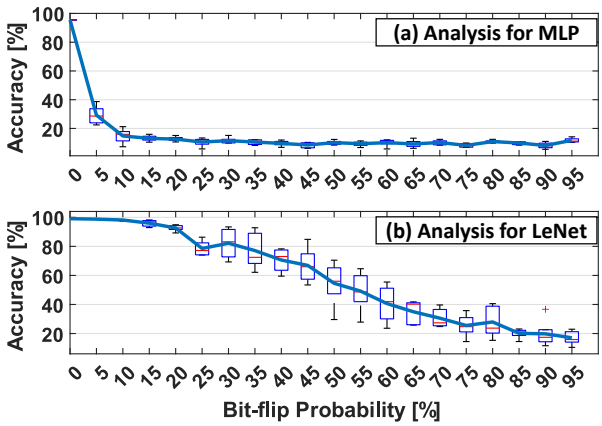

Fig. 2: Accuracy vs bit-flip probability for (a) MLP, and (b) LeNet network.

of the convolutional layers sweep the input image with a certain stride, and have shown excellent capabilities to extract features in the images given as inputs. This trait led to reach an outstanding performance in many image-recognition and classification tasks. One example of CNN is the LeNet-5, whose structure is shown in Table II. It achieves excellent capabilities in classifing images belonging to the MNIST dataset.

The two networks have been trained for 30 epochs to reach the top accuracy of 95.54% and 99.05% on the MNIST dataset for the MLP and the LeNet, respectively. Weights and biases are then quantized to 8 bits. The first investigation is a statistical analysis of both networks. The *bit-flip probability* is set between 0% and 95% to have 20 different points, and it represents the probability for which a weight is subjected to bit-flip. The results are averaged over 5 different iterations. The results of accuracy against the *bit-flip probability* for both the MLP and the LeNet are shown in Figures 2-a and 2-b, respectively.

These results show that in the MLP, the accuracy is reduced significantly also for a low *bit-flip probability*. However, for networks with huge amount of parameters, a higher number of parameters undergo bit-flip also for low values of *bit-flip probability*. The situation is clear looking at Figure 3-a and Figure 3-b which depict the average accuracy (red line, right axis) compared to the average number of bits flipped (blue line, left axis), for MLP and LeNet respectively. The number of bits flipped with the same bit-flip probability appear to be at least one order of magnitude less in the LeNet with respect to the MLP. This analysis shows the high resilience of a neural network whose performance is degraded just for a huge amount of errors in the network parameters. However, these networks, as demonstrated in the following section, are resilient only for probabilistic attacks, while showing very different behavior in case of well-targeted errors that can be applied by an adversary.

B. Bit-Flip with Gradient Search Algorithm

Analysis for the MINIST Dataset: In this section, we describe a way to reduce the accuracy of a network by applying errors on the lowest possible amount of bits. The gradients of the loss function with respect to the parameters of the network are analyzed in a similar way to what is done during the

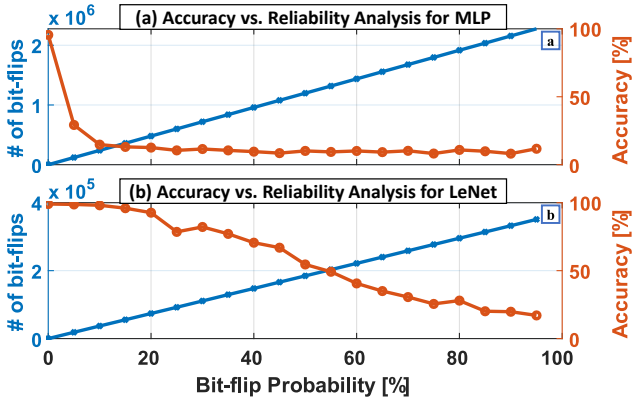

Fig. 3: Accuracy and number of bit-flips vs bit-flip probability for (a) MLP and (b) LeNet network.

learning, while taking an inspiration from the work of [35]. The computation of gradients returns a list of n-dimensional arrays of the same shape of the parameters. The highest gradient in absolute value is taken and the corresponding parameter is considered as the target parameter. One of the bits of the target parameter is flipped to have the maximum reduction of accuracy. The target parameter is then masked, so that it is not considered at the next iteration. The results show that the accuracy is highly reduced for very low number of bit-flips for the MLP (see the blue line in Figure 4) and for the LeNet (see the red line in Figure 4), considering a global analysis of the parameters. Note, only 30 bit-flips are sufficient to completely crush the accuracy of the two considered networks.

Analysis for the CIFAR10 Dataset: Similar experiments have been performed also for the CIFAR10 dataset [19], which is composed of 60,000 training and 10,000 test RGB 32x32 images. The CNN used in our experiments, whose structure is reported in Table III, reaches 79% of accuracy after 50 epochs of training.

TABLE III: CNN structure providing 79% accuracy on CIFAR10.

Layer	Output shape	Output maps	Kernel size	Strides
Input	(32, 32, 3)	-	-	-
Conv2D	(32, 32, 32)	32	(3,3)	(1,1)
Conv2D	(30, 30, 32)	32	(3,3)	(1,1)
MaxPool2D	(15, 15, 32)	-	-	(2,2)
Dropout 0.25	(15, 15, 32)	-	-	-
Conv2D	(15, 15, 64)	64	(3,3)	(1,1)
Conv2D	(13, 13, 64)	64	(3,3)	(1,1)
MaxPool2D	(6, 6, 64)	-	-	(2,2)
Dropout 0.25	(6, 6, 64)	-	-	-
Dense	512	-	-	-
Dropout 0.25	512	-	-	-
Dense	10	-	-	-

The *gradient search algorithm* is applied on all the parameters of the network, and similar results w.r.t. the previous cases are obtained. However, as shown by the orange line in Figure 4, the accuracy drop is far more emphatic. In fact, the accuracy reaches a plateau around 10% for just 4 bit-flips, which is a more critical result than the one obtained with the LeNet and the MLP working on the MNIST dataset.

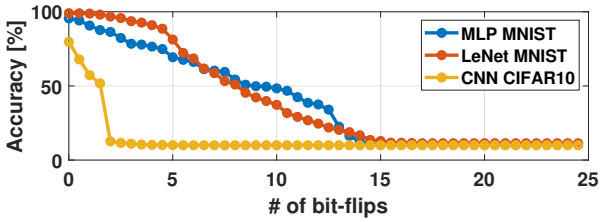

Fig. 4: Accuracy vs number of bit-flips for MLP@MNIST, LeNet@MNIST and CNN@CIFAR10.

IV. NeuroAttack Methodology

A. Threat Model

The attack phase is supposed to be within the supply chain where a malicious actor can insert hardware Trojans. In fact, modern integrated circuit design often involves a number of design houses, fabrication houses, third-party IP, and electronic design automation tools that are all supplied by different vendors. Such a horizontal business model makes the security extremely difficult to manage during the supply chain [1][6]. Moreover, the attack is in a *grey-box* setting, i.e., the attacker has a complete knowledge of the system architecture and internal parameters but is not aware of the training set and training hyperparameters.

B. Hardware Trojan Design

The hardware Trojan is designed to perform fault-injection (i.e., bit-flips) in the network parameters to undermine its integrity and degrade its accuracy. The malicious behavior is triggered from the input through a specifically crafted input noise. The idea is to trigger a fewer number of hardware Trojans hidden in the circuit during the supply chain. Taking advantage of the analysis carried on in Section III-B, hardware stealthy Trojans are inserted at appropriate locations. Each Trojan consists of a 2-way multiplexer with one input which is the original bit, whereas the other input is the complemented bit obtained through an inverter. The multiplexer's selection signal is a signal which is at logic value high only when a trigger is added to the input image. In this way, the network will behave correctly when an untouched input is supplied, providing high accuracy for the original dataset. However, when a trigger is inserted in the input image in form of hidden noise, the fault-injections will be activated, and therefore the accuracy will be degraded significantly. The setting is explained in Figure 5, in which the thick orange arrows represent the synapses with bit-flip applied, and the grey neuron is the target neuron. To produce the selection signal of the multiplexers, the output of a selected neuron is compared against a threshold through a comparator, chosen according to the results of our experiments. Note, the goal is for the output of the neuron to exceed the threshold when the trigger is added to the dataset, and not when the original dataset is given as an input. The first step of the work is to select a particular neuron to satisfy the desired behavior. To transfer the methodology from the DNN to the SNN domain, a counter that accumulates the number of spikes is needed at the input of the comparator. Moreover, the threshold must be transferred from its analog value to the corresponding value of spike rates. The counter is cleared at the end of the processing of each input.

C. Trigger Pattern Design

Since there can be a direct relationship between the analog output value of a neuron and the corresponding spike rate, the knowledge obtained through the analysis of the DNN can be transferred to the SNN implementation. Moreover, a good correlation between analog output value and spike rate is a necessary condition when using the SNN toolbox for DNN-to-SNN conversion. Our goal is to embed the trigger inside one neuron of the network, which we call the *target neuron*. In other words, the goal of our proposed technique is that such a target neuron is activated by a carefully designed mask in the input image.

1) Choosing the target layer: The selection of the target neuron strongly depends on the target layer. In case of a CNN, the choice of the layer is directly connected to the choice of the size of the trigger mask. This is due to the fact that neurons belonging to deeper convolutional layers are related to a larger area of the input image. For example, by looking at Figure 6, the gradients of a neuron belonging to the first and second convolutional layers are reported. The higher the order of the layer is, the larger the area of the image that will account for the trigger. At the first convolutional layer, the shape, position and value of the gradients are quite clear, and corresponds to the *feature map* of the neurons. For neural networks which have only dense layers (e.g., MLPs) the gradients cover the entire image. In this case, if a smaller trigger is desired, a mask that does not comprehend all the area covered by the gradients can be crafted.

2) Choosing the target neuron: The target neuron is chosen as the one with thehighest value among the sum of absolute values of weights connected to the neurons of the previous layer. This is modeled by the following equation:

$$argmax_t(\sum_{i=1}^{N} ABS(W_{layer_{i,t}})).$$

3) Choosing the triggering mask: A *random initial image* is created and the network is inferred with that image, leading to a value $initial_{OUTPUT_k}$ at the output of the target neuron. The parameter $target_{OUTPUT_k}$ is chosen to be much higher than $initial_{OUTPUT_k}$. A cost function is then defined as follows:

$$cost = \frac{\sum_{i=1}^{N} \delta_i^2}{N},$$

Where $\delta_i = target_{OUTPUT_i} - initial_{OUTPUT_i}$, *i* is the index of each neuron in the target layer. Being *k* the index of the target neuron, we rewrite the expression as:

$$cost = \frac{\delta_1^2 + \delta_2^2 + ... + \delta_k^2 + ... + \delta_N^2}{N}$$

For each δ_i it is imposed that $target_{OUTPUT_i} = initial_{OUTPUT_i}$ except for δ_k, where $target_{OUTPUT_k} \neq initial_{OUTPUT_k}$. The derivative of the cost function is computed with respect to the pixels of the random input image, to understand which part of the input image influences the target neuron. Based on this, a mask *M* is created and a *random initial trigger* is generated by the dot product between the mask and the *random initial image*. The mask can also be chosen differently, but it must have some overlap with the gradient matrix, otherwise the loop that has to be described, will not work.

4) Generating the trigger: The trigger generation algorithm (see Algorithm 1) is inspired by the work of Liu et al. on Trojan attacks [25]. In the first rows, some initialization parameters are set. val_{min} and val_{max} are useful to manage the imperceptibility characteristics of the trigger, but should always lay in the range (0,1). The loop proceeds until the cost reaches a particular threshold, or until a maximum number of iterations. The gradients Δ are first calculated and then limited by a mask that can be suited for the gradients (in that case, line 4 of Algorithm 1 can be skipped), or can be decided in another way. Compared to algorithm in [25], line 6 is added to limit the maximum and minimum values for the pixels in the trigger.

Algorithm 1 Trigger generation loop

1: **INIT**$(val_{min}, val_{max}, lr, epc, epochs, th, cost)$
2: **while** *cost* $<$ *th* and *epc* $<$ *epochs* **do**
3: $\quad \Delta = \frac{\partial cost}{\partial x}$
4: $\quad \Delta = \Delta \cdot M$
5: $\quad x = x - lr \cdot \Delta$
6: $\quad x = clip(x, val_{min}, val_{max})$
7: $\quad epc = epc + 1$
8: **return** x;

At the end of the loop, a new trigger is generated with pixels' values optimized to provoke the saturation of the target neuron. If the parameter $target_{value_k}$ is set too high, in general, the target neuron will not reach that value but a lower value, which we call $final_{value_k}$. A *threshold* is chosen, such as that if the neuron's output value exceeds it, the output of the comparator is set to high and the multiplexers are switched. Then, for each targeted weight, the selected bit is complemented. The *threshold* is calculated through the following formula:

$$threshold = final_{value_k} - \xi,$$

where ξ is a parameter, which can be chosen according to the parallelism of the network and the method of the attack.

5) Trigger application: The trigger can be applied on the image in mainly two ways: (1) as a stamp in the image, or (2) as a noise in the image. In the first case, the values of the pixel in the trigger area are exactly the optimal ones as generated by the loop described in the lines 2-7 of Algorithm 1. However, this solution could be less imperceptible, and in that case a careful choice of the layer and/or a careful choice of the trigger mask parameters (position, dimension, max_{val}) should be taken into consideration. The second case could be of a more general interest and it produced good results, due to a better imperceptibility, as it will be shown in the following Section V. Moreover, supposing to have some general knowledge about the pixel intensity distribution on the image dataset targeted by the network, the choice of the trigger parameters can rely also on this information.

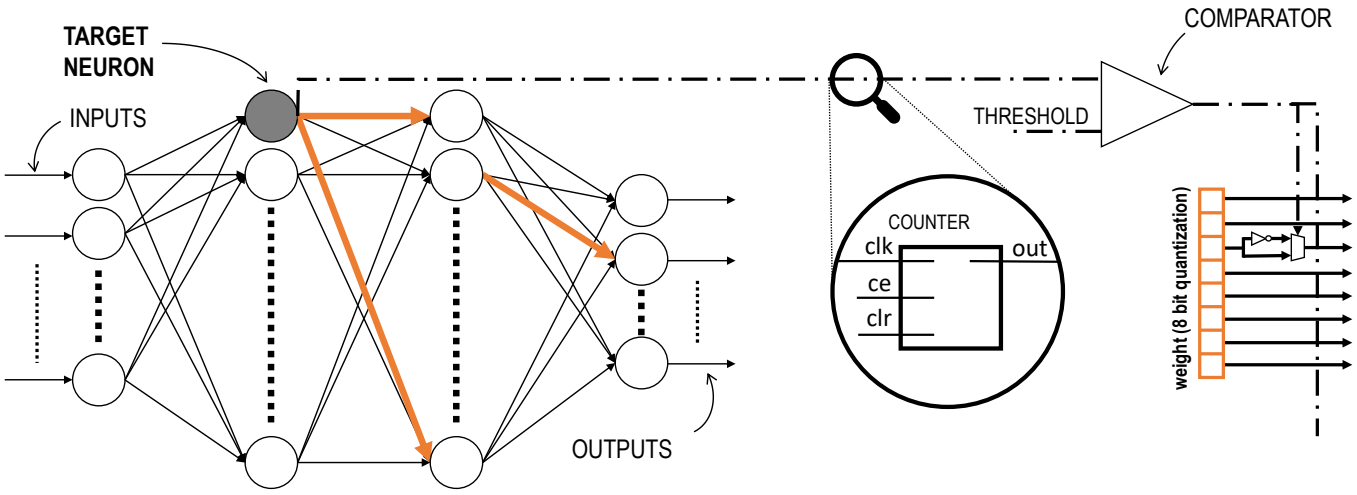

Fig. 5: Scheme of the Trojan attack for the MLP network with the counter added present only in SNN implementation.

V. Results and Discussion

A. Experimental Setup

Both the original and the modified dataset are used for inference, and the amount of times for which both dataset make the target neuron exceed the threshold is recorded. There is the possibility that some images from the original dataset produce the saturation of the neuron, causing an unwanted activation of the Trojans for an $exceed_{ORIGINAL}$ amount of times. However, for a stealthy attack purpose, a carefully crafted trigger should lead to a situation in which this value is kept to almost zero. Therefore, the accuracy is not noticeably reduced when the input trigger is not present, i.e., the presence of hardware Trojans is stealthy. We call $dim_{DATASET}$ the number of images in the dataset, $exceed_{ORIGINAL}$ the number of images from the original dataset for which the threshold for the target neuron is exceeded, and $exceed_{MODIFIED}$ the number of images from the modified dataset for which the threshold for the target neuron is exceeded. Hence, the attack aims at being both effective and stealthy, and thereby to simultaneously satisfy the following conditions:

1) $exceed_{ORIGINAL} << exceed_{MODIFIED}$
2) $exceed_{ORIGINAL} << dim_{DATASET}$
3) $exceed_{MODIFIED} \simeq dim_{DATASET}$

In the following, the results obtained using the MNIST and the CIFAR10 datasets are discussed.

1) Results on the MNIST dataset: Targeting the **first convolutional layer** of the LeNet-5 with parameters listed in the first row of Table IV, the trigger shown in Figures 7 (d) is produced.

Fig. 6: Gradient representation of a random neuron from **(left)** the first and **(right)** the second convolutional layer of the LeNet.

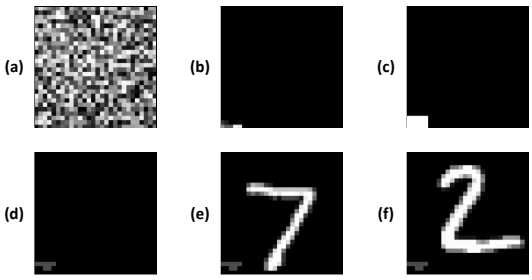

Fig. 7: From top-left to bottom-right: (a) initial input trigger, (b) gradients of the selected neuron, (c) mask created through gradients, (d) final trigger after loop, (e) and (f) two images with applied trigger.

In Figure 7 (a), (b) and (c), the *random initial image*, the initial gradients and the mask *M* are shown respectively. The mask is crafted to follow the shape of the gradients. The images from both the original and the modified test set (two examples from this last image set are shown in Figures 7 (e) and (f)) are inferred and the results, as reported in Table IV: $exceed_{ORIGINAL} = 0$ and $exceed_{MODIFIED} = 10000$.

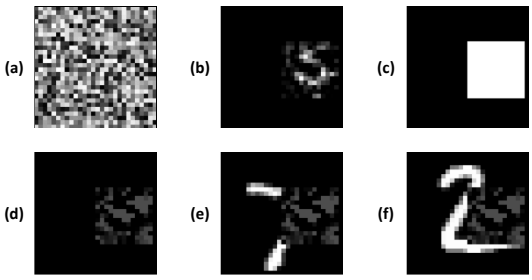

Fig. 8: From top-left to bottom-right: (a) initial input trigger, (b) gradients of the selected neuron, (c) mask created through gradients, (d) final trigger after loop, (e) and (f) two images with applied trigger.

Targeting the **second convolutional layer**, the produced results are significantly different. In fact, the trigger is far more perceptible and superimposed with a significant part of the images, as can be seen in Figure 8. In this case, with the same

experimental settings as explained earlier, the obtained statistics about the threshold exceeding are: $exceed_{ORIGINAL} = 5$ and $exceed_{MODIFIED} = 7585$, as also reported in Table IV. This demonstrates that targeting a neuron belonging to the second convolution layer leads to a relatively worse result. In fact, it can be pointed out that the gradients are, on average, higher than the gradients corresponding to a target neuron belonging to the first convolution layer. We define the correlation between the target neuron and the masked part of the image *S* as follows:

$$S = \frac{\sum_{i,j}^{N} \gamma_{i,j}}{M^2},$$

Where $\gamma_{i,j}$ is the gradient corresponding to the pixel with indexes *i,j* in the trigger mask, and *M* is the size of the side trigger, in case of a square trigger. It can be seen that in the first convolution layer $S = 2.21 \cdot 10^{-5}$, whereas in the second convolution layer $S = 1.4 \cdot 10^{-6}$. This clearly shows that, for a neuron in the 2^{nd} layer, the variation with the input pixel is much lower. If we call ρ the value

$$\rho = exceed_{MODIFIED} - exceed_{ORIGINAL},$$

we can see that it is getting lower when choosing target neurons belonging to deeper layers.

Taking into consideration the MLP, a square mask is created and put in the bottom-right corner. Its side is varied between 5 and 17 pixels, with steps of 2 pixels. Since, at the beginning, the area of the trigger is too small, there are not enough pixels to optimize the saturation of the target neuron. The difference between $initial_{value_k}$ and $final_{value_k}$ results in a small value. Moreover, a huge number of images from the original dataset make the target neuron exceed the threshold, leading to a small value of ρ. A larger area of the trigger, on one hand, increases ρ as can be seen in Figure 9 and, on the other hand, leads to a less stealthy trigger.

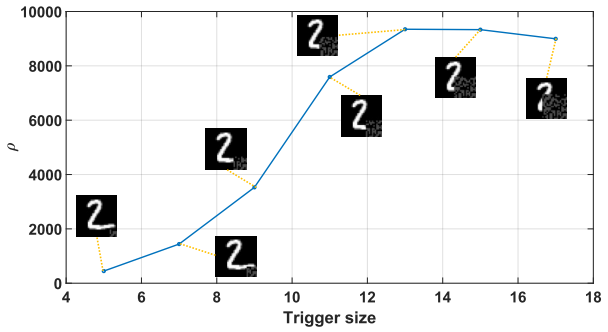

Fig. 9: Plot of ρ with respect to the trigger size.

In the case of the MLP network, an interesting result is obtained with a lower value of $max_{val} = 0.1$. Even though we are targeting the first layer, the gradients are covering the complete image (Figure 10 (b)), since it is a fully-connected layer. Hence, we create a mask suited for the gradient, which spans across the whole image, as shown in Figure 10 (c). In this case, the second method described in Section IV-C5 is used to apply the trigger. Due to the low value of val_{max}, the trigger results to be imperceptible, as shown in Figures 10 (e) and (f). We obtained a very high ρ, shown in Table IV, and high imperceptibility, at the expense of a harder applicability.

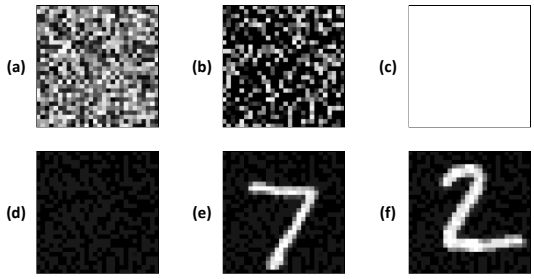

Fig. 10: From top-left to bottom-right (a) initial input trigger, (b) gradients of the selected neuron, (c) mask created through gradients, (d) final trigger after loop, (e) and (f) two images with applied trigger.

2) Results on the CIFAR10 dataset: In this case, targeting the first layer, with parameters set as shown in Table IV, the trigger shown in Figure 11 (d) is produced. The superposition of the trigger on the original images (two examples) is shown in Figures 11 (f) and (h)).

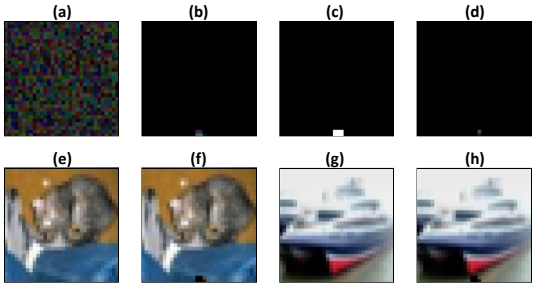

Fig. 11: From top-left to bottom-right: (a) initial input trigger, (b) gradients of the selected neuron, (c) mask created through gradients, (d) final trigger after loop, (e) first image from the dataset (f) first image with trigger applied (g) second image from the dataset (h) second image with trigger applied.

B. Hardware Overhead

Given the amount M of bit-flips applied, the hardware overhead is constituted as the following.

1) M inverters, constituted by 2 transistors each.
2) M 2-way multiplexer, constituted by 16 transistors each in a 4 NANDs implementation.
3) *In the case of a DNN*, a digital comparator, whose complexity depends on the parallelism of the neuron's output result, which is connected to the target neuron's output.
4) *In the case of an SNN*, a counter, to count the spikes, plus a compartor which is set when the counter reaches a particular value.

The overhead of multiplexers and inverters can be estimated as $(2+16) \times M$. From the experiments reported in Section III-B, it is clear that an amount of about just 30 bit-flips is enough to completely crash the performances of the DNN for the two networks operating on MNIST dataset, or 4 bit-flips in the case of the CNN operating on the CIFAR10 dataset. The hardware overhead of inverters and multiplexers, calculated in terms of

TABLE IV: Structure of the networks, parameters and results for our experiments.

Net	Layer	val_{max}	ξ	$target_{OUTPUT_k}$	$initial_{VAL_k}$	$final_{VAL_k}$	$exceed_{ORIGINAL}$	$exceed_{MODIFIED}$
MNIST LeNet	1st Conv2D	0.3	0.1	100	0.04	0.21	0	**10000**
MNIST LeNet	2nd Conv2D	0.3	0.1	100	0.08	1.56	5	7585
MNIST MLP	1st Dense	0.1	0.1	100	0.05	1.21	15	9904
CIFAR10 CNN	1st Conv2D	0.3	0.1	100	0.02	0.23	4	**10000**

transistors, is about $(2 + 16) \times 30 = 540$ in the first case, whereas it is just $(2 + 16) \times 4 = 72$ in the second case. In the case of a SNN, a counter is added, whose module should be at least as much as the maximum spiking rate a neuron can have. The amount of transistors needed for a module N counter are given by $\#_{transistors} = (N-2) \times 6 + (N \times 4) \times 4$, where the first addend gives the contribution of the AND gates, whereas the second gives the contribution of the T-type flip-flops.

VI. Conclusion

In this paper, we propose *NeuroAttack*, a cross-layer attack against DNNs and SNNs, that exploits a circuit-level vulnerability to threaten security. In particular, we demonstrated that NeuroAttack can drastically degrade the accuracy of a DNN or an SNN by applying a few number of bit-flips on its parameters, through a hardware Trojan triggered externally by an adversarial input noise. The security issue is made more severe by the stealthiness of the attack, since it is only effective when triggered by the external adversarial noise, and practically imperceptible elsewhere. Due to the linear relationship between DNN activations and SNN spike rates, the obtained results are transferred to SNN models to corroborate the fact that the demonstrated attack presents a clear threat to both SNNs and DNNs.

Acknowledgments

This work has been partially supported by the Doctoral College Resilient Embedded Systems which is run jointly by TU Wien's Faculty of Informatics and FH-Technikum Wien.

References

[1] I. H. Abbassi et al. Trojanzero: Switching activity-aware design of undetectable hardware trojans with zero power and area footprint. *DATE*, 2018.

[2] D. Beeman. Hodgkin-huxley model. In *Encyclopedia of Computational Neuroscience*, 2013.

[3] S. M. Bohte, J. N. Kok, and H. L. Poutré. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 2002.

[4] M. Bouvier et al. Spiking neural networks hardware implementations and challenges: A survey. 2019.

[5] P. Cardaliaguet and G. Euvrard. Approximation of a function and its derivative with a neural network. *Neural Networks*, 1992.

[6] J. Clements and Y. Lao. Hardware trojan attacks on neural networks. *CoRR*, abs/1806.05768, 2018.

[7] M. Davies et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 2018.

[8] G. J. Gibson, S. Siu, and C. F. N. Cowen. Multilayer perceptron structures applied to adaptive equalisers for data communications. In *ICASSP*, 1989.

[9] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples, 2014.

[10] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *ICLR*, 2016.

[11] M. A. Hanif and M. Shafique. Salvagednn: salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping. *Philosophical Transactions of the Royal Society A*, 2019.

[12] M. A. Hanif, R. Hafiz, and M. Shafique. Error resilience analysis for systematically employing approximate computing in convolutional neural networks. In *DATE*, 2018.

[13] M. A. Hanif et al. Robust machine learning systems: Reliability and security for deep neural networks. *IOLTS*, 2018.

[14] M. A. Hanif et al. X-dnns: Systematic cross-layer approximations for energy-efficient deep neural networks. *J. Low Power Electronics*, 2018.

[15] H. Hazan et al. Bindsnet: A machine learning-oriented spiking neural networks library in python. *Frontiers in Neuroinformatics*, 2018.

[16] L.-H. Hoang, M. A. Hanif, and M. Shafique. Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation. In *DATE*, 2020.

[17] E. M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 2003.

[18] Y. Kim et al. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ISCA*, 2014.

[19] A. Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 2012.

[20] Y. LeCun et al. Gradient-based learning applied to document recognition. 1998.

[21] C. Lee, S. S. Sarwar, and K. Roy. Enabling spike-based backpropagation in state-of-the-art deep neural network architectures. *CoRR*, abs/1903.06379, 2019.

[22] C. Lee et al. Training deep spiking convolutional neural networks with stdp-based unsupervised pre-training followed by supervised fine-tuning. *Frontiers in Neuroscience*, 2018.

[23] J. Lee, T. Delbrück, and M. Pfeiffer. Training deep spiking neural networks using backpropagation. *CoRR*, abs/1608.08782, 2016.

[24] Y. Liu et al. Fault injection attack on deep neural network. In *ICCAD*, 2017.

[25] Y. Liu et al. Trojaning attack on neural networks. 2018.

[26] A. Marchisio, M. A. Hanif, M. Martina, and M. Shafique. Prunet: Class-blind pruning method for deep neural networks. In *IJCNN*, 2018.

[27] A. Marchisio, V. Mrazek, M. A. Hanif, and M. Shafique. Red-cane: A systematic methodology for resilience analysis and design of capsule networks under approximations. In *DATE*, 2020.

[28] A. Marchisio et al. Deep learning for edge computing: Current trends, cross-layer optimizations, and open research challenges. In *ISVLSI*, 2019.

[29] A. Marchisio et al. Is spiking secure? a comparative study on the security vulnerabilities of spiking and deep neural networks. *IJCNN*, 2020.

[30] A. Marchisio et al. Q-capsnets: A specialized framework for quantizing capsule networks. *DAC*, 2020.

[31] P. A. Merolla et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 2014.

[32] E. O. Neftci, H. Mostafa, and F. Zenke. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. *Signal Processing Magazine*, 2019.

[33] M. A. Neggaz, I. Alouani, P. R. Lorenzo, and S. Niar. A reliability study on cnns for critical embedded systems. In *ICCD*, 2018.

[34] M. A. Neggaz, I. Alouani, S. Niar, and F. Kurdahi. Are cnns reliable enough for critical applications? an exploratory study. *IEEE Design Test*, 2019.

[35] A. S. Rakin, Z. He, and D. Fan. Bit-flip attack: Crushing neural network withprogressive bit search. *CoRR*, abs/1903.12269, 2019.

[36] B. Reagen et al. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ISCA*, 2016.

[37] B. Reagen et al. Ares: A framework for quantifying the resilience of deep neural networks. In *DAC*, 2018.

[38] J. Rodriguez et al. Llfi: Lateral laser fault injection attack. In *FDT)*, 2019.

[39] B. Rueckauer et al. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in Neuroscience*.

[40] M. Shafique et al. Robust machine learning systems: Challenges,current trends, perspectives, and the road ahead. *IEEE Design Test*, 2020.

[41] R. Vaila, J. Chiasson, and V. Saxena. Deep convolutional spiking neural networks for image classification. *CoRR*, abs/1903.12272, 2019.

[42] Z. Wang, L. Guo, and M. Adjouadi. A generalized leaky integrate-and-fire neuron model with fast implementation method. *International journal of neural systems*, 2014.

[43] X. Yuan et al. Adversarial examples: Attacks and defenses for deep learning. *CoRR*, abs/1712.07107, 2017.

[44] J. J. Zhang et al. Building robust machine learning systems: Current progress, research challenges, and opportunities. *DAC*, 2019.