Universiteit Leiden

Master Computer Science

Exploring fault injection attacks against embedded neural networks

Name: Shuaizhen Ren
Student ID: s2876655

Date : 18/02/2024

Specialisation: Computer Science: Artificial Intelligence

Company: SGS Brightsight

1st supervisor: Guilherme Perin
Company supervisor : Lichao Wu
Company supervisor : Mathieu Dumont
2nd supervisor : Nele Mentens

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Deep neural networks can provide highly accurate classifications with limited resources, which makes their applications convenient in embedded electronic devices. Unfortunately, these devices are vulnerable to several hardware attack techniques that aim to intentionally affect their functionalities, such as adversarial attacks, or to recover private intellectual property like model parameters and hyperparameters. Among these attacks, fault injection poses serious concerns for the security of deep neural network models as state-of-the-art research has demonstrated the high capability of fault injection to affect deep neural network model accuracy. Moreover, as attack models become more sophisticated, recovering model parameters through fault injection is also an increasing concern.

This work explores and studies the robustness and security of the multi-layer perceptron against simulated fault injection attacks. First, we perform model characterization of the different neural networks against fault injection. Second, we evaluate the possibility of recovering model parameters using the stuck-at-zero fault model. Our results indicate that the number of trainable parameters and activation functions impact the robustness, and the attacker can get certain leakage information through fault injection.

Contents

1 Introduction

In recent years, due to the continuous development of AI technology, people's lives have gradually become closely inseparable from AI innovations. Consequently, there is an increasing need to deploy neural networks on embedded computing systems. Neural networks are of paramount importance and are applied in various fields, such as computer vision [19], image analysis [1], medical treatment [35], autonomous driving [2], and even security domains [45]. Due to the wide range of applications, the security of deep neural networks has become increasingly critical. Concerning user privacy and security, the robustness of neural networks cannot be disregarded. When deployed on embedded devices, neural networks may exhibit several vulnerabilities to adversarial attacks and model stealing. To thwart these malicious attacks, it is crucial to conduct security assessments of neural networks and characterize their primary vulnerable aspects.

Running on embedded devices, deep neural networks may offer several serious vulnerabilities. Nowadays, it is well-known that deep neural networks are highly vulnerable against adversarial attacks [48]. These attacks implement input data modifications to intentionally affect the accuracy of a model and achieve peculiar goals, such as system malfunction, which can lead to safety and security concerns. Another big concern is model stealing [36]. Manufacturers invest significant resources to train neural network models and want to prevent that model's parameters, hyperparameters, and training data leak to malicious players.

With the development of hardware computing systems and the current research on neural network compression technology, neural networks can also be configured on various microcontrollers, such as Arduino, STM32, etc. At the same time, with the development of the Internet of Things(IoT), attackers can more easily attack computing components through physical approaches [6]. And the security of these mobile computing platforms is indispensable.

The thesis considers simulated fault attack models for two main goals. First, we consider fault models for characterizing neural networks to identify the most vulnerable parts of a model. Second, by using a well-defined simulated fault model, we theoretically evaluate the possibility of reversing engineering neural networks. Our analysis focuses on small multi-layer perceptrons (MLPs), and we train these models with different datasets. This thesis has the following main contributions:

- We train several different neural networks on different datasets, and we inject faults into the weights of the pre-trained models. We would consider

several datasets, iris, WDBC, wine and Spine, we would pre-train neural networks on these datasets and conduct fault injection attack experiments on these pre-trained models. Our experiments and analysis are divided into two parts.

- The first step is to characterize the injection errors of the pre-trained neural network model and obtain the faults distribution by injecting different types of faults, thereby analyzing which parts of the neural network are more sensitive to faults and which layers or neurons are more vulnerable to faults. In this process, we would also analyze what kind of neural network structure is more resistant to fault injection errors. Thus we can propose countermeasures to reduce the impact of neural network failures.
- The second part of the experiment is based on the above characterization and error distribution. We would use stuck-at-zero to inject the fault model, restore all the bits of each weight by analyzing the changes in model accuracy before and after each attack, and finally analyze the entire The extracted model is analyzed for accuracy and loss values to determine what extent the model can be restored in this way.

1.1 Problem definition

This work explores the following research questions:

1. For a white-box threat model, in the situation that the attacker knows everything except the training data, what is the effectiveness of fault attacks against deep neural networks?
2. What components of a neural network model demonstrate the highest resilience and susceptibility to fault attacks?
3. In a threat model in which the adversary has access to the model's hyperparameters and is based on an optimistic fault model, to what extent can model parameters (i.e., weights) be recovered?
4. In the previous fault model for the model's parameters stealing, can all bits be restored, and what is the accuracy of the recovered models?

1.2 Structure of the thesis

Section 2 provides the relevant knowledge background used in the thesis, including machine learning security issues, model extraction, and fault injection. Section

3 describes related works on fault injection against deep neural networks. In Section 4, the details of our approach are demonstrated, including simulated fault models for model characterization and model parameters extraction. Simulations experiment results are shown in Section 5. The discussion of our results is given in Section 6. Finally, in Section 7, we draw a conclusion of our analyses and describe prospects for future works.

2 Background

In this section, we elaborate on the background of our work. We start with the main concepts of multi-layer perceptrons and with a review of relevant topics on the security of AI. In this scope, we also briefly discuss the main threat models and how they impact the decisions for attacks and countermeasures. A description of fault injection attacks and their impact on deep neural networks is also provided. Finally, this section describes the datasets used in this work.

2.1 Deep neural networks

2.1.1 Multi-Layer Perceptrons

This work focuses on multi-layer perceptrons (MLPs) as the target architecture type. Thus, in this section, we review the main concepts of this type of neural network.

MLPs are artificial neural networks that have the characteristics of a forward-structured directed graph. The example diagram of the MLP is shown in Figure 1. These models consist of an input layer, one or more hidden layers and an output layer. Each layer has a certain number of neurons. The neurons of each layer l are fully-connected to all neurons in the subsequent layer $l+1$. The connections are defined by a weight value $\omega_{i,j}$ that connects the neuron i-th neuron from layer l to the j-th neuron from layer $l+1$ [38]. The output of a neuron, x_j^{l+1} is computed from the weighted summation of its inputs x_i using the weights $\omega_{i,j}$ associated to these inputs:

$$x_j^{l+1} = \sum_{i=1}^{L} x_i^l . \omega_{i,j} + b_j^{l+1} \tag{1}$$

where b_j^{l+1} is the bias of the neuron j in the layer $l+1$ and L is the size of layer l.

Each neuron is a node with a non-linear activation function. Different activation functions also have different effects on the training of neural network models and have different responses to attacks. Commonly used activation functions include sigmoid, ReLU, identity, and tanh. The equations and images of these activation functions are shown in Figure 2.

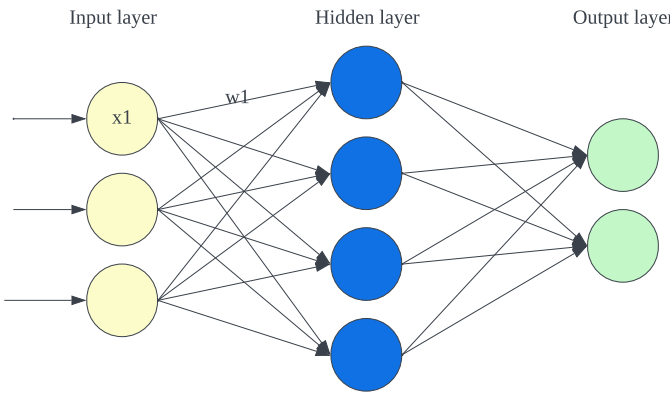

Figure 1: An example diagram of a MLP, which contains the input layer, the hidden layer, and the output layer [25].

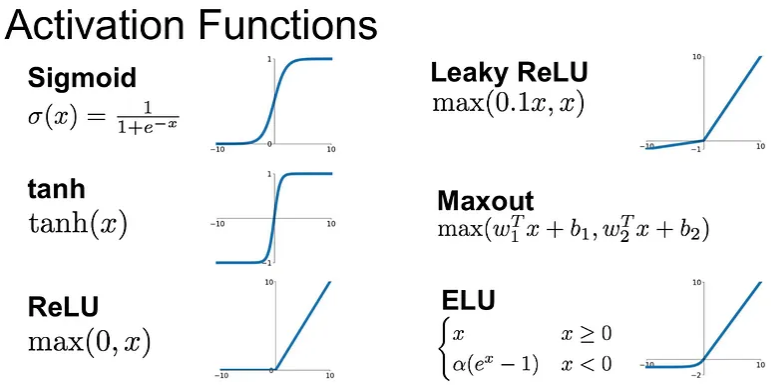

Figure 2: Different activation functions in neural network models [24].

The training process of a MLP involves several key steps aimed at optimizing the network's parameters to accurately map inputs to desired outputs. Initially, the weights and biases of the MLP are randomly initialized. Then, during each iteration of training, a batch of input-output pairs is fed forward through the network, producing predictions. The difference between these predictions and the actual target outputs is measured using a loss function, such as mean squared error (MSE) or cross-entropy loss. The type of loss function depends on the underlying task that the MLP receives. In regression-based tasks, the MSE loss function is commonly employed. For classification tasks, categorical cross-entropy is usually considered and the output layer is set with *softmax* activation function, which maps the output activations from precedent layer into class probabilities.

The backpropagation method is employed to compute the gradients of the

loss function with respect to each parameter in the network. These gradients indicate the direction and magnitude of adjustments needed to minimize the loss. The optimizer algorithm, like stochastic gradient descent (SGD) or Adam, then updates the weights and biases in the network, minimizing the loss. This process iterates over the training dataset multiple times (epochs), gradually refining the model's parameters until the desired level of performance is achieved, or until a stopping criterion is met.

Regularization Regularization is a method in machine learning to prevent deep neural network models from overfitting during the training process. Overfitting typically happens when the model memorizes the training data too well while it demonstrates very poor performance to the validation set. Regularization prevents the model from becoming too complex by employing a penalty mechanism for the weight updates in the loss calculation. Commonly used regularization methods include L1, L2, weight decay and dropout [8]. The L1 and L2 methods separately utilize the sum and square sum of the absolute values of the weights in the loss function, which are utilized to prevent the weights from being too large [16]. Dropout refers to the method of randomly throwing away the output of a certain proportion of neurons in each epoch of the training process to make the model more versatile [41].

In this work, to analyze if the regularization has impacts on DNN models' resistance against fault injection, we also experiment on models with L2 and dropout layers and compare the results with the models without regularization.

2.1.2 Embedded deep neural networks

Embedded deep neural networks (DNNs) enable complex computations to be performed directly on embedded systems with limited computational resources. These networks are carefully designed to find a delicate balance between accuracy and efficiency, making them suitable for deployment in various real-world applications, including robotics, Internet of Things (IoT) devices, and autonomous vehicles. Unlike traditional DNN architectures that rely on high-performance computing resources, embedded DNNs are tailored to operate within the constraints of embedded platforms, optimizing model size, computational complexity, and power consumption while maintaining impressive performance levels [43].

One of the key challenges in developing embedded DNNs lies in achieving the desired level of functionality within the resource constraints imposed by embedded

systems. Researchers and engineers continually explore innovative techniques such as model compression, quantization, and specialized hardware accelerators to enhance the efficiency of embedded DNNs. Moreover, the deployment of embedded DNNs introduces unique considerations regarding real-time inference, robustness to environmental variations, and energy efficiency, further driving advancements in algorithmic and architectural design. As embedded systems become increasingly pervasive across diverse domains, the evolution of embedded DNNs promises to revolutionize the landscape of intelligent edge computing, empowering devices to exhibit sophisticated cognitive capabilities in a compact and energy-efficient manner.

Several techniques enable DNNs to fit within embedded systems. Model compression techniques, such as pruning [28], quantization [23], and knowledge distillation [17], reduce the size of DNNs by eliminating redundant parameters or representing them with fewer bits. As we discuss in the next sections, compressed DNNs also become attractive targets for attackers regarding model performance and model stealing.

2.2 AI security

AI-based systems can exhibit numerous vulnerabilities if their security is neglected. With the rapid evolution of AI across various domains in recent years, a plethora of attacks targeting deep neural networks have been showcased to be viable [14]. Among these threats, adversarial attacks stand out as perhaps the most prominent, aiming to undermine the accuracy of models by manipulating input data [10].

To establish robust defenses against strong adversaries, it's essential to comprehend the workings of attacks and the primary elements within deep neural networks that facilitate effective attack outcomes. Risks are categorized into environmental exposure, data manipulation, model training, and usage, as well as the threat of producing misleading results [48]. For instance, in [9], Carlini *et al* proposed a methodology for the best practices in deep neural network security evaluations against adversarial attacks. Given the high complexity of such attacks and also the fast evolution of this field, defining efficient protections that cover all types of attacks methods becomes nearly impossible. Even state-of-the-art AI models such as ChatGPT can be highly vulnerable to adversarial attacks. For instance, in [32], authors demonstrated how simple manipulation of input data

can extract blocks of training data, exposing confidential information from the model manufacturer and from private users.

Attacks on environmental exposures. Environmental exposures include backend software, docker attacks, hardware and supply chain attacks, etc. Backend software attacks refer to attacks that rely on Python libraries and backend structures. The architecture and application of neural networks rely on many machine learning frameworks, such as TensorFlow and PyTorch, hundreds of computing frameworks, and third-party dependent libraries, such as NumPy, SciPy, etc. However, people have found some security vulnerabilities in the libraries and the machine learning systems that rely on these libraries, for instance, the attacks on pointers, arrays, servers, etc. [46]. Docker attack refers to the attack on important computing nodes of the neural network through the KubeFlow library. The attacker deploys the container by adding malicious additional Python code. Hardware Trojan attacks are attacks that make small changes to the lookup table of a machine learning model that has been trained and deployed on hardware without being discovered by humans [29]. Supply chain attacks involve the attackers providing malicious models on open-source platforms, and the poisoned models are downloaded by consumers, leading to malicious backdoor vulnerability on the consumers' computers.

Attacks on datasets. This category includes malicious data poisoning and data backdoor attacks. Malicious data poisoning refers to feeding the model with contaminated data during training [11]. There are two situations. In the first situation, the attacker makes the model perform normally during training. Still, when the pre-training process is completed, the model incorrectly classified specific categories. The second case is that the attacker gives more poisoned training data, so the model performs poorly during the training process and is difficult to converge [40]. Data poisoning in this process also includes clean data poisoning and wrong data poisoning. The former refers to adding errors difficult for humans to detect in the data. These errors are relatively hidden, and the trainer treats these data as normal data. The latter refers to introducing wrong labels into the training set, which has low concealment and is easy for humans to discover. However, this method has a higher attack success rate. Data poisoning is also called adversarial instance attack, which we will introduce in detail in section 2.3. At present, security attacks on data-level machine learning models

are relatively mature. Corresponding countermeasures include increasing vigilance against data from unknown sources and conducting adversarial detection and analysis before using datasets to prevent the use of poisonous data that is difficult for humans to identify during training. Backdoor attacks are currently a relatively novel method of attacking neural networks. The machine learning model has a backdoor installed by the attacker in the neural network so that the attacked model behaves normally when the backdoor is not activated [11]. However, when the attacker activates the backdoor, the infected model would output specified malicious results.

2.3 Adversarial Attacks for Model Parameters Manipulation

An increasing number of methods for assessing the security robustness of neural networks has emerged [32], notably including the study of adversarial examples. Adversarial examples are crafted input data designed to perturb models during training, inducing misclassifications and vulnerabilities in deep neural network (DNN) models [48].

In the work of Madry *et al*, adversarial examples are viewed as manifestations of inherent instability within neural networks, prompting efforts to enhance the resilience of these networks against such attacks [30]. Illustrated in Figure 3, their approach involves introducing noise into the original image dataset, causing instances initially labeled as cats to be misclassified as dogs by DNN models. Despite the introduction of noise, the alterations to pixel values remain imperceptible to human vision, indicating the powerful capability of adversarial attacks.

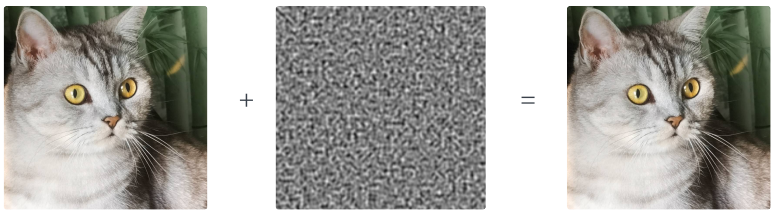

Figure 3: Adversarial attack examples.

Nowadays, while the technology for attacking deep neural network models and implementing countermeasures has significantly evolved, attacks aiming at

recovering and manipulating model parameters still has several limitations. This type of attack scenario is more feasible when attacker can manipulate neural network parameters in memory systems, such as embedded devices. Rowhammer attacks are examples of adversarial attacks DNNs[42] by manipulating DRAM memories. Additionally, neural networks are deployed on high-performance physical devices, such as CPUs and GPUs. However, the landscape has evolved, with embedded neural network models increasingly integrated into computing systems, often employing compression techniques like pruning and quantization [49]. These embedded systems, exemplified by platforms like ARM-based systems or Arduino for low-power applications (e.g., Internet-of-Things), often feature limited storage and memory capacities, rendering them less resilient compared to their high-performance counterparts like CPUs and GPUs [3].

In contrast to the well-established strategies for safeguarding data on CPUs and GPUs, such as hardware performance counters (HPC) utilized for debugging programs and analyzing security vulnerabilities in hardware data [21], embedded systems lack comparable robust security measures. This disparity indicates the need for novel strategies to improve the security and stability of embedded neural network deployments, particularly as these systems become increasingly applicable in diverse computing environments. However, the mobile platforms might lack data integrity checking mechanisms due to limited resources, leading to vulnerability to fault injection attacks [37]. This vulnerability of DNN models might result in serious consequences, mainly due to the misclassification of input data due to affected parameters during model inference.

This work focus on vulnerabilities that may arise in embedded neural networks when fault injection attacks are considered. Our simulated fault model is described in Section 4.4.1.

2.4 Attacks for model parameters extraction

Model extraction involves the retrieval of critical information from the model, such as its trainable parameters and structure. This tactic is often utilized when the attacker can access the model's input, output, or even the underlying dataset [18]. There are different types of model extraction attacks with different goals [34]. In some cases, the main goal of an adversary is to recover specific hyperparameters such as regularization values in hidden layers. Other scenatios [44]. Other attacks attempt to recover the information about the model architecture [33]. Other tech-

niques attempt to extract the trained parameters of a model, even with the usage of physical side-channel information [47].

In our work, we combine model extraction with fault injection to extract and recover the weights of trained MLP models. As will described in Section 4.4.2, our simulated fault model allows an adversary to partially extract model weights from an MLP. This partial information allows the adversary to initialize the model and fine-tune it to achieve similar or better performance than the original model.

2.5 Fault Injection Attack

Fault injection attacks are a form of security breach in which an attacker deliberately introduces faults or errors into a system's hardware or software to compromise its integrity or confidentiality [39]. These attacks exploit vulnerabilities in the system's design or implementation, taking advantage of unexpected behaviors triggered by injected faults. The primary goal of fault injection attacks is to disrupt the normal operation of a system, potentially leading to unauthorized access, data leakage, or system failure.

One common method of fault injection is through physical means, such as manipulating power sources, temperature, or electromagnetic fields to induce errors in hardware components. For example, an attacker might disrupt the power supply to a device or expose it to extreme temperatures to cause malfunctions. Another approach involves exploiting software vulnerabilities to inject faults directly into the code execution process. This could include exploiting buffer overflows to, for instance, inject malicious code.

Physical methods include voltage, electromagnetic manipulation, or laser methods. For voltage, there is voltage manipulation (VM), which is the approach in which faults are induced by manipulating the voltage of the hardware devices; the attackers can steal information. Also, body bias injection (BBI) focuses on the threshold voltage of semiconductor components, and the attackers induce errors by manipulating the minimum voltage. Meanwhile, the attackers can also get leaky information by modifying electromagnetic radiation. In addition, laser attack is also a relatively simple and efficient method.

In the last two decades, fault injection has been widely explored against cryptographic systems and secure bootloaders in embedded devices. Given that computing systems without any protection can be easily compromised by the injection of malicious faults, different solutions based fault tolerant systems have been pro-

posed, such as redundancy, error checking and error correction codes. Because deep neural networks started to be highly deployed on embedded systems, these type of implementations also became quite vulnerable to fault attacks [27]. Because the output predictions deep neural networks are highly dependent on almost all model weights, affecting some weights with fault injection can already completely compromise the model's accuracy. In this work, we explore simulated fault models as a way to characterize the most vulnerable elements of neural network. Moreover, we also explore how fault injection can be used for model extraction.

2.6 Datasets

In our experiments, we trained MLP models on four different datasets, which are widely used in the machine learning field. In the first stage of our experiments, we use Spine and WDBC datasets for characterization. In the second part of our experiments, we consider MLP models trained with Iris and Wine datasets for model weights recovery. The example instances of iris and Spine are shown in Figures 4 and 5, respectively. These four datasets are all available from the Python library *scikit-learn*, which is open-source library that includes several machine learning methods. Our MLP models are also trained with the classes and function provided by this library.

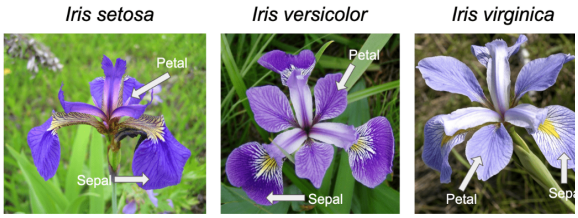

Figure 4: Iris dataset

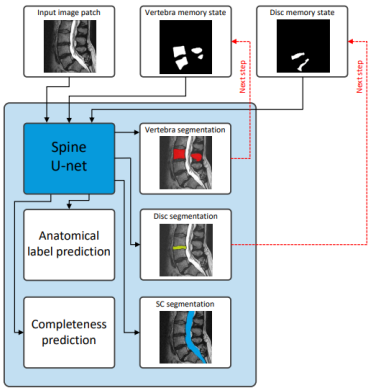

Figure 5: Spine dataset

The WDBC (Breast Cancer Wisconsin Diagnostic) dataset [4] contains the characteristics of sampled cell nucleus images, which is possible to determine whether the tumor is benign or malignant. There are 569 instances in the WDBC dataset, and 31 features to describe each instance. The Spine dataset is collected from information on several patients with spinal diseases. In this dataset, 11

attributes are used to demonstrate the Spine information of 310 patients. These instances would be classified into three categories: lumbar disc herniation, spondylolisthesis, and normal. The first two of them are also classified as abnormal. The iris dataset is also a commonly used classification dataset. It contains 3 types of iris flowers, with 50 examples in each category. The wine dataset includes 178 instances of three different classes. These instances are described by 13 features, including alcohol type, content of organic and inorganic substances such as anthocyanins, etc. We considered these dataset which contain small amounts of training data to make the simulated process faster either for model training or inference.

3 Related work

As mentioned in the last section, several types of AI security threats exist. The following research has been conducted on environmental exposure attacks. In the work of Liu et al. [29], they conducted research on commonly used neural network frameworks, including TensorFlow, PyTorch, etc., and discovered security vulnerabilities in them, including causing the framework service to crash, program hangs, data stream hijacking, etc., affecting images and speech. Identification has a greater impact. In the work of Clements et al. [13], they explored the supply chain security threats posed by neural network hardware Trojans. It causes huge damage to the neural network by introducing small Trojan errors that cannot be found on the testset [3].

In current attacks on neural networks, researchers are still focusing on the malicious manipulation of data. This includes data poisoning and backdoor attacks on data. As introduced in the previous section. Taking into account white-box attacks, the attacker even only needs one poisoning instance to cause the model to misclassify a specific class [40]. For black-box models, attackers can also use multi-model integration methods to increase the proportion of contaminated data to cause misclassification with a higher success rate [50]. An attacker can make the testset data be classified as the result specified by the attacker by only modifying a few training set images. At the same time, another method targets data. The attacker conducts more malicious injections and manipulations of the training data, reducing the accuracy of the training set during the training process and thus greatly reducing the convergence speed. Attackers use this method to damage the performance of some traditional machine learning models, such as SVM [5], logistic regression [31], etc.

However, the above methods are all aimed at the environment and data level during the neural network training period, and the research on malicious data attacking neural networks has been relatively mature. There are very few attacks on neural network structures. In Rakin et al. [37], a method called Bit-Flip Attack (BFA) is used to attack deep neural networks; in the case of limited storage space, malicious bit-flipping is performed on the neural network to destroy the model, and the number of flipping bits is as small as possible. Clavier et al., [12], defined ineffective fault injection, which can be used to recover and detect intermediate values by using the XOR instruction to determine whether the detected value was 0 before being attacked. In the work of Dobraunig et al., the definition of

statistics ineffective fault attack is given, which means ineffective fault attack with the probability of successful attack [15]. In the work of Breier et al. [6], they attack mainly by injecting faults into different activation functions on the hidden layers of neural networks. In the work of Breier et al. [7], they improved the accuracy of the restored neural network model by performing bit-flip attacks on the signal-bit. However, so far, there has been almost no overall characterization of neural networks through fault injection, including analyzing how the structure of a neural network can be more resistant to fault injection attacks and how an attacker can quickly destroy a neural network. Therefore, we make more exploration related to these issues.

4 Simulated Fault Injection on MLPs

In this section, we elaborate and discuss specific fault models using simulated experiments. We start by defining a threat model to clarify the assumptions about the adversary's capabilities. Next, we discuss the fault models and methods of weights encoding to represent weights in binary form.

In our experiments, small MLP models are trained with datasets described in Section 2.6 and then we conduct different attacks based on two fault models: bit-flip and stuck-at-zero models, as described in Section 4.2. The detailed simulation methods with the two models are demonstrated in Section 4.4.

4.1 Threat Model

In order to thoroughly examine how well MLP models stands up to fault injection, it's crucial to examine scenarios from the perspectives of both potential attackers seeking to compromise the model and the legitimate users who train and utilize it. Throughout our research, we'll distinguish between these two parties as the "attacker" and the "model user" respectively, for clarity. By understanding what information and capabilities are available to potential attackers, we can refine our strategies for optimizing attack methodologies and outcomes. Ultimately, this understanding allows model users to implement effective countermeasures to safeguard against such attacks.

In the experimental characterization phase, we utilize the white-box attack model. The objective is to assess the model's resilience against fault injection attacks. In this context, the attacker is presumed to possess comprehensive knowledge about the models, including their structure and parameters.

In the subsequent phase of our experiments, we focus on black-box models for weight recovery. In this scenario, the attacker is aware of the structure of the targeted model, including details like the number of layers and neurons within the neural network. However, crucial parameters of the MLP, such as weights and biases, remain undisclosed. Additionally, we assume that the attacker has access to the test dataset but lacks the capability to acquire the original training data utilized in the neural network's pre-training. Despite this limitation, the attacker can still evaluate the model's performance on the test set to gauge its accuracy.

In the subsequent simulation experiments, we operate under the assumption that the attacker possesses the capability to manipulate all bits of the weights being targeted. This manipulation involves the conversion of bits from 0 to 1 or

from 1 to 0, and it can be executed on a hardware device, as described by Hector et al [20]. By systematically altering different bits of the weights and monitoring the resulting accuracy of the neural network model for each bit change, we can conduct an analysis to detect any leakage of information through discrepancies in accuracy.

4.2 Fault Model

Our experiments aim at simulating the process of injecting faults on real hardware devices. In our fault model, the weights being attacked are treated in binary form. The process of injection faults on a MLP architecture is shown in Figure 6.

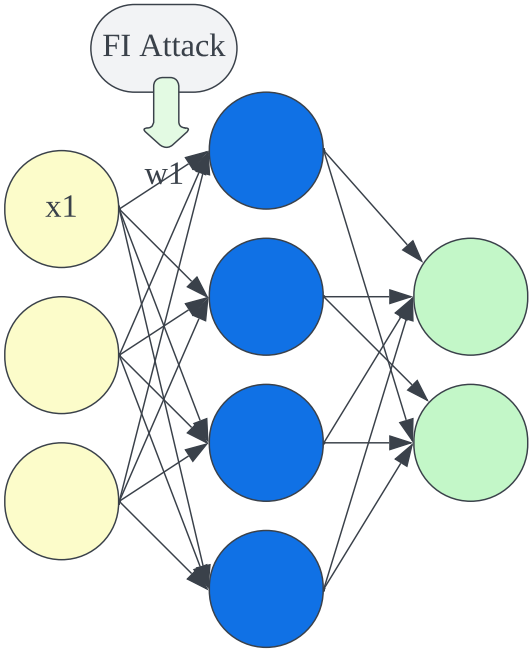

Figure 6: Fault injection attack on neural network example.

4.2.1 Bit-flip Attack

Bit-flip attack is a common method of fault injection attack to deliberately inducing bit errors in memory of a computing system to cause wrong data storage and, consequently, wrong or faulty data processing [37]. This alteration can lead to unexpected behavior in the targeted system, potentially allowing the attacker to gain unauthorized access, bypass security measures, or cause the system to

malfunction in a controlled manner. For the case of deep neural networks, bit-flip attacks may drastically alter the behavior of model, affecting its performance through aggressive accuracy drops.

The attacker can use bit-flip attack strategy to affect specific bits on the neural network. In order to evaluate the robustness of MLP models against fault injection attacks, we use simulated bit-flip attack model to characterize the model's behavior as a first part of our experiments. Through this method, we are able to analyze which layer or weight in the neural network is more sensitive or vulnerable to bit-flip by considering an adversary as defined in the threat model of Section 4.1. As an expected behavior of MLP models, faults injected in different layers should produce different impacts on the model's accuracy. Because our threat model is white-box (i.e., the adversary has access to model hyperparameters and model weights), the adopted strategy for our simulations assume that the attacker can target specific layers and weights. Moreover, we consider the worst-case scenario for this characterization in which the adversary can also target specific bits in the model weights. Although this sounds a very optimistic fault model, our study serves as baseline to understand the damages that could be caused to the model by a very strong adversary.

4.2.2 Stuck-at-zero Attack

Ineffective Fault Attack and Stuck-at-zero Attack In the work of Clavier et al [12], the definition of ineffective fault attack was first proposed that certain errors in embedded systems can be used to infer the intermediate bits of the keys of encryption algorithms. Now we can extend to the stuck-at-zero attack. For example, in a fault injection attack on an AES key, the attacker converts the bit under attack to 0 but leaves the other bits unchanged. If the encrypted text can still be received, it proves that the attacked bit was originally 0; the attacker does not change anything through the attack. On the contrary, if the encryption text cannot be received, the attack can know that the attacked bit should be 1 at first. The attacker can obtain the complete encryption key by attacking bit by bit. Meanwhile, this method can be used to crack encryption keys and extended to crack any data that can be represented in binary. Therefore, we would apply this method to attack and restore the trainable parameters of a certain neural network model, such as the weights we want to recover, etc.

Stuck-at-zero Attack on neural network Now we utilize the stuck-at-zero attack on neural network model. When we attack the weights of the neural network, every time we conduct an attack on a bit, we convert it into 0, no matter if it was 0 or 1 before. Then we observe the accuracy, if the accuracy changes, we can infer that the original bit is 1. If the accuracy does not change, the original bit might be 0, or the bit being attacked is not a sensitive weight, and the impact of the bit change is negligible for the model due to the complexity of the neural network, but here we assume it is 0. Thus, through this method, we can theoretically recover all the bits of the neural network.

In other work, stuck-at-one attack [20] is also used, which converts the bit being attacked into 1. Theoretically, there are not any differences between the two methods, but in real physical experiments on hardware devices, the difficulty of converting the bit into 0 and 1 can be different, which is based on the attacking methods(EMFI, voltage manipulation, or laser injection) or the hardware devices [22].

4.2.3 The effect of faults on intermediate data

During the encryption process, the results caused by injected faults are affected by many aspects, such as physical attack methods(laser, EMFI, etc.) or calculation parameters [15]. However, once faults occur, they have a common characteristic. They all change the original correct n-bit intermediate value x to a fault value x'. By analyzing different fault models, we can get the fault distribution table as shown in Figure 7, which can be used to analyze the impact of different faults. In real physical experiments, a very strong attack is required to ensure a perfect attack. We also need to add noise to the experiment to change the probability of a successful attack.

(a) Stuck-at-0

x \ x'	00	01	10	11
00	1	0	0	0
01	1	0	0	0
10	1	0	0	0
11	1	0	0	0

(b) Random-And

x \ x'	00	01	10	11
00	1	0	0	0
01	$\frac{1}{2}$	$\frac{1}{2}$	0	0
10	$\frac{1}{2}$	0	$\frac{1}{2}$	0
11	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

(c) Bit-flip

x \ x'	00	01	10	11
00	0	0	0	1
01	0	0	1	0
10	0	1	0	0
11	1	0	0	0

(d) Random fault

x \ x'	00	01	10	11
00	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
01	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
10	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
11	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

Figure 7: Fault distribution [15].

4.3 Weights Encoding

As discussed in Section 2, ensuring security against attacks on embedded computing systems, such as micro-controllers, is crucial. In almost all of computing systems, data transmission, data storage and data processing rely on information treated as binary data. Since neural network weights can also be represented as negative floating numbers, the binary form is much different from encryption processes, which are common fault injection attack targets. Thus, to implement a fault model characterization for neural networks, we also need to take into account the specific representations of the weights in binary form. We need a complement form with a sign bit for the negative numbers and the float part needs a special representation as well. Now, when we combine these components, we can get the following equation to define the conversion from binary $\left(\boldsymbol{b} = \left[b_{N_q-1}, \ldots, b_\epsilon\right] \in \{0,1\}^{N_q}\right)$ to decimal $w \in \mathbf{W}$ form of positive and negative floating numbers [37]:

$$w = -2^{N_q-1} \cdot b_{N_q-1} + \sum_{i=\epsilon}^{N_q-2} 2^i \cdot b_i \tag{2}$$

where w is a weight in the neural network in decimal format. In the equation (2), N_q is the length of the binary form, ϵ represents the number of digits after the decimal *dot* of weight w. According to this equation, we can also derive the reverse process of converting from decimal form into binary form correspondingly. This is a significant component if we aim to inject faults into the weights' bits, it is the prior condition for bit-flip and stuck-at-zero attack.

Based on the equation above, to simulate the process of injecting faults on real

hardware devices, in our experiments, we use the binary representation method as shown in Figure 8. Note that neural network weights may also contain negative float numbers. In the binary format of a number for the neural networks considered in this works, there are 4 bits before the decimal dot. The first bit represents the sign. If the bit is 1, the weight is a negative number; otherwise, it is positive. To represent the weight value with good precision, there are 8 bits after the dot in the binary format. This means that we can represent float numbers in the range $[-15.99609375, 15.99609375]$ (the corresponding binary format: $[0111.11111111, 1111.11111111]$). This range covers all the trained weights in our models, which are mostly within the range of $(-1, 1)$. Because when we initialize the MLP models, the range of weights is set to $(-1, 1)$. Such expressions cover the range and ensure precision, making our simulation experiments more reliable.

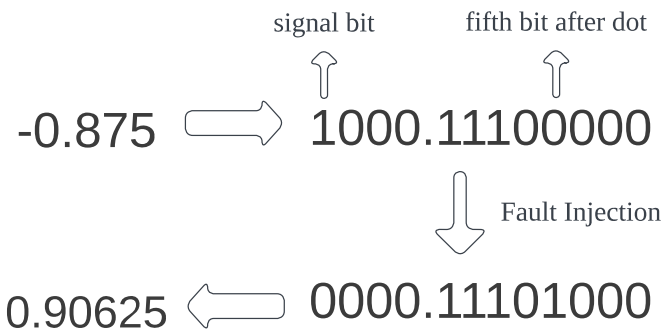

Figure 8: The converting of numbers from decimal to binary form.

A simple example of a fault attack to flip one bit of a weight value is shown in Figure 8 as a flowchart process. In this case, we simulate a fault injection on a weight which as the the value -0.875 in decimal format. However, in memory, this weight value is stored in its binary format. Suppose the adversary wants to perform a bit-flip attack on both the signal bit and on the fifth bit after the decimal dot. Then we flip the fifth bit after dot from 0 to 1 and convert the signal bit from 1 to 0. Figure 8 also shows the resulting faulty weight in decimal format. As a result, the weight value after the fault injection attack becomes completely different. For the weights of a neural network, such changes could have significant impacts on the accuracy.

4.4 Simulation

So far we have introduced fault attack considered in this work. Now, we go further to discuss how to combine these methods with our experimental setup to carry out the following analysis. In this section, we provide details about simulated fault injection experiments. We first start with the characterization of the model and after we describe how we use the stuck-at-zero fault model for model weights recovery. We also provide pseudo-codes for our simulated fault models.

4.4.1 Model characterization using bit-flip attacks

In evaluating MLP models against fault attacks, we focus on the bit-flip attack model. In this model, each attack involves flipping a targeted bit to its complementary value: a 0 becomes a 1, and vice versa. We systematically record the model's accuracy after each attack, allowing us to analyze how various attack scenarios affect accuracy. This analysis enables us to assess the sensitivity and robustness of different layers and individual bits within the model. The procedure for injecting a bit-flip fault into a trained MLP model is outlined in Algorithm 1.

Algorithm 1 Simulated Bit-flip Fault Attack for Model Characterization

Input: $weights, bit\_position, mlp\_model, test\_set$

Output: $weights\_faults$

procedure SimulatedBitFlipAttack(mlp_model)
  for $i = 0$ to $len(weights)$ **do**
    Decimal_to_Binary($weight$)
    $weight\_fault$= inject_fault_bitflip($bit\_position, weight$)
    Binary_to_Decimal($weight\_fault$)
  end for
  evaluate_accuracy($faulty\_model, testset$)
end procedure

Algorithm 1 takes several inputs including the weights of the model, the position of the bit to be manipulated, the MLP model itself, and a test set to check the accuracy after the fault. The goal is to simulate the effect of bit-flip faults on the model's weights and assess its performance. The algorithm works as follows:

- Input parameters: the algorithm takes as input the weights of the model, the position of the bit to be flipped, the MLP model itself, and a test set.

- Procedure initialization: the procedure *SimulatedBitFlipAttack* is initiated.
- Iterative weight manipulation: For each weight in the model:
  - Convert the weight from decimal to binary representation to facilitate bit manipulation.
  - Inject a fault by flipping the bit at the specified position using the $inject\_fault\_bitflip$ function.
  - Convert the manipulated weight back to decimal.
- Model evaluation: after all weights have been manipulated, the accuracy of the faulty model is evaluated using the *evaluate_accuracy* function with the provided test set.

Using this approach, we can carry out various characterization experiments. For instance, we can compare the relative sensitivity of different layers within neural networks or identify the most vulnerable bit positions. Concurrently, we can explore strategies to enhance the stability of the neural network. In the upcoming section, we will systematically investigate and analyze these issues, providing insights into the research questions through an examination of the corresponding results.

4.4.2 Model weights recovery using stuck-at-zero attack

In the subsequent phase of our experiments, we implement a black-box attack strategy for weight recovery. Here, the attacker possesses knowledge regarding the model's structure, including the number of layers and neurons, as well as access to the test dataset. However, crucially, the trainable parameters remain undisclosed, and access to the training dataset is restricted. Consequently, the attacker is tasked with recovering the model's weights by leveraging fault injection attacks on the original neural network. To accomplish this, we adopt the stuck-at-zero model. The pseudocode outlining this process is delineated in Algorithm 2.

Algorithm 2 takes as input the weights of the model, the position of the bit to be manipulated, the MLP model itself, and a test set. The primary objective is to recover the weights of the model through a fault injection attack, specifically utilizing the stuck-at-zero model. Here's a breakdown of how the algorithm operates:

Algorithm 2 Simulation Stuck-at-zero Attack

Input: $weights, bit\_position, mlp\_model, test\_set$
Output: $weights\_faults$
procedure SIMULATEDMODELRECOVERYATTACK(mlp_model)
  for $i = 0$ to $len(layers)$ **do**
    for $j = 0$ to $len(weights)$ **do**
      Decimal_to_Binary($weight$)
      $weight\_fault$= inject_fault_stuck_at_0($bit\_position, weight$)
      Binary_to_Decimal($weight\_fault$)
    end for
    set_weights(layer_i)
  end for
  evaluate_accuracy($recovered\_model, testset$)
end procedure

- Input parameters: the algorithm requires the weights of the model, the bit position to be manipulated, the MLP model, and the test set as input.
- Procedure initialization: the procedure $SimulatedModelRecoveryAttack$ is initiated.
- Layer-wise weight manipulation: For each layer in the neural network:
  - Iterate through each weight within the layer.
  - Convert the weight from decimal to binary representation for manipulation.
  - Inject a fault using the stuck-at-zero model to set the targeted bit to 0.
  - Convert the manipulated weight back to decimal.
- Updating weights: after manipulating all weights within a layer, update the weights of the layer in the model.
- Model evaluation: once all layers have been processed, evaluate the accuracy of the recovered model using the provided test set.

Our approach involves systematically attacking the neural network layer by layer, targeting different bits within each weight. Specifically, we flip each targeted bit to 0 using the stuck-at-zero fault model. If we observe a drop in the model's accuracy on the test set, it indicates that the originally targeted bit was 1. Conversely, if there's no discernible change in accuracy, we infer that the attacked bit was originally 0. This distinction is crucial, as it helps identify sensitive bits within the model. Once we've successfully recovered all weights across all layers,

we evaluate the accuracy of the extracted model on the test dataset, comparing its performance metrics with those of the original model.

Given the inherent difficulty in achieving a perfect attack in practice, we also conduct weight recovery experiments with varying levels of noise. This involves simulating attacks with success rates of 70%, 80%, and 90%. Furthermore, since the attacker has access to the test dataset, we explore the option of training the extracted model (recovered model) using this data. Subsequently, we compare its performance with that of the original model and analyze the convergence process. This comprehensive approach enables us to gain insights into the robustness of the model against adversarial attacks and the effectiveness of our recovery techniques.

4.5 Summary

The present section described two simulated fault attack scenarios against MLP model weights: bit-flip model and stuck-at-zero model. Based on these two fault models, we provided the description of the algorithms considered in our simulated fault injection attacks for model characterization and model stealing (weights recovery).

5 Experimental results

5.1 Model characterization

In the initial phase of our experimentation, we explore the characterization of fault injection on both the MLP models trained with Spine and WDBC datasets. Through the injection of various errors, we meticulously examine the distribution of faults, shedding light on the sensitivity of bits positioned at various locations, such as sign bits and those immediately following the decimal dot. Additionally, we examine the susceptibility of different model layers to fault injection attacks. Subsequently, we extend our analysis to models with diverse structures, aiming to extract information about their resilience against such attacks. To mitigate the impact of randomness, all experiment results are derived from the average values of ten trials conducted with distinct random seeds. It's noteworthy that this segment of experiments employs white-box attacks, assuming the attacker possesses comprehensive access to all information pertaining to the targeted models, except for the training dataset.

5.1.1 Characterizing the impact of individual bits in model weights

Table 1 presents the distribution of weights across the layers of the MLP models trained on the Spine and WDBC datasets. For clarity, we distinguish between these models as the Spine model and the WDBC model, respectively. As both models are fully-connected neural networks, the number of weights in each layer is determined by multiplying the number of neurons in the preceding layer by the number of neurons in the current layer. Notably, due to discrepancies in the feature counts of the datasets, the input layer of the Spine model comprises 6 neurons, contrasting with the WDBC model's input layer, which encompasses 30 neurons. However, we maintain uniformity across the hidden layers and output layer in both models, with each hidden layer housing 10 neurons and the output layer featuring a single neuron.

Model	Hidden layer 1	Hidden layer 2	Output layer
Spine model	60	100	10
WDBC model	300	100	10

Table 1: Number of weights in each layer of the Spine and WDBC models.

Firstly, we initiate training for Spine and WDBC models utilizing the initial structures outlined in Table 1. Subsequently, leveraging the two MLP models as foundations, we introduce faults across various bit positions and layers to verify the fault distribution. This analysis enables us to identify which components of these MLP models are more susceptible to bit errors. Furthermore, we explore fault induction in models employing diverse activation functions and regularization methods, as elaborated in Section 2. Activation functions encompass ReLU, Sigmoid, Tanh, and Identity, while regularization methods include L2 regularization and dropout layers, both method to reduce overfitting.

To evaluate the impact of individual bits on robustness, we systematically attack bits at various positions within each weight of the two models. At each iteration, upon detecting an error in a bit, we record the accuracy of the affected model on the test set. Subsequently, we restore the neural network to its original state and proceed to attack the next bit, repeating the process.

To quantify and illustrate the results obtained after injecting simulated faults, we keep the *worst accuracy* obtained by flipping a specific bit position in each weight of a layer. Going weight by weight in the layer, we keep the worst accuracy. It's important to note that the accuracy values presented represent the worst-case scenario obtained by attacking bits one by one. This allows us to verify what is the most vulnerable bit and most vulnerable weight in the attacked layer. Results of simulated fault attack results against Spine and WDBC models are shown in Figures 9 and 10. These figures show results for fault injected in the first layer. The x-axis represents the index of the affected weight in the layer. The y-axis shows the worst accuracy obtained until certain weight index (which explains why the worst accuracy has a decreasing trend). These figures show worst accuracy results for simulated faults on one bit of the weight at the time, including the sign bit. The faults are injected only on the bits after the dot as for that all the weights of the two models are in the range of $(-1, 1)$.

As shown in Figures 9 and 10, the signal bit exhibits the most pronounced influence on the neural network's accuracy, followed by the first bit subsequent to the decimal point, and so forth. This aligns with our initial assumptions, as the signal bit dictates the sign of the weights under attack, resulting in more significant disruptions to the model. Additionally, the n-th bit after the decimal point regulates the range by a factor of 2^n (where n ranges from -1 to negative infinity). For instance, the first bit after the decimal point governs a range of 0.5, while the second bit controls a range of 0.25, and so forth. Notably, the

controllable range diminishes with each subsequent bit after the signal bit. When we apply our bit-flip fault model to 5th to 8th bit after the dot, the accuracy drops also happens, however with less impact. However, this shows that even when affecting bits with less importance to the weight value, the model's accuracy is also affected.

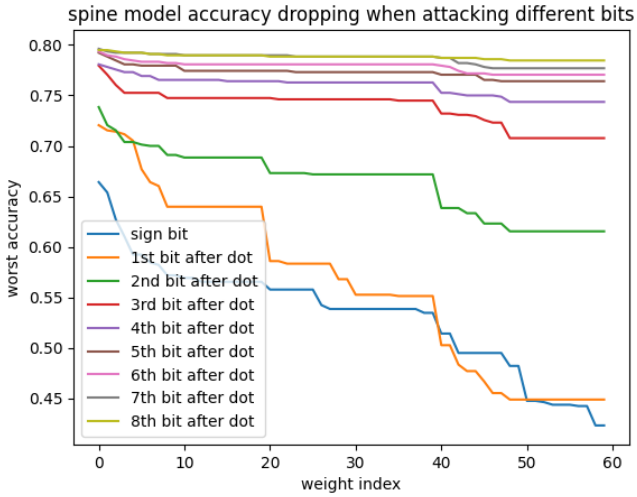

Figure 9: Worst accuracy obtained when attacking different bits of every weight in the first hidden layer of Spine model

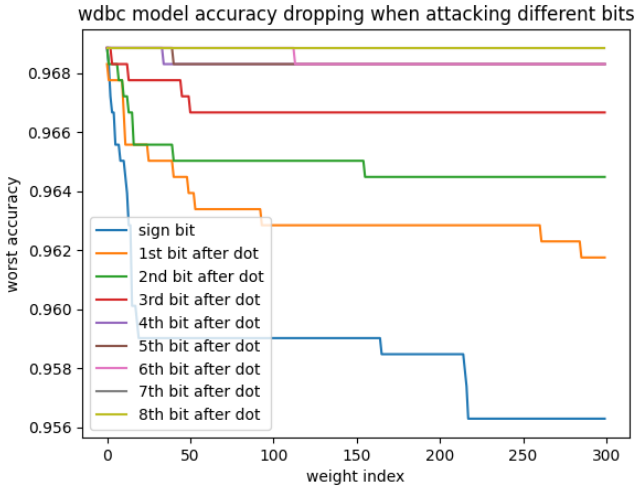

Figure 10: Worst accuracy obtained when attacking different bits of every weight in the first hidden layer of WDBC model

5.1.2 Characterizing the impact of regularization on model robustness

To get the fault distribution and assess the impact of errors on various layers of MLP models, we conduct experiments targeting the first bit of every weight within different layers of both the Spine and WDBC models. Building upon our previous experiments characterizing the effects of bits at different positions, we recognize that the first bit of the weights demonstrates significant impact and serves as a representative measure. Consequently, we opt to target the first bit for the subsequent experiments. Moreover, we conduct comparisons between models with and without L2 regularization to analyze the influence of regularization techniques on robustness against the evaluated fault model, and the hyperparameter of L2 is set as 0.001, which is a value in the reasonable range of $(0, 0.1)$ [26].

The results, as illustrated in Figure 11 and 12, showcase the worst accuracy observed in both models when targeting the first bit of every weight across different layers. Notably, it's evident that the first layer exerts a more substantial impact on accuracy. This observation can be attributed to the inherent characteristic of forward propagation in neural networks, wherein the output of each layer serves as input to the subsequent layer. Consequently, errors occurring in the first layer can propagate to subsequent layers, potentially amplifying and culminating in classification errors at the output layer, thereby leading to a drop in accuracy. In Figures 11 and 12, the line plots have different lengths because each layer has a different number of weights. It is clear that L2 regularization does not create any positive impact for model robustness as the accuracy drops is the same as when L2 regularization is not considered.

Additionally, we incorporate another regularization method during the pre-training phase of our models, namely dropout layers with a dropout rate of 20% inserted between every two layers. Figures 13 and 14 illustrate that the utilization of dropout layers leads to increased vulnerability of the neural networks to faults. This observation suggests that while dropout layers enhance model versatility by randomly dropping out outputs from certain neurons to combat overfitting, they concurrently compromise performance and render the models more susceptible to the implemented fault model.

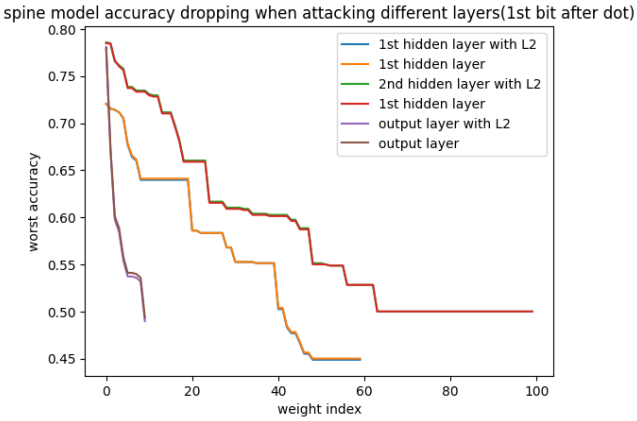

Figure 11: Results when attacking the first bit of every weight in different layers of Spine model.

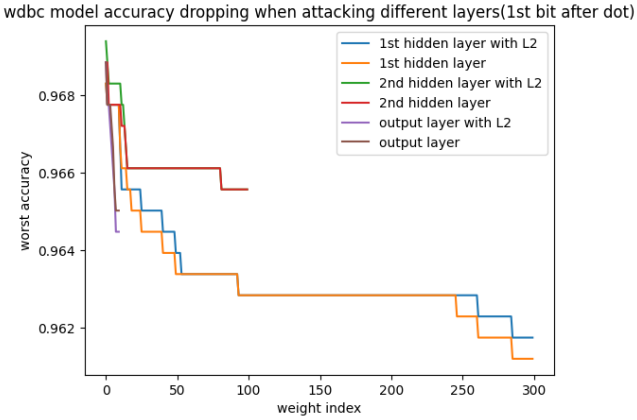

Figure 12: Results when attacking the first bit of every weight in different layers of WDBC model.

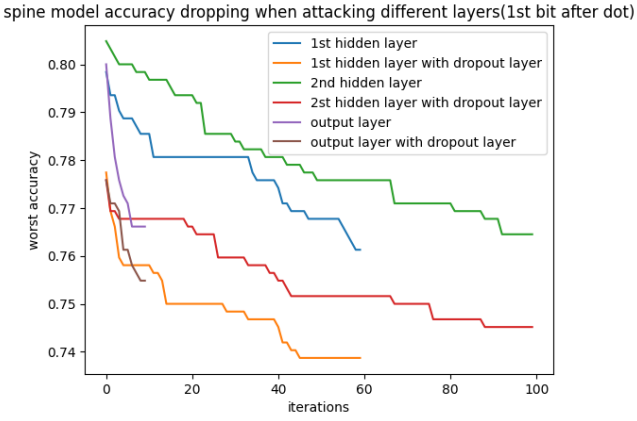

Figure 13: Attacking different layers of Spine model on the 1st bit of every weights.

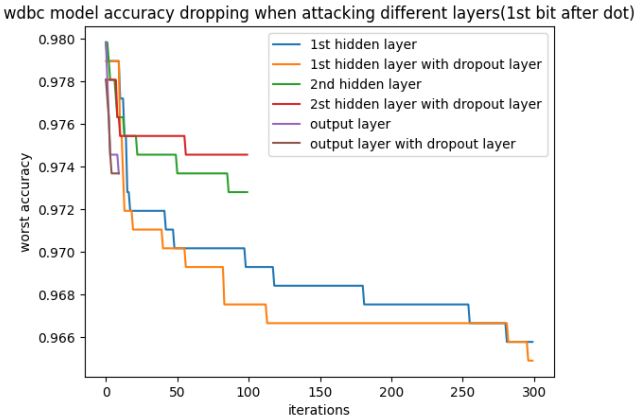

Figure 14: Attacking different layers of Spine model on the 1st bit of every weights.

5.1.3 Characterizing the impact of different activation functions on model robustness

Next, we assess how the choice of activation function influences the model's robustness against the considered fault model. Analysis of the results depicted in Figures 15 and 16 reveals notable variations in the worst accuracy observed when targeting models trained with different activation functions. In the case of the Spine model, attacking models employing tanh, ReLU, and identity activation functions leads to varying degrees of accuracy decline, with the identity activation function exhibiting the highest vulnerability. However, when using sigmoid activation function, the model is very stable. Intriguingly, when attacking the first bit after the decimal point of the model with sigmoid activation function, no discernible drop in accuracy is evident. This indicates the significant impact of the choice of activation function on the model's resilience against the evaluated fault model, with certain functions demonstrating greater robustness compared to others under the specified fault model.

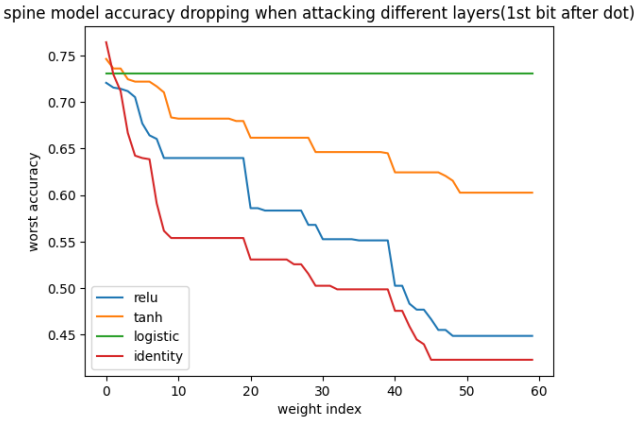

Figure 15: Results when attacking on Spine model with different activation function.

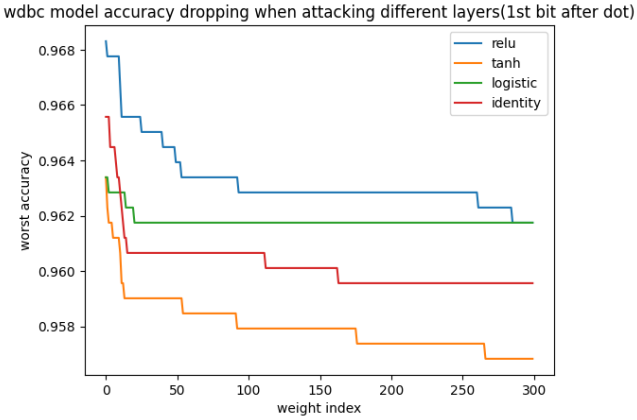

Figure 16: Results when attacking on WDBC model with different activation function.

Similarly, as depicted in Figure 16 for the WDBC model, attacking the sigmoid model results in a negligible decrease in accuracy, less than 0.002, indicating remarkable robustness compared to the other three activation function models. However, it's crucial to acknowledge that the sigmoid model also exhibits certain shortcomings. For instance, its initial accuracy is not as high as that of the trained models utilizing the other three activation functions.

Therefore, model owners are faced with a decision-making dilemma, needing to weigh the importance of initial accuracy against the model's robustness. This trade-off underscores the necessity for careful consideration and prioritization based on the specific requirements and objectives of the application at hand.

5.1.4 Characterizing the most-related-bits in the model

At this stage, we introduce a novel concept termed "most-related-bits," which encapsulates our comprehensive attack strategy targeting all bits of all weights across all layers of the model. This approach subjects each bit to a bit-flip attack and recording the resulting model accuracy. Subsequently, we sort the accuracy values obtained from attacking every bit, thereby identifying the top bit in this sorted list as the most relevant bit.

Moving forward, we utilize the Table 2 as a guideline for conducting subsequent experiments, attacking the model in the order prescribed by this table. This systematic approach ensures that we prioritize the bits with the highest relevance, facilitating a more focused and targeted analysis of the model's vulnerability to bit-flip attacks.

Model	layer	row	column	position	accuracy
most-related-bit 1	1	5	8	signal bit	0.41
most-related-bit 2	1	4	9	signal bit	0.470
most-related-bit 3	1	4	8	#1 bit	0.471
most-related-bit 4	2	8	9	signal bit	0.471
most-related-bit 5	1	2	3	signal bit	0.471
...					

Table 2: Example of most-related-bits of Spine model, row and column represent the position of the weight in its layer, and the #1 bit refers to the first bit after dot.

From the experimental results shown in Figures 17 and 18, we can see that accuracy drops are observed in both models under attack. Specifically, in the Spine model, attacking the top-3 most-related bits collectively results in a substantial 40% decrease in accuracy. Intriguingly, a slight increase in accuracy is observed upon attacking more bits of the Spine model. We speculate that this phenomenon may arise from the intricate interactions and combinations of altered bits, which can impact the model's stability due to the inherent complexity of neural networks.

Conversely, in the WDBC model, attacking the 20 most-related bits leads to only a modest 5% decrease in accuracy. However, a steady decline in accuracy is observed as more most-related bits are targeted. Notably, when attacking 100 most-related bits simultaneously, the accuracy of the attacked WDBC model drops to approximately 70%. These findings show the varying degrees of vulnerability exhibited by different models and highlight the importance of understanding

the complex relation between bit alterations and their collective impact on model performance.

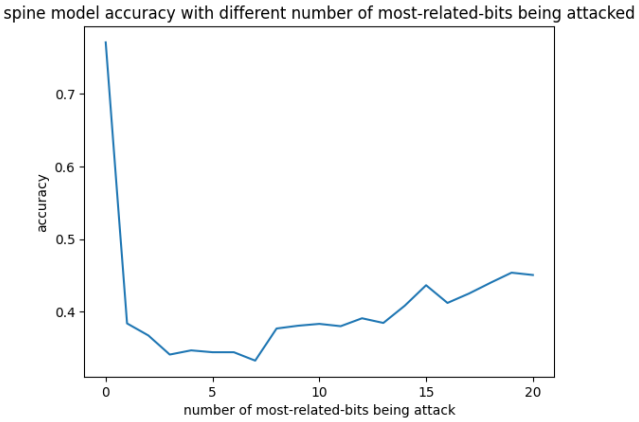

Figure 17: Results when attacking on Spine model with different numbers of most-related-bits being attack.

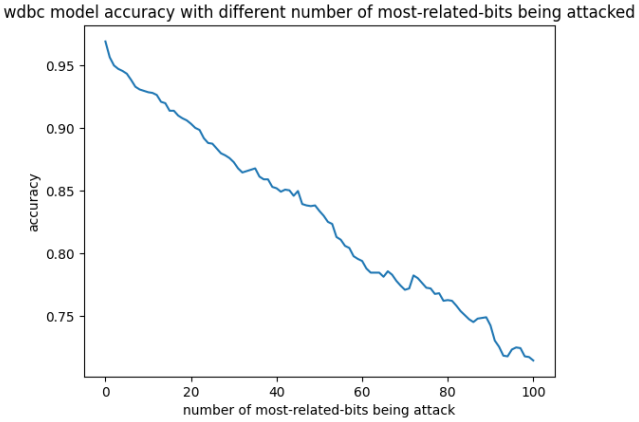

Figure 18: Results when attacking on WDBC model with different numbers of most-related-bits being attack.

The substantial difference in accuracy drop observed when attacking the most-related bits of the two models suggests a potential correlation with the number of weights in the first hidden layer. The structural dissimilarity lies primarily in the number of weights present in this layer: 60 weights for the Spine model and 300 weights for the WDBC model. This variance in the number of trainable parameters in the first hidden layer indicates that it may significantly influence the robustness of MLP models against fault injection for the fault model considered in this work.

5.1.5 Characterizing the impact of first layer size in model's robustness

In the subsequent stage of characterization, we intend to systematically experiment with varying numbers of trainable parameters in the first hidden layer of the two models. This analysis aims to verify whether this hyperparameter (i.e., number of neurons in the first layer) indeed plays a crucial role in determining the models' susceptibility to the evaluated fault model, thereby providing insights into optimizing MLP model architectures for enhanced robustness and resilience.

The results presented in Figures 19 and 21 indicate that increasing the number of trainable parameters in the first layer leads to a gradual and consistent improvement in the minimum accuracy of the models after the fault. Moreover, as shown in Figures 20 and 22, even when errors are injected into the second layer while the number of trainable parameters in the first hidden layer is increased, a similar trend of gradual improvement in minimum accuracy is observed. This suggests that augmenting the neural network parameters of the first layer exerts a stabilizing influence on the overall robustness of the neural network. By enhancing the complexity and capacity of the first layer, the neural network becomes better equipped to resist and recover from errors injected at subsequent layers. This indicates the critical role played by the architecture and configuration of the initial layers in determining the overall resilience and performance of MLP neural networks against the evaluated fault model. Finally, Figures 19 to 22 also include results when L2 regularization is considered. The illustrated results indicate that L2 regularization plays a insignificant role in model's robustness against our fault model.

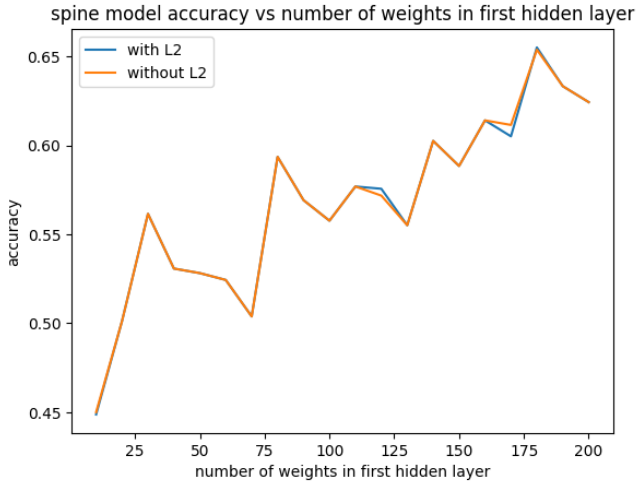

Figure 19: Results when attacking on Spine model with different numbers of parameters (weights) in the first layer.

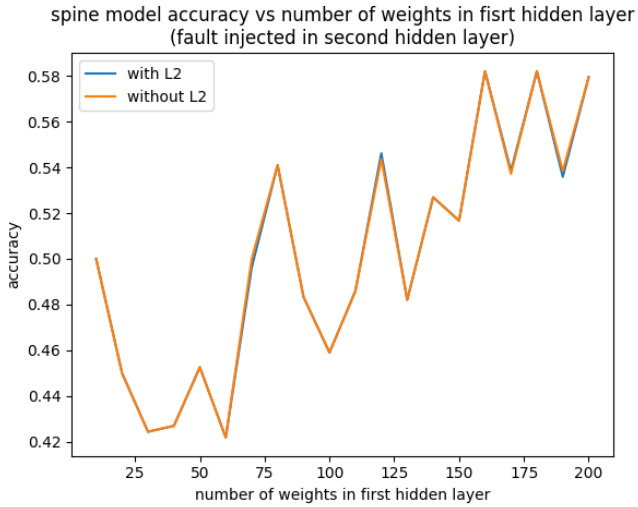

Figure 20: Results when attacking on Spine model with different numbers of parameters (weights) in the first layer.

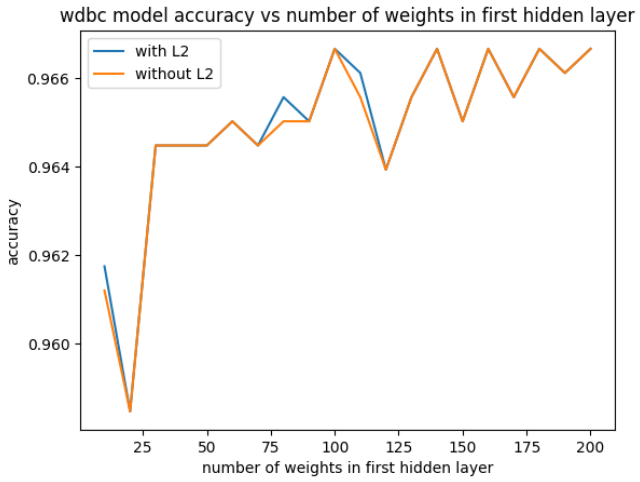

Figure 21: Results when attacking on WDBC model with different numbers of parameters (weights) in the first layer..

Figure 22: Results when attacking on WDBC model with different numbers of parameters (weights) in the first layer.

5.2 Model weights recovery

In this section, we employ the stuck-at-zero fault model to deliberately induce faults in the weights of MLP models trained with iris and wine datasets. We refer to these two MLP models as iris and wine models, respectively. Subsequently, we attempt to recover these faulty weights and assess the accuracy of the models using the recovered weights. This evaluation serves as a crucial measure to measure

the efficacy and integrity of our weight recovery approach.

In these experiments, we operate under a specific threat model in which the attacker possesses knowledge solely of the structure of the trained MLP model (i.e., its hyperparameters), which includes the number of layers and neurons within the MLP models. However, the attacker lacks access to the trainable parameters, such as weights and biases. Under this threat model, the dataset is partitioned such that half is allocated for training and the other half for testing. The attacker has no access to the training set, however has access to the test dataset.

During the attack, the adversary employs the stuck-at-zero fault model to manipulate all weight values of the MLP model. The test dataset is utilized to evaluate the accuracy of the recovered model. This evaluation process involves comparing the performance of the recovered model against the original model, which was trained using the training dataset. Through this methodology, we aim to assess the efficacy of the recovery process and see how the recovered model compares to the original model.

The numbers of weights of the iris and wine trained MLP models are shown in Table 3. Considering that the numbers of features are different in two datasets, separately there are 4 neurons and 11 neurons in the input layer, the numbers of weights and layers are also different in the two MLP models. The iris model is smaller with only one hidden layer and, therefore, less weights. In the wine model, there are two hidden layers and more neurons in each layer.

Model	Hidden layer 1	Hidden layer 2	Output layer
iris model	40	————	30
wine model	260	200	30

Table 3: The number of weights in each layer of iris and wine model.

As stuck-at-zero fault model can be employed to recover keys in cryptographic algorithms [3], we suppose that it is also feasible to recover parameters of neural networks through stuck-at-zero injection fault attacks. We aim to verify our assumption through two models of different sizes. Additionally, we would employ this method to see the process of restoring the neural network weights and see the completeness of the recovery by comparing the accuracy of the extracted model and the original model with the test set. In the following, we would refer to the model that we recovered and extracted through the stuck-at-zero fault injection attack as the extracted model, and the model obtained through the original training but without attack is called the original model.

As shown in Figures 23 and 24, we perform stuck-at-zero attacks on MLP models trained with iris and wine datasets. We attack the model layer-by-layer and weight-by-weight. For example, for the first hidden layer of the iris model, we first attack the sign bit of every weight. After conducting a stuck-at-zero attack on the sign bit of one weight, we record the accuracy. Subsequently, we restore the neural network to its original state and proceed to attack the signal bit of the next weight. Restoring the model is assumed to be possible, e.g., by the attacker by simply turning-on and off the target embedded device.

Following, we attack the first bit after the decimal point for every weight, and so on, until we attack all the bits of all layers. We can also show the accuracy drop as heatmaps. Through the heatmaps, we can clearly observe the accuracy differences when we deduce faults on different bits. If there are accuracy drops, we set these bits to 1, other bits are assumed as 0.

Figure 23: Iris model attacking accuracy array and heatmap, the left one is when we attack the first layer of the iris dataset, the model accuracy after attacking the bits at different positions of each weight, and the right picture is when we attack the first layer heatmap of bit time relative to accuracy.

Model	iris model	wine model
original Model without attack	0.501	0.220
attack without noise	0.735	0.717
with 90% Probability	0.691	0.709
with 80% Probability	0.548	0.564
with 70% Probability	0.508	0.560

Table 4: The accuracy of the extracted models and the original models.

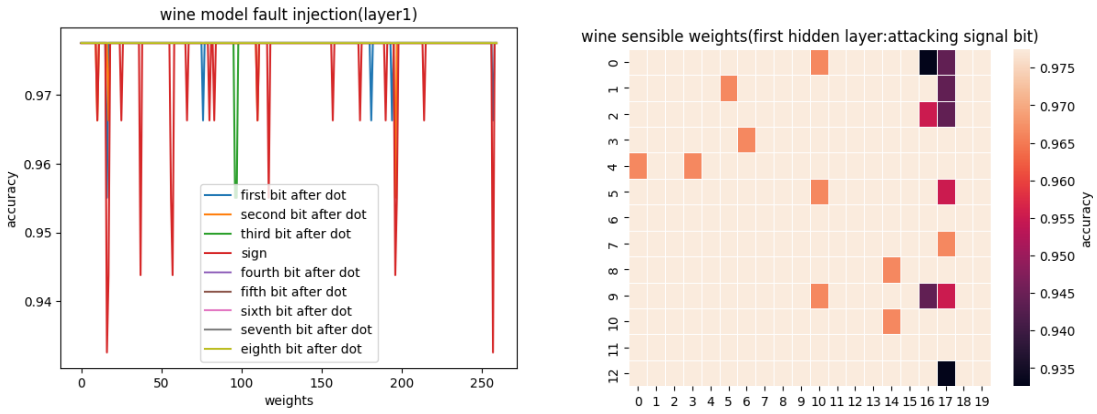

Figure 24: Wine model attacking accuracy array and heatmap, the left one is when we attack the first layer of the wine dataset, the model accuracy after attacking the bits at different positions of each weight, and the right picture is the accuracy heatmap when we attack the signal bit of different weights on the first hidden layer.

We observed larger accuracy drop on iris model. When attacking a certain bit, the worst accuracy of iris model is 65%, but the lowest accuracy of the wine model is 93%. We think this is due to the wine model is much larger compared to the iris model. There are more layers and more trainable parameters in the wine model, so the wine model would be less affected and more robust and stable than the iris model. This is also consistent with the conclusion drawn in our characterization experiments.

In our threat model, we also assume that the attacker has access to the test set but not to the training set. Thus, we also retrain the recovered models (i.e., the models with recovered weights) with test set and record the training process. We are able to make a comparison with the original model's performance with randomly initialized weights (note that the attacker can perform model initialization as the hyperparamaters are known). In addition, to simulate the process in real physical experiments, we enhance our evaluation with noise, which is the possibility of a successful attack, as shown in Figures 25 and 27. In these figures, "attack with 90% possibility" means that we have 90% possibility to convert a bit to 0, and 10% possibility to conduct an unsuccessful attack. Because in real physical experiments, due to the complexity of the hardware computing system, perfect attacks cannot be guaranteed [3].

Based on the results shown in Figure 25 and 26, we notice that when we use the extracted models for training, the model's training process converges much faster than the original model. For the iris model, the original model takes 35 epochs

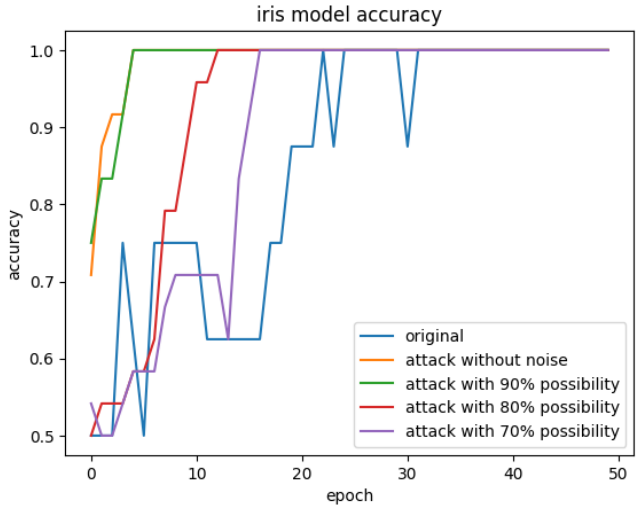

Figure 25: iris model accuracy, attacking the model

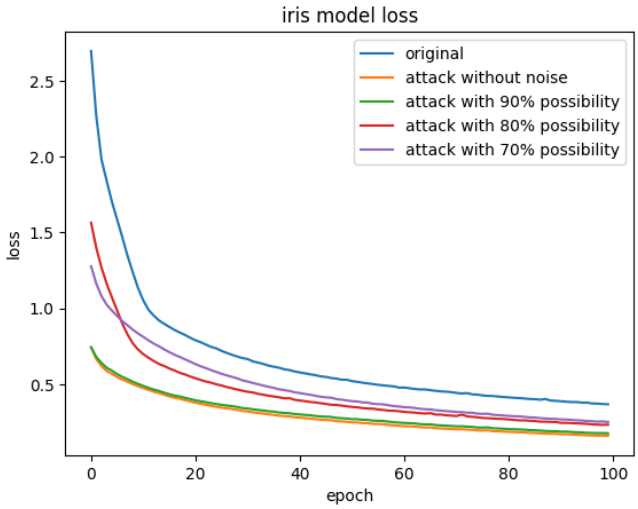

Figure 26: iris model loss, attacking the model

to converge, while the extracted model with perfect attack and the model with a 90% probability of success only take 5 epochs to converge. When the noise value is higher, the extracted models converge within 16 epochs. In addition, according to Figure 26, we can observe that the loss values of our extracted models are overall lower than the original model.

We can obtain similar results in the wine model experiments. The extracted model recovered by the perfect attack can achieve a high accuracy of 75% at the beginning. Then we continue to use the test set to train the extracted model, which only takes 20 epochs to converge. From the results with noise, we can

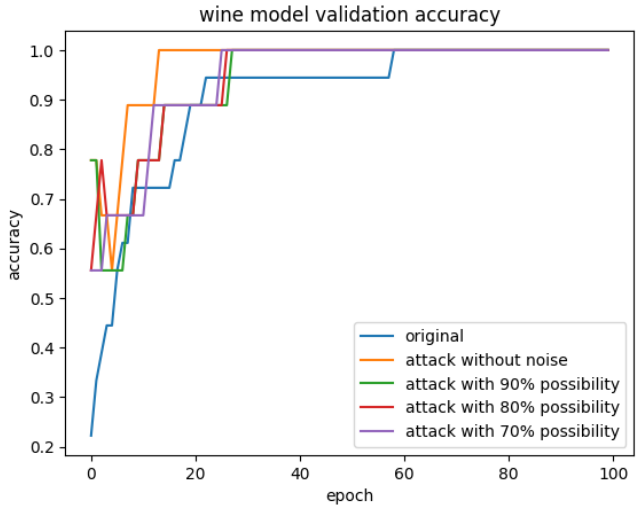

Figure 27: Extracted wine model validation accuracy compared to the original model

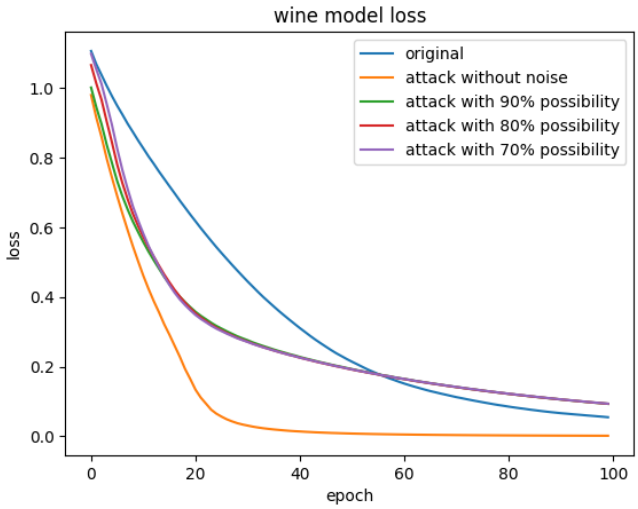

Figure 28: Extracted wine model loss compared to original model

see that the model converges within 30 epochs. According to Figure 28, it is notable that the loss value of our extraction model with perfect attacks can reach convergence rapidly, and the converged loss is also lower than the original model. For attacks with noise, the losses also converge fast. But we can also notice that the final loss result of the noisy attack is still worse than the original one. We guess this is because the noisy attack gives incorrect initialization, which leads to poor performance in the subsequent training process.

Through the simulation experiments of weights recovering, we confirm our hypothesis that using fault injection attack, the parameters of the neural network

can be recovered and extracted. When accuracy of the original models is nearly 100%, through fault injection attack, the models we recover can reach 75% accuracy. If the possibility of a successful attack is 90%, the extracted models can reach 70% accuracy. In addition, if we retrain the recovered models with the test sets, they converged much faster compared to the training process of the original models. In addition, we converted all weights of the original models and the extracted model weights into binary format. Consider the optimal attack model (i.e., a perfect attack without noise), we recovered 81.1% and 79.0% bits of the iris and wine models respectively.

6 Discussion

6.1 Answers to research questions

Based on the experiment results, now we can discuss in detail towards the research questions.

1. For a white-box threat model (i.e., the attacker knows everything except the training data), what is the effectiveness of fault attacks against deep neural networks?

   For the Spine and WDBC models, WDBC is more stable than Spine, According to the experimental results of our characterization, we can find that for relatively small models such as the Spine model, even only 3 of the most-related-bits need to be attacked to achieve the lowest accuracy of the model. If the attacker wants to destroy a larger model like WDBC, more bits need to be affect with a bit-flip fault model. Based on the results, attacking 20 most-related-bits can make the model drop to 90% of accuracy. When attacking 100 most-related-bits, the accuracy can drop to 75%. Nevertheless, our simulated results emphasize that even under a white-box threat model, significant accuracy drops are achieved when affecting a larger amount of weight bits. For the strongest attacker to affect model's accuracy by changing only a few bits of a model, the selection of these most important bits for model's robustness is a difficult task and would require an excessive amount of injected faults.

2. What components of a neural network model demonstrate the highest resilience and susceptibility to fault attacks?

   According to the second stage of the characterization experiments, it is notable that when we use the sigmoid activation function, the model is the most robust in the face of attacks, but the performance of the model using the logistic activation function without attack is not as good as others. Thus the model user should decide which characteristic is more important, the security of the model or the performance without attack. Secondly, we also conducted a regularization analysis of the model. By comparing the

worst accuracy of attacking the models using L2 and dropout layers with the model without regularization, we found that L2 regularization can make the model slightly more stable (although the effect is almost imperceptible), but models with dropout layers are more vulnerable to fault injection attacks.

In addition, weights in the first hidden layer are more vulnerable to errors, we also conducted experiments with different trainable parameters of the first hidden layer. We can find that when we gradually increase the number of weights in the first layer, if we are also In the first layer of injection errors, the minimum accuracy we can achieve when attacking a certain bit gradually increases. If increase we the number of neurons in the first layer but deduce errors in the second hidden layer, the models are also more resistant when facing errors.

In general, according to our experimental results, MLP models with the following characteristic are more robust against fault injection attacks: more neurons in the first hidden layer, L2 regularization, and sigmoid activation function, but without dropout layers.

3. In a threat model in which the adversary has access to the model's hyper-parameters and is based on an optimistic fault model, to what extent can model parameters (i.e., weights) be recovered?

   We converted all weights of the original models and the extracted models into binary format. Consider the optimal attack model (i.e., a perfect attack without noise), we recovered 81.1% and 79.0% bits of the iris and wine models respectively.

4. In the previous fault model for the model's parameters stealing, can all bits be restored, and what is the accuracy of the recovered models?

   In the second part of our experiment, when we use stuck-at-zero to attack the neural network to restore the weights of the neural network, the accuracy of the original model can reach 95-99%. In this situation, our extraction model can achieve an accuracy of more than 70% on the test set. We also added noise. When there is a 90% probability of executing a successful attack, the extracted model can achieve performance similar to that of

the perfect attack model but slightly lower. When the probability of success attack is 70-80%, the accuracy is lower. When we use the test set to retrain our extracted models, the convergence speed is much faster than that of the original model. In the case of perfect attacks, the extracted models converge within only 1/5 of the number of epochs required by the original models. Therefore, an attacker can steal information of MLP models through fault injection attacks, and achieve good accuracy. This process is similar to fine-tuning a neural network with another dataset. In our case, our threat model determines that the attacker has access to a test set that is drawn from the same set as the training set. Therefore, fine-tuning the model after weight recovery is expected to provide accuracy similar or even better than the accuracy of the orginal model and with a short number of epochs.

6.2 Future work

In the work we have completed, we focus on using simulation methods to explore the security and stability of neural networks against fault injection attacks. In subsequent work, we hope to extend it to real microcontrollers to continue to verify the correctness of physical experiments. In this work, physical experimental verification could not be completed due to time limitations and the complexity of setting up the physical equipment. In addition, this work has verified the behavior of the MLP against simulated fault models. We hope to continue to advance the method to more complex neural networks, such as convolutional neural networls (CNNs) which are also highly deployed on embedded systems. Moreover, we plan to explore the effectiveness of evaluated fault models when neural network models include adversarial training to improve protection against fault injection.

7 Conclusions

In this work, we conducted two main groups of experiments to attack MLP models with simulated fault injection. In the first part, we performed bit-flip experiments on MLP models trained with Spine and wine datasets. From these experiments, we obtained the distribution of faults and characterized the sensitivity of different parts of the neural network models. We also analyzed what structure is more resistant and stable against fault injection attacks. At the same time, we verified whether neural network hyperparameters may act as natural countermeasures against evaluated faults. In the second part of our experiments, we also explored to what extent the neural network weights can be recovered through stuck-at-zero attacks fault model. this time, we considered a weaker adversary with access to model hyperparameters and to a test set. We found that for a simple model trained with iris dataset and a more complex model trained with wine dataset, and under a perfect attack, it is possible to restore the model to an accuracy of above 70%. In addition, we use the test dataset to retrain the models we extracted. Compared to the training process of the original model, the recovered models converged much faster. When there is attacking noise (i.e., when there are different successful possibilities to convert the attacked bit into 0), the accuracy is lower compared to the perfect attack, but the extracted models still converge faster than the original ones.

Acknowledgements

First of all, I would like to thank my three supervisors. Without their help, I would not have been able to complete my master's thesis. I would also like to thank SGS BrightSight for giving me the opportunity to do a thesis internship. I have received a lot of help from the company. Secondly, I would also like to thank my parents, who have always supported my studies and provided living assistance. I love you. Finally, I would also like to thank my friends who have supported me during this stage and provided emotional support. Even though I was very stressed while writing the thesis, having my friends there was a great help.

References

[1] Laith Alzubaidi, Jinglan Zhang, Amjad J Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, José Santamaría, Mohammed A Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions. *Journal of big Data*, 8:1–74, 2021.

[2] Mrinal R Bachute and Javed M Subhedar. Autonomous driving architectures: insights of machine learning and deep learning algorithms. *Machine Learning with Applications*, 6:100164, 2021.

[3] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[4] Guilherme Barreto and Ajalmar Neto. Vertebral Column. UCI Machine Learning Repository, 2011. DOI: https://doi.org/10.24432/C5K89B.

[5] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389*, 2012.

[6] Jakub Breier, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. Deeplaser: Practical fault attack on deep neural networks. *arXiv preprint arXiv:1806.05859*, 2018.

[7] Jakub Breier, Dirmanto Jap, Xiaolu Hou, Shivam Bhasin, and Yang Liu. Sniff: reverse engineering of neural networks with fault attacks. *IEEE Transactions on Reliability*, 71(4):1527–1539, 2021.

[8] Peter Bühlmann and Sara Van De Geer. *Statistics for high-dimensional data: methods, theory and applications*. Springer Science & Business Media, 2011.

[9] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. On evaluating adversarial robustness, 2019.

[10] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks, 2017.

[11] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

[12] Christophe Clavier. Secret external encodings do not prevent transient fault analysis. In *Cryptographic Hardware and Embedded Systems-CHES 2007: 9th International Workshop, Vienna, Austria, September 10-13, 2007. Proceedings 9*, pages 181–194. Springer, 2007.

[13] Joseph Clements and Yingjie Lao. Hardware trojan attacks on neural networks. *arXiv preprint arXiv:1806.05768*, 2018.

[14] Joana C. Costa, Tiago Roxo, Hugo Proença, and Pedro R. M. Inácio. How deep learning sees the world: A survey on adversarial attacks defenses, 2023.

[15] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. Sifa: exploiting ineffective fault inductions on symmetric cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 547–572, 2018.

[16] Federico Girosi, Michael Jones, and Tomaso Poggio. Regularization theory and neural networks architectures. *Neural computation*, 7(2):219–269, 1995.

[17] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129:1789–1819, 2021.

[18] Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Evaluating backdooring attacks on deep neural networks. *IEEE Access*, 7:47230–47244, 2019.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[20] Kevin Hector, Pierre-Alain Moellic, Mathieu Dumont, and Jean-Max Dutertre. Fault injection and safe-error attack for extraction of embedded neural network models. *arXiv preprint arXiv:2308.16703*, 2023.

[21] Nishad Herath and Anders Fogh. These are not your grand daddys cpu performance counters–cpu hardware performance counters for security. *Black Hat Briefings*, 47:48, 2015.

[22] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[23] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with

low precision weights and activations. *journal of machine learning research*, 18(187):1–30, 2018.

[24] Shruti Jadon. Introduction to different activation functions for deep learning. *Medium, Augmenting Humanity*, 16, 2018.

[25] Rudolf Kruse, Sanaz Mostaghim, Christian Borgelt, Christian Braune, and Matthias Steinbrecher. Multi-layer perceptrons. In *Computational intelligence: a methodological introduction*, pages 53–124. Springer, 2022.

[26] Max Kuhn, Kjell Johnson, et al. *Applied predictive modeling*, volume 26. Springer, 2013.

[27] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. Fault injection attack on deep neural network. In *2017 IEEE/ACM International Conference on Computer-Aided Design ICCAD)*, pages 131–138, 2017.

[28] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018.

[29] Zizhen Liu, Jing Ye, Xing Hu, Huawei Li, Xiaowei Li, and Yu Hu. Sequence triggered hardware trojan in neural network accelerator. In *2020 IEEE 38th VLSI Test Symposium (VTS)*, pages 1–6. IEEE, 2020.

[30] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[31] Shike Mei and Xiaojin Zhu. Using machine teaching to identify optimal training-set attacks on machine learners. In *Proceedings of the aaai conference on artificial intelligence*, volume 29, 2015.

[32] Milad Nasr, Nicholas Carlini, Jonathan Hayase, Matthew Jagielski, A. Feder Cooper, Daphne Ippolito, Christopher A. Choquette-Choo, Eric Wallace, Florian Tramèr, and Katherine Lee. Scalable extraction of training data from (production) language models, 2023.

[33] Seong Joon Oh, Max Augustin, Bernt Schiele, and Mario Fritz. Towards reverse-engineering black-box neural networks, 2018.

[34] Daryna Oliynyk, Rudolf Mayer, and Andreas Rauber. I know what you trained last summer: A survey on stealing machine learning models and defences. *ACM Computing Surveys*, 55(14s):1–41, July 2023.

[35] Pranav Rajpurkar, Emma Chen, Oishi Banerjee, and Eric J Topol. Ai in health and medicine. *Nature medicine*, 28(1):31–38, 2022.

[36] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1157–1174. IEEE, 2022.

[37] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1211–1220, 2019.

[38] Frank Rosenblatt et al. *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*, volume 55. Spartan books Washington, DC, 1962.

[39] Nadir K. Salih, D Satyanarayana, Abdullah Said Alkalbani, and R. Gopal. A survey on software/hardware fault injection tools and techniques. In *2022 IEEE Symposium on Industrial Electronics Applications (ISIEA)*, pages 1–7, 2022.

[40] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. *Advances in neural information processing systems*, 31, 2018.

[41] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[42] M. Caner Tol, Saad Islam, Andrew J. Adiletta, Berk Sunar, and Ziming Zhang. Don't knock! rowhammer at the backdoor of dnn models, 2023.

[43] Marian Verhelst and Bert Moons. Embedded deep neural network processing: Algorithmic and processor techniques bring deep learning to iot and edge devices. *IEEE Solid-State Circuits Magazine*, 9(4):55–65, 2017.

[44] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning, 2019.

[45] Liang Xiao, Xiaoyue Wan, Xiaozhen Lu, Yanyong Zhang, and Di Wu. Iot security techniques based on machine learning: How do iot devices use ai to enhance security? *IEEE Signal Processing Magazine*, 35(5):41–49, 2018.

[46] Qixue Xiao, Kang Li, Deyue Zhang, and Weilin Xu. Security risks in deep learning implementations. In *2018 IEEE Security and privacy workshops (SPW)*, pages 123–128. IEEE, 2018.

[47] H. Yu, H. Ma, K. Yang, Y. Zhao, and Y. Jin. Deepem: Deep neural networks model recovery through em side-channel information leakage. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 209–218, Los Alamitos, CA, USA, dec 2020. IEEE Computer Society.

[48] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9):2805–2824, 2019.

[49] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[50] Chen Zhu, W Ronny Huang, Hengduo Li, Gavin Taylor, Christoph Studer, and Tom Goldstein. Transferable clean-label poisoning attacks on deep neural nets. In *International Conference on Machine Learning*, pages 7614–7623. PMLR, 2019.