POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

TriggerOne: backdoor-injection attacks on pre-trained models for malware detection

Tesi di Laurea Magistrale in
Computer Science Engineering - Ingegneria Informatica

Author: **Federico Di Cesare**

Student ID: 944843
Advisor: Prof. Stefano Zanero
Co-advisors: Michele Carminati, Mario D'Onghia, Mario Polino
Academic Year: 2020-21

Abstract

With the advent of deep neural networks and their application to the field of malware detection, a new powerful tool able to analyze binary files and detect malicious behavior has been introduced. However, neural networks have been proven to be susceptible to different attacks, such as evasion attacks, training set poisoning attacks, and backdoor injection attacks. This thesis focuses on the backdoor injection attacks on pre-trained malware detection models. This attack strategy affects publicly available pre-trained models, which are modified to embed a backdoor, namely a hidden functionality that allows an arbitrary network output whenever the submitted input contains a specific pattern, the trigger. Similar attacks have been performed to backdoor famous models in the field of computer vision; however, to the best of our knowledge, no pre-trained malware detection neural network has ever been attacked with similar techniques. The domain shift is not trivial, as neural networks designed for computer vision tasks are radically different from neural networks for malware detection. We test and adapt to the new domain three attack strategies, specifically targeting MalConv, a convolutional neural network for malware detection. We propose the model updating attack, in which we re-train the pre-trained model with trigger-poisoned data, the weights perturbation attack, in which we analyze the model and carefully modify certain neurons to inject the backdoor, and the subnet replacement attack, in which we train a small neural network which is then injected in the original pre-trained model and activates whenever the input contains the trigger. We also test four possible defense strategies that a victim might adopt to detect and even remove an injected backdoor. Our model updating attack and subnet replacement attack achieved a backdoor success rate of 97%, while the weights perturbation scored 91% on a poisoned test set. Our attacks outperformed existing evasion attacks on MalConv and obtained comparable results to similar attacks on computer vision models.

Keywords: deep learning, pre-trained models, backdoor attack, malware detection, MalConv

Abstract in lingua italiana

Con l'avvento delle reti neurali, e la loro applicazione nel campo della malware detection è stato introdotto un nuovo e potente strumento, in grado di analizzare file binari e identificare attività malevole. Le reti neurali, nonostante le loro infinite potenzialità, sono suscettibili a diversi attacchi, come attacchi evasivi, attacchi di contaminazione del training set e attacchi di inserimento di backdoor. In questa tesi, ci concentriamo sugli attacchi di inserimento backdoor a modelli pre-allenati per la malware detection; questo attacco colpisce modelli pre-allenati pubblici, che vengono modificati per ospitare una backdoor, ovvero una funzionalità nascosta che fa produrre output arbitrario al modello nel caso l'input contenga una sequenza specifica: il trigger. Attacchi simili sono stati eseguiti su modelli nel campo della computer vision, ma riteniamo di essere i primi a proporre attacchi di inserimento backdoor a modelli pre-allenati per malware detection. Questo adattamento di dominio non è banale, considerato che le reti neurali studiate per la computer vision sono radicalmente diverse da quelle per la malware detection. In particolare, adattiamo a questo nuovo dominio tre attacchi, concentrandoci su MalConv, una rete neurale convoluzionale per la malware detection. Proponiamo i seguenti attacchi: il model updating, nel quale ri-alleniamo il modello pre-allenato con nuovi dati contaminati con il trigger, il weights perturbation nel quale analizziamo il modello e modifichiamo alcuni neuroni per inserire la backdoor e, infine, il subnet replacement, nel quale alleniamo una piccola rete che viene inserita nel modello originale, la quale si attiverà ogni qualvolta l'input contenga il trigger. Proponiamo anche quattro possibili difese che una vittima potrebbe utilizzare per individuare e fermare una backdoor. I nostri attacchi di model updating e subnet replacement ottengono una sensibilità del 97% sui test set con trigger, mentre il nostro attacco di weights perturbation arriva al 91%. I nostri attacchi mostrano risultati migliori degli attacchi evasivi già esistenti su MalConv e risultati comparabili ad attacchi simili su modelli per il computer vision.

Parole chiave: deep learning, modelli pre-allenati, attacchi backdoor, malware detection, MalConv

Contents

List of used acronyms and abbreviations

ANN artificial neural network. 5, 12, 13

API application programming interface. 51

CNN convolutional neural network. 8, 25, 27, 41, 54

CPU central processing unit. 63

CV computer vision. 2, 10–12, 16, 20, 29

DL deep learning. 1, 5, 7–9, 17, 48, 89

DNN deep neural network. 1, 8, 11, 84

GAN generative adversarial network. 23

GPU graphic processing unit. 8, 9, 63

K-NN K nearest neighbors. 7

ML machine learning. 7, 8, 51

NLP natural language processing. 20

NN neural network. 1, 8, 14, 15, 21, 23, 29, 87, 93

PE portable executable. 17, 18, 25, 27, 53, 54, 56, 63, 93

PSO particle swarm optimization. 30, 33, 70, 72

SVM support vector machine. 7

1 | Introduction

1.1. Domain contextualization

The problem of malware detection is one of the biggest challenges for researchers and professionals all around the world. The techniques involved in the complex process of determining whether an executable is a malware or a goodware are many and they have been evolving through the decades. Nowadays more and more systems are exploiting the great capabilities of artificial intelligence, in particular the field of deep learning (DL).

1.2. Are DNNs vulnerable?

The field of deep learning is rather young, it is growing quickly and, similarly to all the new technologies, the community has not set in stone yet the *best "safe" practices* to handle the predictive models, the datasets, an so on. There are two main problems with deep neural network (DNN) that make many attacks possible. Firstly, as training from scratch a deep neural network is very expensive in terms of computing resources, untrusted models or datasets may be re-used, compromising the safety of the model. Moreover, deep neural network internal weights are *grey* to a human inspection, as there is no easy way to understand what a network is doing by only looking at the them; hence, when a human looks at the weights of a neural network, they cannot detect eventual malicious behavior. As a consequence, a number of attacks can be carried out on deep neural networks; in this thesis, we will focus on backdoor injection attacks on pre-trained models.

1.3. Backdoor injection attacks

A backdoor injection attack on a pre-trained model aims at modifying the model behavior when an input sample is poisoned by the attacker. In particular, a poisoned model should behave normally when a clean input sample is submitted, while showing an arbitrary behavior when it gets fed a sample containing a specific pattern, the *trigger*; this type of input samples are called *poisoned sample*.

1.4. State-of-the-art attacks

Existing attacks to malware detection models mainly consist in *evasion* attacks [9, 19, 35], where the attacker fools the attacked model by modifying *ad-hoc* each malware sample; the model will classify the modified malware samples as goodware. Other works [37, 38] proposed attacks on malware detection models by poisoning the training set: the model is trained using legitimate data, as well as additional poisoned data capable of injecting the malicious behavior. On the other hand, backdoor injection attacks on pre-trained models have been explored by [8, 10, 14, 30, 31, 43]. These works propose attacks against well-known models such as VGG [40] and ResNet [13], which all belong to the domains of face recognition and hand-written digits recognition. To the best of our knowledge, no existing works address the problem of injecting a backdoor injection into pre-trained models for malware detection.

1.5. Domain shift to the malware detection task

The aim of this work is to perform backdoor injection attacks into a pre-trained model for malware detection. This will involve a domain shift of the existing techniques from the field of computer vision (CV), to the field of malware detection. The task is not trivial, since the aforementioned works attack models which are radically different from state-of-the-art malware detection models, such as MalConv [33]. We specifically attack the MalConv implementation coded and trained by the Ember team [2]. The attacks we propose are:

- **Model updating attack**, in which the backdoor is injected by retraining the pre-trained model using new poisoned data;
- **Weights perturbation attack**, in which the analysis and manual modification of the internal weights of the pre-trained model allows us to inject the backdoor;
- **Subnet replacement attack**, in which the injected backdoor is a small subnetwork which is injected inside the pre-trained model.

1.6. Results obtained

We tested the three attack methodologies on MalConv without any success, realizing that very different model architectures must be attacked in different ways: MalConv is very shallow and utilizes large convolutional filters which require customized approach. Moreover, different optimization techniques were tried out to improve the performance of

our attacks, namely Particle Swarm Optimization, greedy algorithm and gradient descent. All these techniques allow for an optimal selection of the bytes to use as a trigger for the poisoned samples. We then developed and applied the filter alignment technique, for which we inject triggers in the binary files always at the same relative position with respect to the model filters. The filter alignment technique improved the model updating attack, obtaining 97% backdoor success rate. Moreover, the filter alignment allowed us to carry out the weights perturbation and subnet replacement attacks, obtaining respectively 91% and 97% backdoor success rates. All the three attacks did not decrement significantly the model accuracy on clean samples. Finally, we tested four possible defense strategies: accuracy check, network pruning, statistical analysis and transfer learning. Accuracy check is not able to detect malicious activity on our attacks, network pruning manages to block both weights perturbation and subnet replacement, statistical analysis detects the malicious weights in the weights perturbation attack, while transfer learning only manages to weaken the model backdoored through the subnet replacement.

1.7. Contributions

The contributions of this work are:

- We tested the model updating, weights perturbation and subnet replacement attacks;
- We improved the attacks by optimizing the trigger;
- We modified the attacks to make them possible on the peculiar architecture of MalConv;
- We adopted the filter alignment technique, which enables the attacks on models with large filters;
- We produced an experimental validation of the improved attacks;
- We tested four possible countermeasures to our attacks: accuracy check, statistical analysis, network pruning and transfer learning;
- We discussed the limitations of the attacks, highlighting possible future improvements.

2 | Motivation

2.1. Problem statement

The problem of deciding whether a software sample is goodware or malware is, in most cases, a non-trivial task. There exist commercial off-the-shelf antiviruses that automatically scan every file in the hard drive of a computer, reporting any malware they found. However, the problem has not been *"solved"* yet, as no one has discovered the universal formula to detect malware. Researchers are always looking for newer and better ways to analyze software, and, in the recent years, several solutions based on deep learning (DL) have been proposed with interesting results [1, 22, 26]. However, there are new problems that are introduced by the application of deep learning in this domain: how can I know whether the neural network itself is not malicious? In this thesis, we will demonstrate that even famous pre-trained neural networks for malware detection can be attacked; in particular, we show that it is possible to inject backdoors in them, starting from a pre-trained model distributed over the Internet, without influencing the original training process.

2.2. Malware detection basics

Before delving into problem of injecting a backdoor in an artificial neural network (ANN), we should present the current (and most diffused) malware analysis techniques which do not require the usage of deep learning.

2.2.1. Static analysis and dynamic analysis

The techniques adopted to analyze a piece of software can be divided into two main categories: static and dynamic analysis.

- With **static analysis** we take the binary sequence of the piece of software we want to analyze and we look through the bytes, trying to detect well known patterns, dangerous instructions, system calls, fingerprints and so on.

- With **Dynamic analysis** we execute (often in a protected environment) the piece of software in order to analyze its behavior and see if it performs any malicious activity.

In this work, we will focus on the static analysis approach.

2.2.2. Common malware detection techniques

Commercial off-the-shelf software products offer a malware detection service which uses both static and dynamic analysis. Common techniques are [4]:

- **Signature-based detection**: the anti-malware tool looks for digital footprints in the binary of the file. This footprint is usually a collection of byte strings plus some other metrics, like section offsets, dimensions and so on. The anti-malware then interrogates a local or remote database in order to verify if this signature has already been tagged as malicious.
- **Heuristics-based detection**: the anti-malware tool assigns a suspicion score to the piece of software based on some characteristics of the file. For example camera access, hard drive manipulation or import of some specific libraries.
- **Sandbox**: the suspicious file is ran in a constrained and secure environment to detect malicious activity. The anti-malware tool can now monitor the network traffic generated, the API calls or new files written in memory. If something "bad" happens, or undesired network traffic is noted, the file is tagged as malicious.

Every technique has its own downsides: signature-based detection may fail if the attacker crafts a slightly modified version of the malware, while heuristics-based detection excels at finding out if two pieces of software are indeed the same even if there are some code-level modifications. However, attackers found out that through obfuscation both these techniques will fail to detect malware. Common techniques are malware encryption, dead-code insertion, register reassignment, subroutine reordering, instruction substitution, code transposition and code integration [47]. As dynamic analysis aims at detecting malicious activity at runtime, Sandbox detection is often used along with standard static analysis.

All the aforementioned techniques are overall reliable, but not perfect. The key point is that the malware detection problem is like an everlasting battle between the red side (malware authors) and the blue side (cybersecurity analysts), where the two parts are human and their tools are also crafted by humans. A way the blue side can better defend itself is by exploiting the computational power given by machine learning/deep learning algorithms. In other words, human against human is somehow a balanced fight, but

human against machine could be won by the blue side machine. A deeper review on advantages and disadvantages of adopting ML/DL for the malware detection task can be found in [1, 39].

2.3. What is Machine Learning and why it can be useful

Formally, machine learning (ML) is a branch of Computer Science that makes use of algorithms and statistical models to "teach" a program how to solve a specific task. The program uses data and examples to tune itself and increment step-by-step its accuracy towards the problem solution. This definition suggests us that we can take a machine learning model and teach it to recognize malware.

To make predictions, a machine learning model requires a set of *features* to work on. Features are elaborated pieces of information regarding the sample the model is predicting. For example, if a model wants to analyze a flower, features could be the length and width of the petals and the height of the stem. However, extracting features from software is more complex than extracting them from a flower. An important example can be found in [2], where characteristics of a binary such as file size, executable headers, imported libraries and so on are used as features for a ML classifier.

Once we have the features, we can train a ML model. There are many algorithms used to solve this task, the most common are SVM, Decision Trees, K-NN, Naive Bayes, Random Forests and variants [16, 18, 28].

Why is machine learning so powerful? As we mentioned before, common malware detection techniques are created by humans, and the human is somehow limited by its own mind, creativity and experience. Signatures and heuristics are extracted (or coded to be extracted) by human analysts, sandbox surveillance is coded by a human, but machine learning learns to detect malware in a non-human way. At training time, a machine learning model is only told whether it is predicting correctly or erroneously, it is not told how to make predictions. We could say that machine learning can learn to "think" outside the box.

2.4. What is deep learning and why it can be even better than machine learning

Machine learning still relies on human experience. The feature extraction process is made by humans. The feature extraction process is itself a field of study, there are researches studying how to select the most relevant features out of a dataset. If the goal of the defender is to remove completely the human limitations from the malware detection process, it must remove also the feature extraction process. deep learning (DL) does exactly this.

Deep learning is a sub-branch of machine learning which works with raw data. A deep neural network (DNN) works directly on a raw unmodified binary both for training and prediction. Many DL techniques use convolutional neural networks on images made by converting the file bytes to a grayscale image [5, 7, 26]. Other works also use CNN but with 1-dimensional convolutions and without performing any conversion on the input data [33]. Additional information on neural networks and deep learning will be given in Chapter 3.

2.5. Poisoned models threat

This thesis focuses on the deep learning approach, and we will show that there are some threats when approaching the malware detection problem with it.

2.5.1. Where is the vulnerability?

In a scenario where I construct my dataset and I train my own network, I do not incur any security issue. I am sure that the network does exactly what I trained it for and the results on the test set allow me to have a reliable evaluation of the model performances. There is one problem though, training a DL model is extremely expensive. In order to efficiently train a neural network, a researcher needs powerful GPUs as they allow parallel computation and the evaluation of multiple training samples at once. Even with this powerful hardware, the training process will take days, even months to complete, depending on the complexity of the network and the size of the dataset. Since most of the people interested in deep learning cannot afford to spend so much time and/or money on training, a very common practice is constituted by the so called *Transfer Learning*. Transfer learning works by downloading from the web a pre-trained neural network (usually trained to perform a similar task) and partially re-training it with a second dataset

to adapt the model to the new task. This operation takes significantly less time which makes it very popular since it gives access to deep learning also to those who cannot afford expensive GPUs. In this scenario, the researchers who download the pre-trained model are trusting its source, but can they? One might argue that this is an everyday problem when downloading files from the internet; however, it is more subtle than that. When I download something not trustworthy, this may in truth be malware. In fact, I can somehow still detect if the file I downloaded is safe or not and this information is right inside the file I downloaded, by analyzing the file itself with an antivirus program. On the other hand, when I download a pre-trained model I do not get software that may hide malware, I get a sequence of weights and biases (numbers, that is) which are grey to both humans and antivirus software.

2.5.2. The pre-trained Neural Network Backdoor attack

A Backdoor attack aims at injecting a neural network model with a *backdoor*. A backdoor is a portion of the network which activates only when the input of the network has a pre-defined sequence, called a *trigger*. A backdoored (or poisoned) model behaves normally when clean samples are used as input, but behaves arbitrarily when a sample with the trigger (which we will call a *poisoned sample*) is used as input. Note that, as the attack is carried out on a **pre-trained** model, the attacker is advantaged as they do not have to train the model from the beginning.

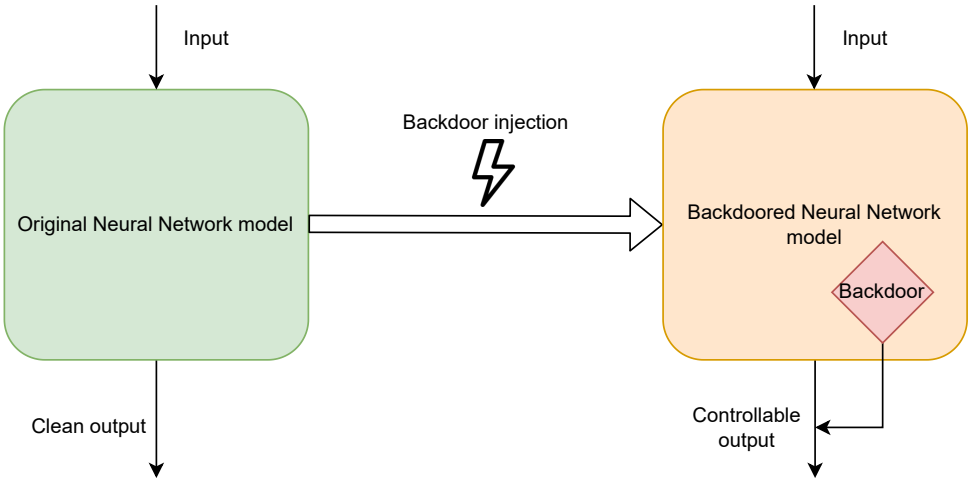

Figure 2.1: Overview of a backdoor attack: starting from a published pre-trained model, the backdoor injection process embeds a hidden functionality in the poisoned model. The backdoor is then capable of manipulating the backdoored model's output.

A general attack pipeline can be summarized as follows:

- The attacker selects a pre-trained victim model to backdoor (often a famous one, like VGG, Resnet, etc...);
- The attacker injects a backdoor in the model;
- The attacker publishes the model online, pretending to offer an improved version of the original model;
- A victim downloads the backdoored (or poisoned) model and uses it for prediction or transfer learning.

The result is that the victim has a poisoned model which behaves normally with standard test datasets; thus, the victim will not suspect anything. However, when the poisoned network is fed with a poisoned sample, the outcome is arbitrated by the attacker.

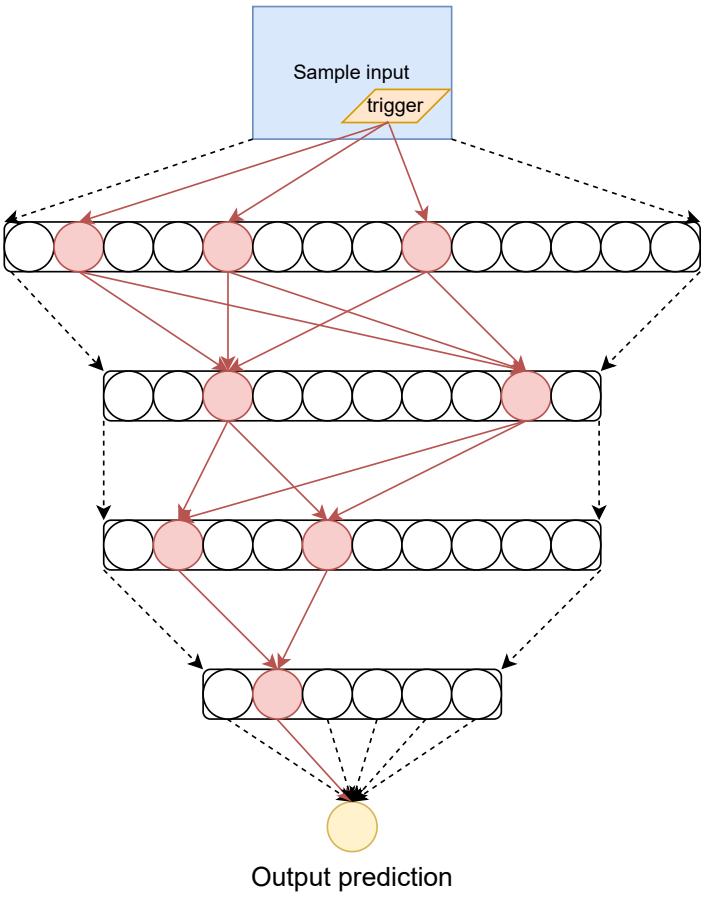

Figure 2.2: Example of a poisoned neural network: the red circles are the backdoor neurons, which react to the presence of the trigger in the input sample.

Proof of concept of this type of attack is given by [8, 10, 30] where we can see how the most common type of attacked networks are computer vision (CV) ones.

2.5.3. Brief problem formalization

Now that we have explained what are the existing vulnerabilities, we can formalize the problem we want to solve for our attack to be successful.

We are given a model $M(w)$ with pre-trained weights W, a set of inputs $\boldsymbol{X}$ and an accuracy $acc = f(M(W), \boldsymbol{X})$. Starting from $\boldsymbol{X}$, we generate a set of poisoned samples $\boldsymbol{X_p}$. Every poisoned sample $x_p \in \boldsymbol{X_p}$ is labeled with target class y_t, and it is generated from a clean sample $x \in \boldsymbol{X}$ labeled with the original class y.

Our goal is to find a modified set of weights W_p such that we maximize the poisoned samples accuracy:

$$\max_{W_p}(f(M(W_p), \boldsymbol{X_p}))$$

with the constraint of not significantly decreasing the model's accuracy on the clean input set:

$$acc - f(M(W_p), \boldsymbol{X}) < t_{acc}$$

where t_{acc} is an arbitrary small threshold factor.

2.6. Goals and Challenges

The goals of this thesis are the following:

- Analyze the state of the art concerning backdoor attacks against neural networks and further study techniques that are, at least in theory, applicable to deep neural networks for malware detection;
- Select among all techniques those that could be applied in the domain of malware detection;
- Modify and adapt the identified techniques in order to make them work in the domain of malware detection.

In other words, we propose a domain shift of known techniques from the field of computer vision to the field of malware detection, knowing that this task is not trivial as those techniques need to be adapted to the new type of input data.

There are a number of challenges ahead of us, since CV is a whole different problem than Malware Detection. The most divergent points are:

- Images are to be seen by a human, while software will be run by a machine. A slight modification of a pixel is invisible to the human eye, while the substitution of

a single byte can lead to catastrophic consequences (binary won't execute, binary won't preserve the functionality);

- Two pixels with similar RGB value are similar and considered similar also by the underneath algorithmic process of artificial neural networks, while two bytes whose values are close (e.g., byte 9 and byte 10) may have completely opposite meanings for a computer;
- State of the art neural networks for CV are very different from neural networks for Malware Detection: this will be discussed more in depth in Chapter 3.

3 | Background

3.1. Artificial neural networks

In this section, some background knowledge on artificial neural networks is provided. In a nutshell, a neural network is a collection of connected *neurons*, which together are able to describe very complex non-linear functions.

3.1.1. Overview

The structure of a neuron is heavily influenced by the structure of a biological neuron: it has a number of inputs (in the biological neuron, the dendrites), an activation threshold and finally an output (in the biological neuron, the axion terminal). A graphic representation of the neuron is depicted in Figure 3.1. The neuron sums together all the (weighted) inputs, subtracts the bias and if the result is greater than 1, it is given as output, otherwise the output is 0. Formally, the output of a neuron is described by the following formula:

$$relu(\sum_i (w_i * x_i) - b)$$

where:

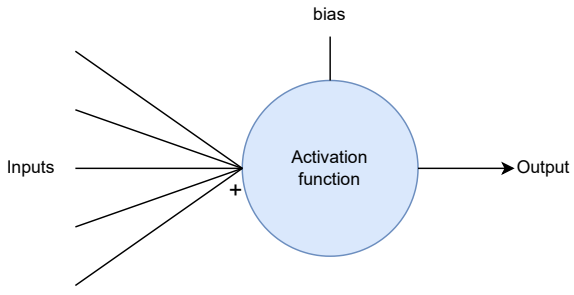

Figure 3.1: The structure of a neuron. The arrows on the left are the inputs (which might come from other neurons), the one on the right the output. The bias is an internal value which controls the activation threshold.

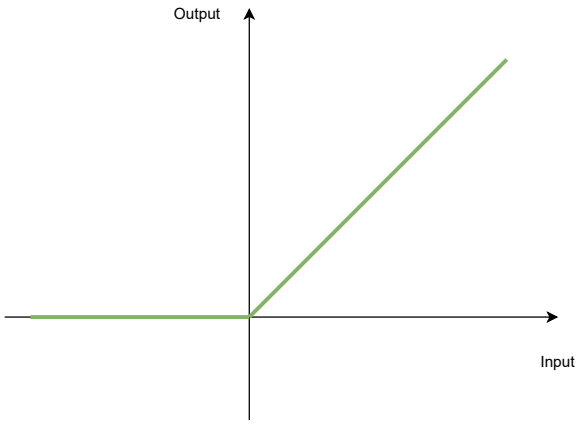

Figure 3.2: The relu function in a Cartesian plane.

- x_i is the i^{th} input;
- w_i is the weight associated to the i^{th} input. It controls "how much important" is the i^{th} input;
- b is the bias, an internal value for each neuron;
- $relu()$ is the activation function. There are many possible activation functions, but the *relu* (Rectified Linear Unit) is the most used.

The *relu* function can be described as $f(x) = max(0, x)$, in other words it keeps the output value as is, only if it is positive; otherwise outputs zero (Figure 3.2).

In the simplest neural network implementation, the neurons are gathered together in *layers*. A neuron at a given layer k gets as input the outputs of all the neurons at the previous layer $k - 1$. The input of the first layer is the input of the network, while the output of the last layer is the output of the network. A visual representation of a basic neural network is depicted in Figure 3.3. The layers described are called *dense* layers, they are the most simple and common layers which can be found in a neural network; however, there are many other types of layer. Later in this section, other types of layer are described.

3.1.2. Learning

As already mentioned, a neural network can describe virtually any non-linear function; this is possible due to the high amount of neurons in the network and the even higher number of parameters. When a neural network learns, it modifies the weights linking the neurons and the internal biases.

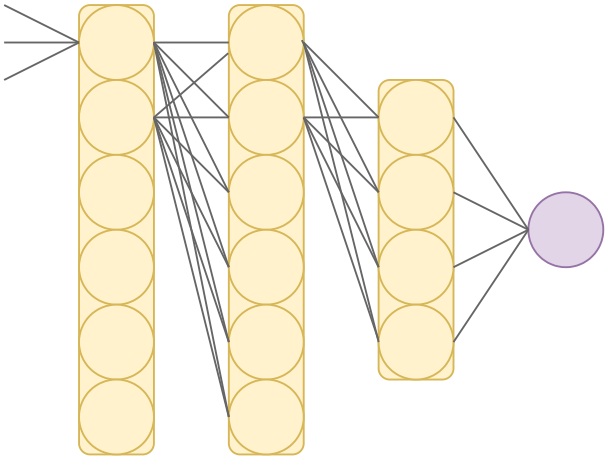

Figure 3.3: A basic neural network structure. For the sake of clarity, only a subset of the edges is shown. In a real network, each neuron of a layer may be linked with all the neurons in the following layer.

The so called *training cycle* teaches the network by challenging it with several examples, contained in a dataset. The network is given a sample input, it computes an output and the result is compared the the expected "ground truth" output. The internal weights and biases are modified so that the next time the network is given the same sample, it outputs a correct result, or at least, a "more" correct result. How much an output is correct, is measured through a *loss function*, which takes as input the predicted output and the real ground truth output, then computes a metric which states how close the two outputs are.

After the neural network has been trained many times with the full dataset, it has learnt the non-linear function desired. The precise algebraic methodology used to modify the weights and biases of a neural network during the training cycle is rather complex, and it is beyond the scope of this thesis.

During the training cycle, it is important to avoid the so called *overfitting*, which is the phenomenon for which the model learns to output correct results for the training set only, and cannot predict correctly when brand new samples are submitted to the network.

3.1.3. Embedding layer

The embedding layer is a peculiar layer which can be found at the beginning of many neural networks. The objective of the embedding layer is to take as input a compact representation of data, and for each datum, output a sparse representation of it.

For example, MalConv [33] uses an embedding layer; it takes as input byte values (integers between 0 and 255) and maps every byte to an 8-dimensional vector, which is the requested

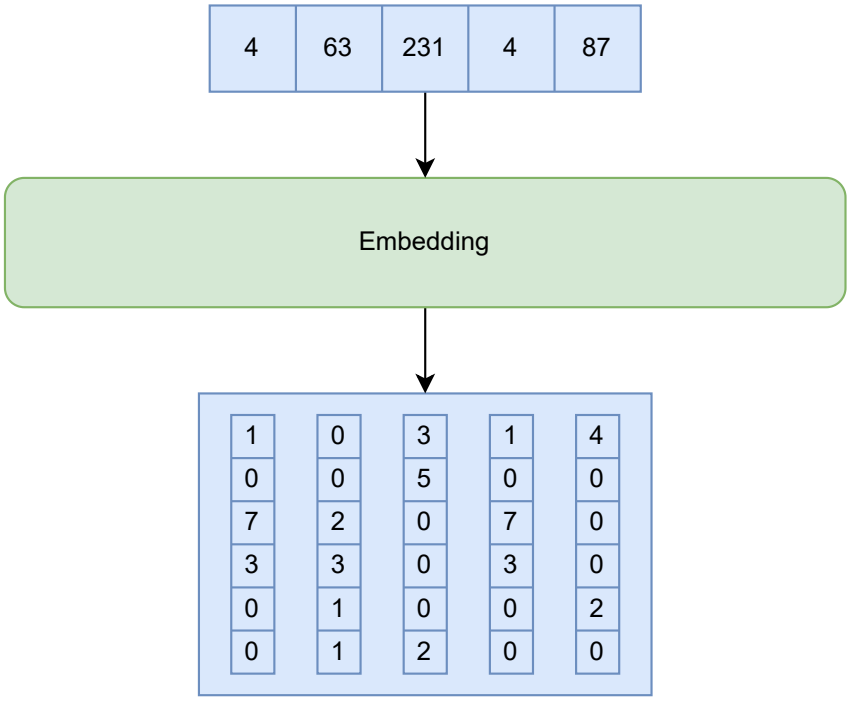

Figure 3.4: Example of the functioning of the embedding layer: every byte is mapped to a representation vector. Notice that the mapping is deterministic, as equal bytes are mapped to equal vectors.

sparse representation. Notice that every time a byte enters the embedding layer, it is mapped to the same 8-dimensional vector. A graphical representation of the embedding layer can be seen in Figure 3.4.

3.1.4. Global-max-pooling layer

A max-pooling layer is often used to reduce the dimensionality of the data passing through the network. In the MalConv implementation attacked in this thesis, a global-max-pooling layer is used; however, for the sake of conciseness, from now on it will be called simply "max-pooling layer". The max-pooling layer takes as input a high dimensional vector and outputs a lower dimensional one, by returning only the maximum element over a certain dimension. For example, in MalConv, it takes as input a 2097x128 matrix, computes the maximum elements over the first axis and outputs a 128 long vector.

3.1.5. Convolutional layer

The convolutional layer is vastly used in computer vision models; it works by "convoluting" different regions of the input vector with a common set of weights, the *filter*. The

filter starts at the beginning of the input vector and progressively slides towards the end of it, each time returning an output value; this way, the output is still a vector. In Section 3.2 the convolutional layer is described more in depth.

3.2. Short recap on convolution math

In order to fully understand the attacks which will be described in Chapter 4, a short recap about convolutions is needed. Only convolutions applied to neural networks are discussed here. The classic convolution is a mathematical tool far more complex but not interesting for this thesis.

The formula for the 1D Convolution is the following:

$$y_i = \sum_{k=i-l}^{i+l} x_k * f_k$$

Where l is the radius of the filter, x is the input array and f is the filter itself.

The convolution is applied to all the elements of the input vector; hence, the output is also a vector. From the formula above, it is evident that when the convolution operates on the first and last l elements, special attention is needed since the index k in the sum falls out of the boundaries of the input vector. Usually, padding is used to convolute these first and last l elements. Another common choice is to avoid the padding and thus ignoring the last elements. DL frameworks use a variant of the convolution in which the filter slides along the input n *by* n steps (the value n is called *strides*). This causes a reduced dimensionality in the output with respect to the input. A visual representation of the convolution operation can be seen in Figure 3.5.

An important property of the convolution is that, given a certain filter, the output is greater where the input array is similar to the given filter. As a consequence, convolutional filters can be used to "look for" similar regions in the input array.

3.3. The Portable Executable format

The portable executable (PE) is a format used in both 32 and 64 bit versions of Microsoft Windows for executables, object files, shared libraries and device drivers.

A PE file consists in a number of headers and sections which instruct the dynamic linker how to map the file in memory. An executable file consists in different section, each section requires different permissions and memory protection; one of the dynamic linker's duties

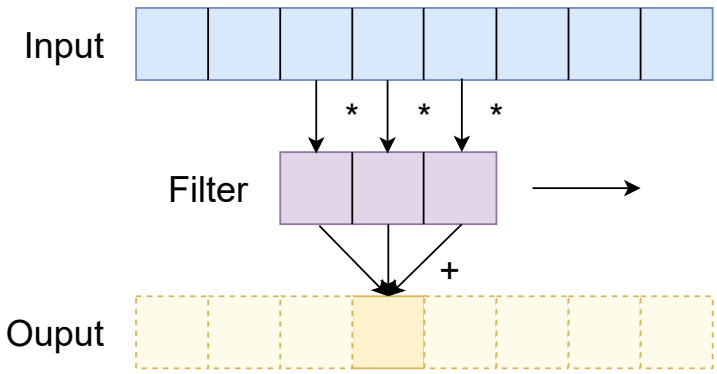

Figure 3.5: The convolution operation: the input signal is in blue, the filter is in purple, while the output of the convolution is in yellow. Note that the filter starts at the beginning of the input array and slides to the right in order to produce an output vector.

is to map every section to its own memory area and to give it the correct permissions, getting the required pieces of information from the PE headers.

From a high level point of view, a PE is composed by the following sections (from low offsets to high offsets):

- **DOS Header** and **DOS Stub**: the DOS header and stub are placed in a PE file only for backwards compatibility. When the PE is executed in MS-DOS, it simply outputs `"This program can not be run in DOS mode"`;
- **COFF Header**: the Common Object File Format header starts with the signature `0X5045` (which are the hexadecimal values for `PE` in the ASCII table). There are also some additional pieces of information like the number of sections in the dedicated table;
- **Optional Header**: despite the name this header must be present in image files, other file types are not required to have it;
- **Section Table**: in the section table all information about the sections is stored; for example `Name` contains the name of the section, `VirtualSize` defines the size of the file when loaded into memory, `SizeOfRawData` defines the size of the file on the disk and `PointerToRawData` points to the first page of the section;
- **Section Data**: these are blocks of bytes containing the actual data of the file: code, strings, resources are stored here.

A detailed diagram of the structure of a PE file is depicted in Figure 3.6

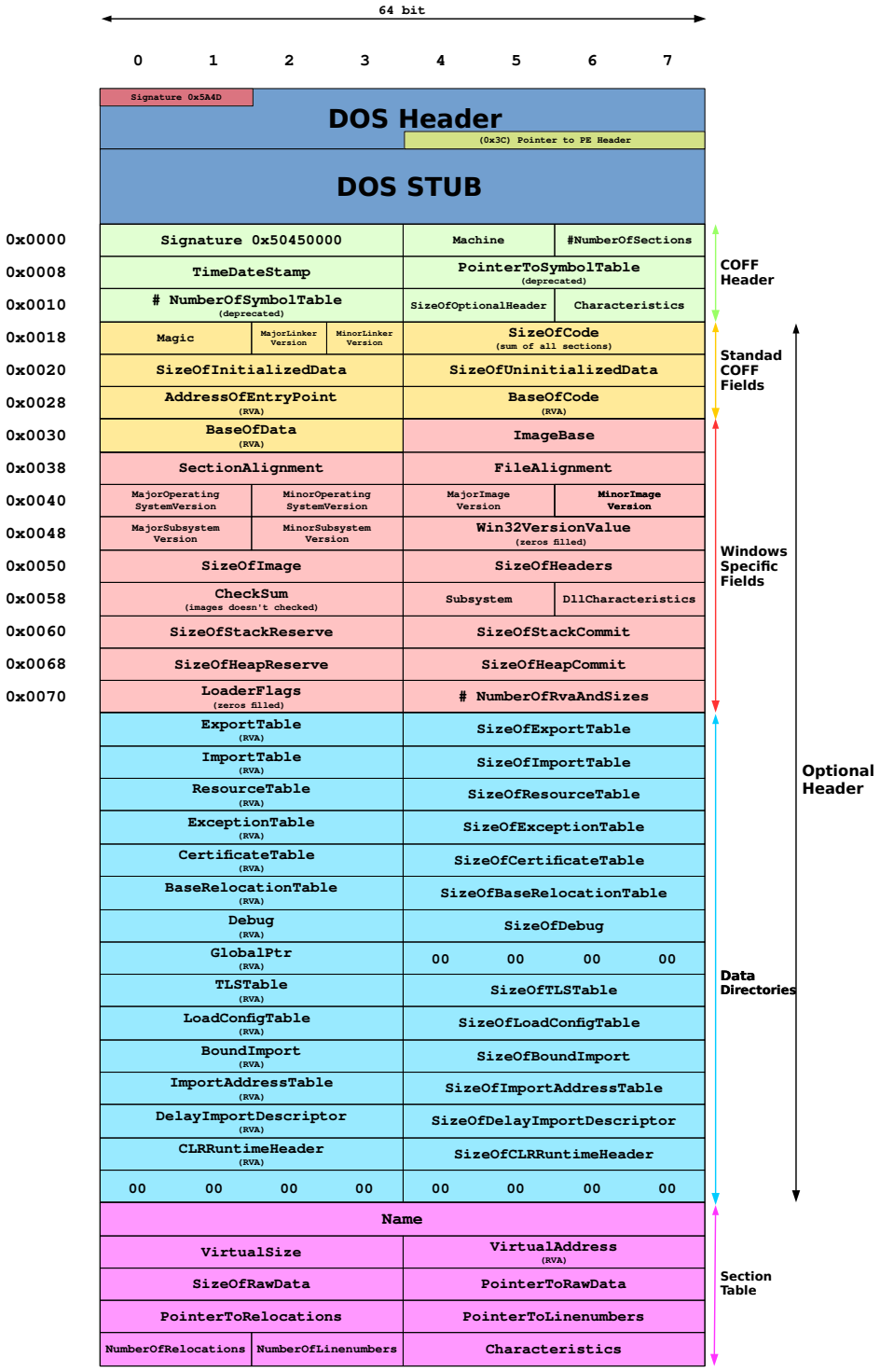

Figure 3.6: The diagram shows the structure of a PE file. Below the Section Table, there is the actual data belonging to the various sections. From: ByteBiter - CC BY 4.0, https://commons.wikimedia.org/w/index.php?curid=39259242

3.4. State of the art

To the best of our knowledge, the techniques which will be described in this section have been tested only on computer vision tasks and natural language processing (NLP). There are other researches who have developed backdoor attacks on neural networks for malware detection, but they didn't attack pre-trained models with backdoor injection. Works like [37, 38] have demonstrated that backdoor attacks are possible through data poisoning: the attacker manages to submit poisoned samples which will be included in the training set. Evasion attacks have been carried out successfully by [9, 35], where they generate adversarial samples to fool malware detection networks. The novel attack introduced in this thesis is a backdoor injection attack on a malware detection pre-trained Model.

In this section it is described the state-of-the-art techniques used for backdoor injection for CV and natural language processing (NLP) networks.

3.4.1. Main attack methodologies

Here we present the three state-of-the-art attack methodologies found in literature; in particular, we present only those that can be transfered to the field of malware detection. Note that these attacks are all methods to find a set of poisoned weights W_p that satisfies the maximization problem introduced in Section 2.5.3. Moreover it is important to notice that the baseline idea behind all these techniques is the same: in many of the known pre-trained models, there is often a subset of weights that do not influence significantly the classification process. We can exploit those weights to carry out our attack, without compromising the accuracy on clean samples.

Model Updating

Model updating is the most simple attack strategy. The attacker simulates a fine-tuning process to inject the backdoor. The attack is composed by the following flow: the attacker attaches a trigger (pixel pattern or byte sequence) to some samples from class A, then labels them as belonging to class B; the attacker then fine-tunes the pre-trained network with the new data. There are, however, some strategies to be discussed before performing a Model Updating attack:

Representation learning If the attacker applies representation learning, the updating process is done against the internal representation of the samples. The attacker starts by

fine-tuning only the first part of the victim network, namely the *feature extractor*. Let us say that for an input X_j, the feature extractor of a neural network at layer K_i produces a representation vector $\boldsymbol{A_j}$. The attacker computes the optimal internal representation for a goodware $\boldsymbol{A_{clean}}$ and uses it as a label to re-train the attacked network. As a consequence the NN learns to represent poisoned samples like goodware ones.

Classification learning This it the most simple approach and can be used as an alternative to Representation Learning, the model updating is done against the final classification label. The attacker takes a malicious sample X belonging to the class $y_{original}$, then produces a poisoned sample X_p and sets its label to y_t, the target class to which we want to classify all poisoned samples.

Dummy samples The most dangerous side-effect of model updating is the loss of functionality due to overfitting to the poisoning data. In the most common scenario for an attack on a pre-trained model, the attacker does not have at its disposal a full training dataset. A reliable workaround for this problem is the generation of *dummy samples*. A dummy sample is an input sample which has been reverse-engineered to be the perfect one for a specific network output. The attacker will use these dummy samples as substitutes of real non-poisoned training samples.

Weights perturbation

The weights perturbation attack aims at injecting the backdoor without any further training, but by carefully analyzing the network behavior and then manipulating the weights by hand in order to enforce the desired backdoor. Examples of this type of attack can be found in [8, 14] The basic idea is to analyze the difference of neurons activation between clean and poisoned samples, then amplify the difference and finally exploit this difference in order to force the network to an arbitrary output in case of poisoned sample.

Subnet replacement

A third, hybrid approach, is called subnet replacement [31]. In this case the attacker trains a small and narrow network to identify the trigger. The only task for the subnetwork is to recognize if the given input is the trigger or not. This subnetwork is then implanted in the attacked model. It is important to point out that the overall performance on clean samples should not be affected significantly since it is supposed that the base model is very big (VGG for example). Note that the subnetwork must have the very same structure of the original one, the only thing that can (and must) change is the width of the network.

Being the subnetwork so small, it should take very little time and a low number of samples to train it. An alternative option for the subnetwork training is to generate some dummy samples both for poisoned class an non-poisoned class.

3.4.2. Auxiliary techniques

In this section, it is discussed a group of auxiliary techniques that improve the performances of the aforementioned attacks. Moreover, we take into consideration in this section only the techniques we could concretely adapt to the malware detection task.

Neuron selection

Neuron selection aims at identifying which neurons can be attacked without compromising the overall model accuracy. Given that all previous works agree on the fact that famous pre-trained models have far more weights than strictly necessary and there are some unused neurons [11, 14, 31, 43], finding these neurons is crucial in order to attack efficiently a pre-trained model. If we manage to detect the neurons which are not used actively by the network, we can use these neurons to inject the backdoor. Since these neurons were not used for the normal network prediction, compromising them will not cause a dramatic drop in accuracy on the clean samples.

The neuron selection can be seen as a knapsack problem, it is required to select the highest amount of neurons without decreasing the model accuracy too much. Formally, given the original model $M(w)$, the set of weights W, the set of inputs $\boldsymbol{X}$, the model accuracy $f(m, x)$, the maximum accuracy drop t_{acc}, the objective is to find the biggest set W_{unused} such that $f(M(W), \boldsymbol{X}) - f(M(W - W_{unused}), \boldsymbol{X}) < t_{acc}$.

A common heuristic for neuron selection is the activation rate on clean samples: neurons which remain silent when the model is predicting clean samples are easier to exploit, given that a change in behavior will not affect significantly the classification outcome. Variants of this technique are used in [14, 43]. Other techniques are possible: neural gradient ranking is used in [34], a "connectivity" metric is used instead in [24].

Static vs Dynamic trigger

A static trigger is the same for every sample involved in the attack. The backdoored network learns to recognize that specific trigger and will work only with that trigger. A dynamic trigger [29] instead, is computed every time and it is different for every input. The most common setting of this approach is the coexistence of two networks: the backdoored

network and a generative network that generates the sample-specific trigger. There is an obvious downside for a static trigger methodology: detection can be simpler. If a victim uses some form of backdoor defense on training data, the first thing they will come up to is a way to detect triggers. It can be done in a naïve way by looking for similar regions in the inputs. On the other hand, a dynamic trigger is safer, but more difficult to implement. It is important to point out that this thesis focuses on attacking pre-trained models, hence the victim will not have the training data at its disposal. In this scenario the two trigger types are equivalent.

Trigger generation

An accurate trigger generation can improve significantly the performances of the attack. The idea is that if an attacker uses an arbitrary trigger, it must adapt the network to the specific trigger in order to complete the attack. If the attacker generates and optimizes the trigger, it does not have to adapt the neural network to the trigger, since a generated trigger is designed to work with that specific network. Many papers stress the necessity of a form of trigger optimization or generation. A way to generate the trigger could be through the use of generative adversarial network (GAN) similar to the ones proposed in [15]. Others [20, 24] propose more traditional approaches like gradient optimization. Since one of the possible defenses against poisoned samples is to pass them through an autoencoder, a possible strategy could be to generate a trigger to be resilient to autoencoder based defenses [43].

Trigger generation, in a similar fashion to the model updating attack, can be done both against the classification label, or against an internal representation [46].

3.4.3. Defenses

These are the backdoor defenses analyzed by the authors of the state-of-the-art papers [23]. In this thesis I will report only the defenses that analyze the neural network itself. Other techniques analyze the input data, but are not relevant for our study.

Accuracy test

This is the most basic and straightforward approach to test whether a model does what it claims to do. The defender can gather a small test set and compare the results to those found in the original paper of the neural network model. The rationale behind this defense is that if someone wants to publish a set of pre-trained weights for a model, those weights should at least give an accuracy comparable to the one claimed in the original paper,

otherwise it would not make sense to adopt those weights in the first place. However, being it a simple defense strategy implies that it is also easy to circumvent. Indeed, for all attacks it is considered a hard constraint to achieve high clean samples accuracy.

Network Pruning

Network pruning defense works by "eliminating" from the network inactive neurons, hoping to delete poisoned ones [23]. In a nutshell the defender submits to the network a small dataset of clean samples, records the activations of the neurons and selects those who remain silent most of the times. These selected neurons are silenced until the accuracy drops below a certain threshold. The rationale behind this defense is the same behind a backdoor injection attack and that's why it can be so powerful. If a neuron remains silent for most of the input samples, it's influence on the overall accuracy of the network should be low, so eliminating it will not cause substantial damage. At the same time, since it is not so important for the basic network functionality, it could be one of those used by an attacker, since also the attacker looks for dormant neurons.

Transfer Learning

Transfer Learning is not an intentional defense, it is a consequence of a very common training pipeline. A person might download a pre-trained neural network for two reasons: to use it *as is*, or to perform transfer learning. Transfer learning aims at training a model for a very specific task, starting from a pre-trained model which is trained on a more generic task. The most basic technique to perform transfer learning consists in removing the last part of the pre-trained neural network (the classifier) and "freezing" the remaining layers (the feature extractor). The missing classifier is replaced with an un-trained classifier which is then trained once it is attached to the pre-trained feature extractor. After transfer learning, it may happen that the backdoor is *washed out*, note that in some cases this is inevitable, since a sufficiently long transfer learning is equal to training from scratch a new model [14, 21].

Statistical Analysis

Statistical Analysis defense [14] relies on the fact that if an attacker tampers with the weights of a model (manually, without any further training involved), it is very hard to do so in a stealthy manner, without compromising the overall weights distribution. A victim, might analyze the weights of the backdoored model, layer-wise, by running an outlier detection algorithm. The outlier detection algorithm reports suspect weights to

the victim who might decide to reject the model. However, common outlier detection rules, such as the Interquantile Range Method, will report a small amount of genuine weights as "outliers", and it is up to the victim to manually run through the values and decide whether to reject the model. It is evident that in some cases this defense can prove itself powerful, but in many situations does not provide a reliable way to clearly state if a given model is malicious or not. Examples of outlier detection algorithms are Standard Deviation method, or Interquantile Range method.

3.5. MalConv

In this section, it is explained the peculiar structure of the Malconv neural network, first proposed in [33], highlighting the differences between Malconv and the most famous convolutional neural networks.

MalConv is considered state-of-the-art in the Malware Detection task with Deep Learning. It is often used as a benchmark to measure the effectiveness of new neural networks. It is important to say that, although MalConv is thought for and trained on portable executable (PE) files, the network can in principle generalize to other formats (such as ELF); however, this is beyond the scope of this thesis.

3.5.1. Structure

The diagram of the MalConv structure is shown in Figure Figure 3.7, we can already see that the high-level design does not follow the current fashion of designing very deep and complex neural networks. Indeed, MalConv is only 4 layers deep and has approximately 1-2 millions trainable parameters.

Authors provided strong motivations for this design which are reported here:

- **Raw bytes input** - Many modern CNN approaches to Malware Detection with Deep Learning make use of 2D Convolutions an thus need a 2D input [41, 45, 48]. Usually a 2D input is built by taking a binary file and then mapping every byte to its corresponding grayscale value. These grayscale values become pixels and are used to build a 2D image. Here lies a problem: how to set the width of the image? Moreover this type of representation implies a spatial correlation that does not really exist. Adopting a 1D convolution allows the input to be 1-dimensional and more "natural".
- **Initial embedding layer** - Without an embedding layer, due to the math behind convolutions, the network would think that similar byte values are somehow corre-

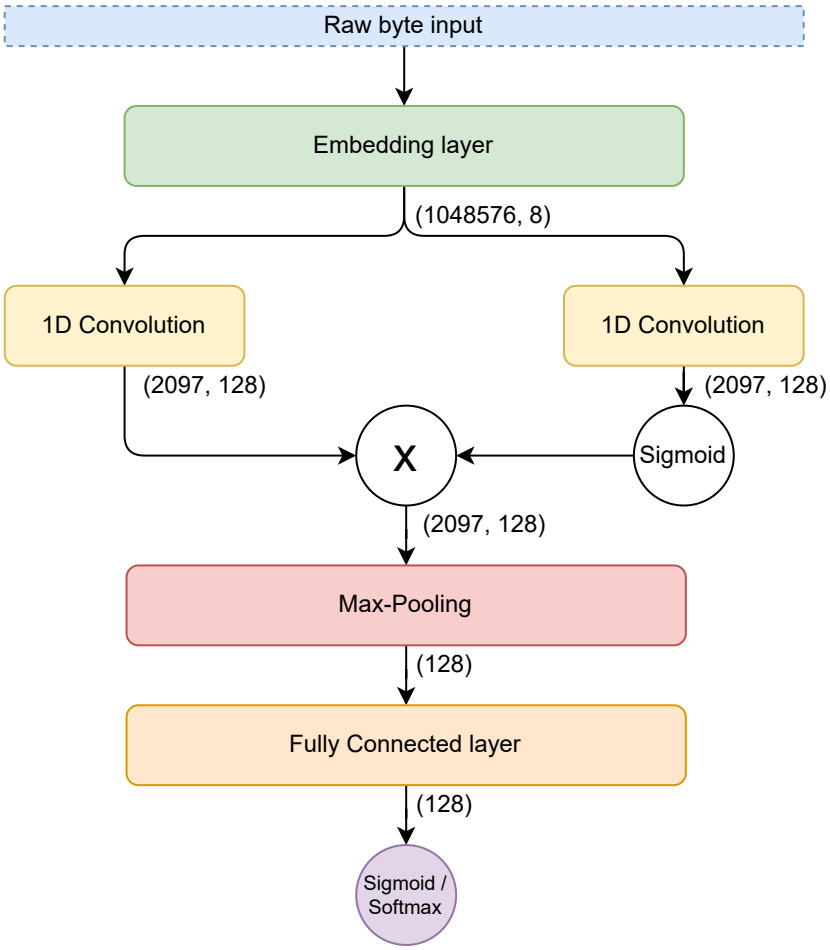

Figure 3.7: High-level structure of the Malconv model. The labels at the output of the layers refer to the output shape in the Ember implementation.

lated, which we know a priori it is not true. The meaning of a byte in a PE file depends on the context: a byte can be part of an instruction, a data string, a flag, and so on. MalConv has a trained embedding layer at the very beginning of the network, which for every byte value outputs an 8-dimensional feature vector.

- **Convolutions with MaxPooling** - Many interesting features of PE files are location independent. The only fixed part is the MS-DOS Header which ends with a pointer to the PE Header. Sections, pointers, data, code are all position independent. To better catch this high level position independence, MalConv uses 1D Convolutions with a MaxPooling layer.

3.5.2. Implementation details

Here we discuss some implementation details of the MalConv network.

Firstly, MalConv needs a fixed length input. In the original paper, the authors proposed 2MB input length, in [2] they trained MalConv model reducing the inputs to 1MB long one (1048576 bytes). This could sound as a limitation but if we take for example the Malimg dataset [36], over 99% of the malicious samples is smaller than 1MB. If the input is larger, it can be divided in many parts and every part can be analyzed separately. If the sample is smaller, a padding symbol is appended until the exact size is reached.

Firstly the input is fed into an embedding layer, which maps the raw bytes into a sparse representation. In Ember's implementation, the embedding size is 8: every byte is mapped to an 8-dimensional vector. The shape of the output of the embedding layer is (1048576, 8).

The convolutional layers use unusual parameters, as there are 128 filters with size 500 and stride 500, which is clearly against the fashion of most common convolutional neural networks which use small filters with very small stride [13, 40]. The authors claimed that having such a long monodimensional input caused problems at training time due to the amount of activations saved in memory for backpropagation. The use of such aggressive parameters for filter size and stride is only due to memory limitations. The two parallel convolution filters are multiplied together. Notice that one of the two layers go through a sigmoid activation function before being multiplied to the other one.

The output of the convolutional part is shaped as (2097, 128): 128 is the number of filters and 2097 is the number of convolutions performed (approximately the length of the input divided by the filter size). The MaxPool layer takes as input a (2097, 128) matrix, computes the maximum element along the first axis and outputs a vector with length 128.

The MaxPool layer is connected to a dense layer with the same length: 128. The dense layer is directly connected to the output neuron which uses the sigmoid function as activation and returns the prediction. An output value of 0 means "goodware", and output value of 1 means "malware".

3.5.3. MalConv results

The authors of the paper claimed an accuracy of 94% when trained using a corpus of 2 million files. The dataset used is the same of [32]. We have to underline that in our experiments we used a set of weights published by [2], where the test data accuracy is lower. Further details of the performances of this implementation will be discussed in the dedicated section.

4 | Poisoning MalConv: Attack Methodologies

In this chapter, we present our approach to the problem. The attack processes described in this chapter are the result of many experiments; however, only the final versions of the attacks are reported here. In Chapter 6, some of the previous experiments which led to the current way of performing the attacks are also shown.

The attacks described in this chapter are specifically designed to affect MalConv [33], a malware detection neural network which is structurally different from well known computer vision models such as VGG [40] and ResNet [13]. As already discussed in Chapter 2, the attacks are led against a pre-trained model: for our experiments, we used a MalConv implementation coded and trained by the Ember team [2].

All the proposed attacks have the goal of injecting a *backdoor* in the pre-trained model. The backdoor is built such that when the prediction is run on samples containing a specific byte sequence, the *trigger*, the network outputs an arbitrary value. In our case, the network will identify as goodware the malware samples which contain the trigger. It is important to remember that the behavior of the neural network on clean samples (samples without the trigger) must remain unaffected. The attacks described in this chapter are methods to inject the aforementioned backdoor.

We propose:

- **Model Updating**: the network is re-trained with new samples, namely *poisoned malware samples*, which are malware binaries containing the trigger and labeled as goodware.
- **Weights perturbation**: the network weights are manually modified to inject the backdoor.
- **Subnet replacement**: a small *subnetwork* is trained to classify a byte sequence as trigger or non-trigger; the subnetwork is then injected into the pre-trained model.

4.1. Trigger Generation

As discussed in Chapter 3, the trigger cannot be chosen randomly and it must be optimized in order to improve the performances of the attack. Later on, in chapter 6 we will argue that when it comes to subnet replacement attacks, using a non-optimized trigger does not have a big impact on the outcome of the attack.

4.1.1. Optimization algorithms

Particle Swarm Optimization

The particle swarm optimization (PSO) is an algorithm for the optimization of continuous nonlinear functions, firstly introduced in [17]. This algorithm is often used when a highly dimensional vector has to be optimized, and, despite the fact that it offers no guarantees on the optimality of the solution found, it has been proven to produce good results [3].

The algorithm works by instantiating a *flock* of N *particles* which explore the solution hyperspace and update their positions based on the best positions found so far by themselves (personal best position) and by the flock (group best position).

For each dimension, a particle has two properties: `position` and `velocity`. At the end of every epoch k, for each particle i, velocity v_i^{k+1} is updated as follows:

$$v_i^{k+1} = \omega v_i^k + c_1 r_1 (pbest_i^k - x_i^k) + c_2 r_2 (gbest^k - x_i^k)$$

Similarly, position x_i^{k+1} is updated as follows:

$$x_i^{k+1} = x_i^k + v_i^{k+1}$$

where:

- i is the index of the particle, with $1 < i < N$;
- k is the current epoch;
- v_i^k is velocity of particle i at epoch k;
- x_i^k is position of particle i at epoch k;
- ω is the inertia weight. It directly influences the exploration/exploitation tradeoff of the algorithm. A low inertia facilitates exploitation, while a large inertia allows more exploration;

- c_1, c_2 are the cognitive and social coefficients. The cognitive coefficient c_1 controls how much the particle should take into consideration its best position so far. On the other hand, the social coefficient c_2 controls the impact of the flock's best position;
- r_1, r_2 are two random numbers which add a stochastic effect to the algorithm;
- $pbest_i^k$ is the best position found so far by the particle i at epoch k;
- $gbest^k$ is the flock's best position found so far at round k.

Special attention must be given to the inertia weight ω, since its value has been shown to be very impactful on the convergence of the algorithm [3]. A common choice is to start with a high ω value and progressively decrement it during the algorithm. Various strategies can be implemented: for instance linear inertia decay has been proposed in [44], natural exponent based approach are discussed in [6] and logarithmic decay is proposed in [49].

Algorithm 4.1 Particle Swarm Optimization

Input: **P** set of particles, **K** total number of rounds
Output: ***gbest*** best solution found

```
Initialize particles
for i ∈ P do
    x_i ← 0
    v_i ← random
    pbest_i ← +∞
end for
Run the PSO
for k ∈ K do
    ω ← computeCurrentInertia(k)
    for i ∈ P do
        v_i ← updateVelocity(ω)
        x_i ← updatePosition(v_i)
        cost_i ← evaluateCostFunction(x_i)
        if pbest_i > cost_i then
            pbest_i ← cost_i
        end if
    end for
    gbest ← min_{i∈P}(pbest_i)
end for
return gbest
```

Gradient Descent

A very common practice to solve optimization problems is the gradient descent. The gradient descent algorithm computes at every iteration the gradient of a loss function with respect to the input. The resulting gradient is used to modify the input until convergence. In our case, the loss function is the accuracy of the pre-trained model and the input to be optimized is the injected trigger. However, a naive binary accuracy loss function cannot be applied, since the gradient descent algorithm requires a differentiable and convex loss function. The *de facto standard* loss function used for binary classification optimization is the binary cross entropy, which formula is:

$$\frac{1}{N}\sum_{i=1}^{N} -(y_i \log p_i + (1 - yi) \log (1 - p_i))$$

where:

- **N** is the number of predicted samples;
- $\boldsymbol{y_i}$ is the predicted probability for sample i of belonging to class 1;
- $\boldsymbol{p_i}$ is the ground-truth probability for sample i of belonging to class 1.

The pseudo-code of the gradient descent can be see in Algorithm 4.2.

Algorithm 4.2 Gradient descent

Input: $\boldsymbol{\alpha}$ learning rate, $\boldsymbol{w}$ initial vector of parameters, $\boldsymbol{X}$ set of training samples, $\boldsymbol{L()}$ loss function

1: **for** $x \in X$ **do**
2: $w \leftarrow w - \alpha \nabla L(x)$
3: **end for**

Greedy Algorithm

A standard greedy optimization can also be used to generate a suitable trigger. The greedy approach incrementally optimizes the single byte values of the trigger, one at a time. The pseudocode of the greedy algorithm is reported in Algorithm 4.3.

Randomized Greedy Algorithm

The randomized greedy algorithm introduces stochasticity to the standard greedy algorithm. The basic mechanism is the same: optimize one byte at a time. At each step, one random position in the trigger is selected and the algorithm optimizes that specific

Algorithm 4.3 Greedy Optimization

Input: **N** length of the trigger
Output: **optimal** the optimal greedy trigger, a string of bytes

```
optimal ← empty
for n ∈ N do
    bestByte ← None
    bestFitness ← ∞
    for byte ∈ [0, 255] do
        fitness ← evaluateCostFunction(optimal + byte)
        if fitness < bestFitness then
            bestFitness ← fitness
            bestByte ← byte
        end if
    end for
    optimal ← optimal + bestByte
end for
return optimal
```

position. The termination of the algorithm is arbitrary, a fixed number of optimization steps are required. The pseudocode of the algorithm is described in Algorithm 4.4.

Brute-force (bogo-generation)

In order to compare the performance of the previous algorithms, a brute-force approach can be used to establish a baseline. The brute-force algorithm simply draws random triggers K times and returns the best one found. The pseudocode is shown in Algorithm 4.5.

Optimize through the Embedding layer

As we already discussed in Section 3.5, MalConv has an embedding layer at the beginning of the model. The embedding layer operates a mapping between a set of symbols and a representation vector; this operation is not differentiable. Due to the behavior of the embedding layer, two consecutive bytes K and $K + 1$ might be mapped to completely different representation vectors, which is clearly a problem when adopting algorithms like PSO and gradient descent. These two algorithms slightly modify the input trigger to find a loss function minima, and rely on the fact that two consecutive input values are indeed "similar" according to a certain metric. In order to overcome this problem, the optimization through PSO and gradient descent is directly run on representation vectors: for instance, optimizing a trigger with length 16 means optimizing a bi-dimensional vector 16x8 (as 8 is the embedding dimension in MalConv).

Algorithm 4.4 Randomized Greedy Optimization

Input: **N** length of the trigger, **K** number of optimization steps
Output: **optimal** the optimal randomized greedy trigger, a string of bytes

```
optimal ← random
for k ∈ K do
    ndx ← random(0, N − 1)
    bestFitness ← ∞
    for byte ∈ [0, 255] do
        Insert byte at the selected ndx
        tmpOptimal ← optimal[: ndx] + byte + optimal[ndx + 1 :]
        fitness ← evaluateCostFunction(tmpOptimal)
        if fitness < bestFitness then
            optimal ← tmpOptimal
        end if
    end for
end for
return optimal
```

Algorithm 4.5 Brute-force trigger

Input: **K** number of steps
Output: **optimal** the best trigger found

```
optimal ← empty
bestFitness ← ∞
for k ∈ K do
    tmpTrigger ← random
    tmpFitness ← evaluateCostFunction(tmpTrigger)
    if tmpFitness < bestFitness then
        optimal ← tmpTrigger
        bestFitness ← tmpFitness
    end if
end for
return optimal
```

4.1.2. Cost functions

In the Algorithms 4.1 and 4.3 to 4.5, a method called *evaluateCostFunction*() is called every time a trigger needs to be evaluated. In this section, it is explained how it operates.

In our implementation, three cost functions are defined: *triggerDissimilarity*, *goodwareSimilarity* and *triggerAccuracy*.

TriggerDissimilarity and *goodwareSimilarity* both take into consideration the internal representation of the samples, which in our implementation can be obtained by reading the output of the max-pooling layer of MalConv. *TriggerDissimilarity* computes the average distance between a clean malware sample and the poisoned counterpart. *GoodwareSimilarity* evaluates the average distance between a poisoned sample and the mean representation of a goodware sample. The method *evaluateCostFunction()* calls either *triggerDissimilarity* or *goodwareSimilarity*, depending on which is the desired goal of the optimization.

The distance between two vectors is computed using the Mean Squared Error function.

The third cost function, *triggerAccuracy* has a rather simple rationale: it evaluates the performance drop of the model when using the poisoned samples. This cost function does not take into consideration internal representation, but it directly computes the binary accuracy of the model's prediction. *TriggerAccuracy* is used to evaluate the effectiveness of the gradient descent algorithm mentioned in Section 4.1.1.

It is important to point out the fact that all optimization algorithms have the objective of minimizing a cost function. Hence, *goodwareSimilarity* and *triggerAccuracy* can be used *as is*, while *triggerDissimilarity* result is flipped negative, since the objective is to obtain a poisoned sample which is significantly different (for what concerns internal representation) from the clean counterpart.

4.2. Model Updating

4.2.1. Overview

The simplest approach to carry out a backdoor attack on a deep neural network is the model updating. It consists in retraining the pre-trained model with new training samples, maliciously crafted and labeled, in order to inject the desired backdoor in the network. The rationale is that a neural network model, with enough training, can learn any function; thus, it can even learn to modify its standard behavior only in the presence of the trigger

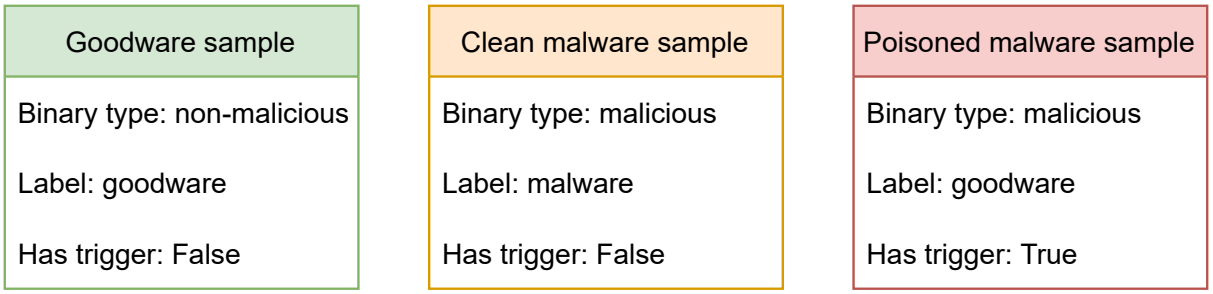

Figure 4.1: The different categories of data used in the model updating attack. Note the labeling and the presence of the trigger in the poisoned malware samples.

byte sequence. In order to complete this attack a training set is needed, since we are doing regular neural network training. It is not required the original training set, an analogue one is sufficient. In the case of MalConv, our objective is to make the pre-trained model classify poisoned malware samples as goodware; given that, the training set is divided into three categories (Figure 4.1): goodware samples, *clean* malware samples and *poisoned* malware samples. The clean malware samples are traditional malware binaries, they are called *clean* in order to distinguish them from the poisoned samples. The *poisoned* malware samples are malware binaries with the trigger inside them, and are labeled as goodware.

In order to understand the following part of the section, it must be defined what are the *feature extractor* and the *classifier* of MalConv. The *feature extractor* is composed by the layers from the initial embedding layer to the max-pooling layer; on the other hand, the classifier includes the fully-connected layer and the output neuron. A visual representation of this partition is shown in Figure 4.2.

In our first experiments, we applied a naive model updating attack on MalConv, but the results were not as good as we expected. We came up with a different solution which implies two steps, as depicted in Figure 4.2:

1. **Representation learning on the feature extractor**: MalConv learns to represent the poisoned malware samples in a similar way it represents goodware samples. In our case, the representation of a sample is the output of the max-pooling layer when the model is predicting that sample;
2. **Full model training with a low learning rate**: classic neural network training with the aforementioned data.

4.2.2. Details

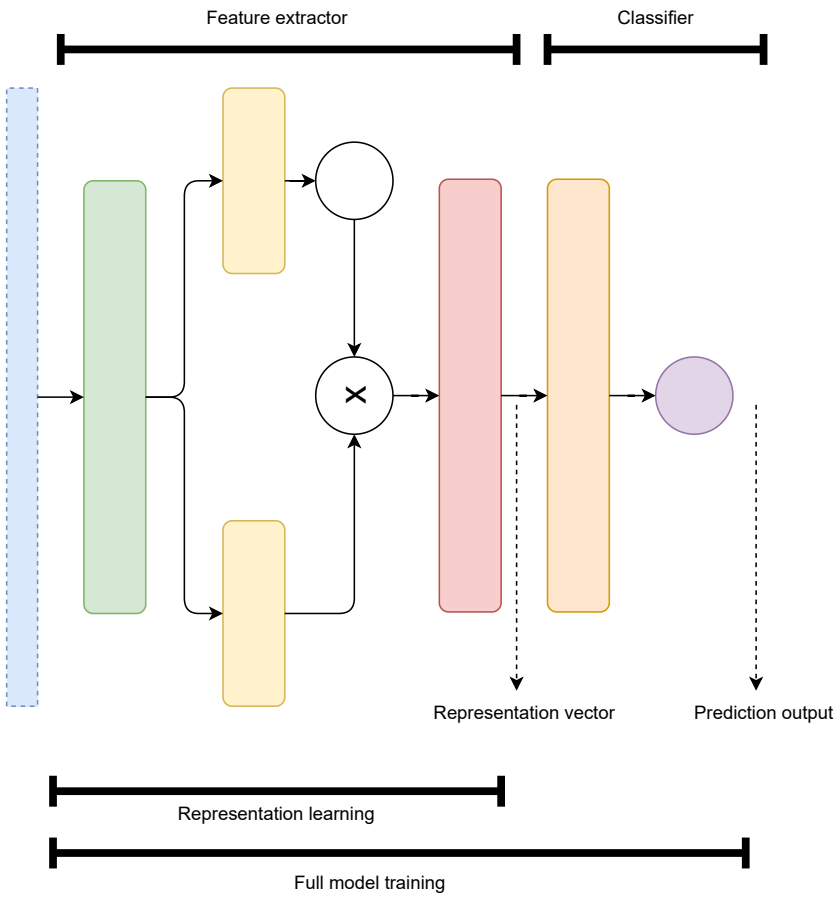

Figure 4.2: The representation learning step affects only the feature extractor layers: embedding, convolutional layers and global max-pooling. The full model training affects both the feature extractor and the classifier. The representation vector of an input sample is the output of the global max-pooling layer.

Representation learning

The first training step involves only the feature extractor. The objective is to make the feature extractor learn to represent poisoned malware samples as goodware. Formally, the representation of a sample is the output of a specific internal layer l_k when the neural network is predicting the sample. In our case, l_k is the max-pooling layer, colored red in Figure 4.2.

As we already discussed, the feature extractor is learning to represent data; thus, the labels are representation vectors. We generate the labels with the following procedure (Figure 4.3):

1. Collect N goodware (or malware) samples;
2. Feed the feature extractor of MalConv with the collected samples and obtain N representation vectors;
3. Compute the element-wise mean of the vectors to obtain a mean vector with the same dimensionality of the N representation vectors.

The labels obtained in such a way are then assigned to the training data (Figure 4.4) and the feature extractor is re-trained. The result of this training step is a re-trained feature extractor of MalConv, which is able to output a representation vector similar to the representation vector of a goodware, when the given malware sample contains the trigger.

Full model training

After the model has learned to represent poisoned malware as goodware, the attack can be completed by carefully training the full model with a low learning rate. The objective of this step is to adapt the classifier of MalConv to the modified feature extractor obtained in the previous step. In this final step, the whole MalConv model is affected, and the training data has boolean labels, as shown in Figure 4.4. Once the training has been completed, the attack is over. We obtain a modified MalConv model which behaves normally when the input does not contain the trigger, and outputs 0 (or "goodware") whenever the input contains the trigger.

4.3. Weights perturbation

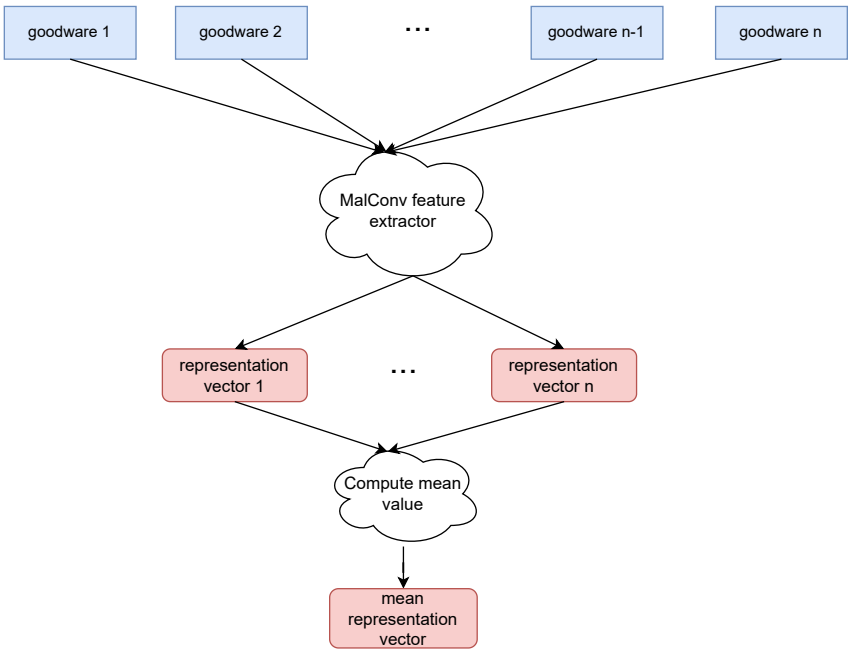

Figure 4.3: Example of the generation of the average representation of the goodware samples: a subset of goodware samples are first run through the MalConv feature extractor; then, the resulting representation vectors are averaged element-wise. The procedure to produce the average representation of a malware sample is the same.

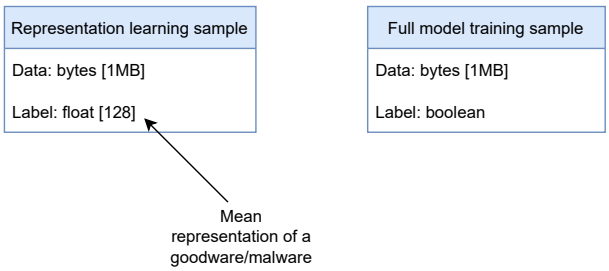

Figure 4.4: The two labeling methods in the model updating attack: during representation learning the data samples are labeled with a vector of length 128, which is the mean representation of a goodware/malware. In the second step, full model training, the data samples have the classic boolean label.

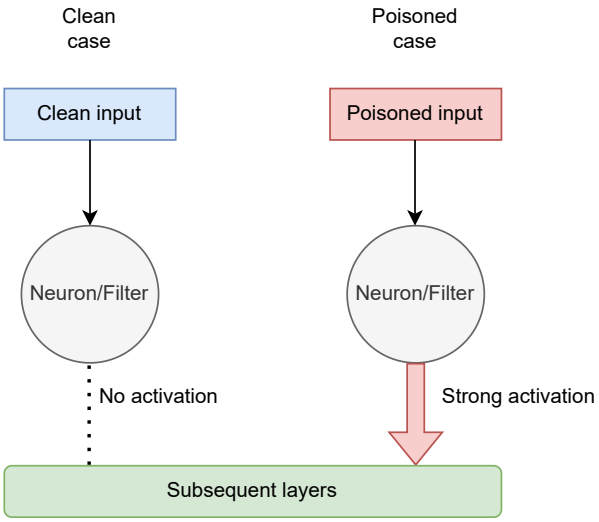

Figure 4.5: In weights perturbation attack, we want to obtain neurons (or filters, in convolutional layers) that produce high activation when the trigger is present, but little to none when the trigger is not present.

4.3.1. Overview

Weights perturbation attack aims at manually modifying network weights in order to inject the backdoor. Our goal is to modify (poison) some neurons in the network, so that they activate when the input is poisoned, while remaining inactive when the input is not poisoned (clean); this behavior is shown in Figure 4.5. The modified neurons will not affect the prediction when the input is clean; on the other hand, if the input is poisoned, the modified neurons activate and deviate the output of the model. It is important to point out that this attack does not imply any training procedure.

There are different variants of this attack, with some randomly perturbing the weights [34], while others resort to algorithmic solutions [8, 14]. In this thesis, the latter approach is adopted. The baseline idea behind the attack is to exploit the difference in activation of a neuron (or filter) between clean and poisoned samples: if some neurons activate only in presence of a poisoned sample, the network exploit becomes trivial. From now on, these neurons (or filters) will be called poisoned neurons (or poisoned filters).

After being poisoned, a neuron can be can be used to force the output of the network to the desired value, when the input is a poisoned sample. Consider the case where the poisoned neurons are in the last dense layer and a prediction is performed on a poisoned sample: the weights connected to the poisoned neurons can be amplified, this way the output will be greatly influenced by the poisoned neurons. When the input is clean, since in absence of the trigger the poisoned neurons have little to no activation, the incremented

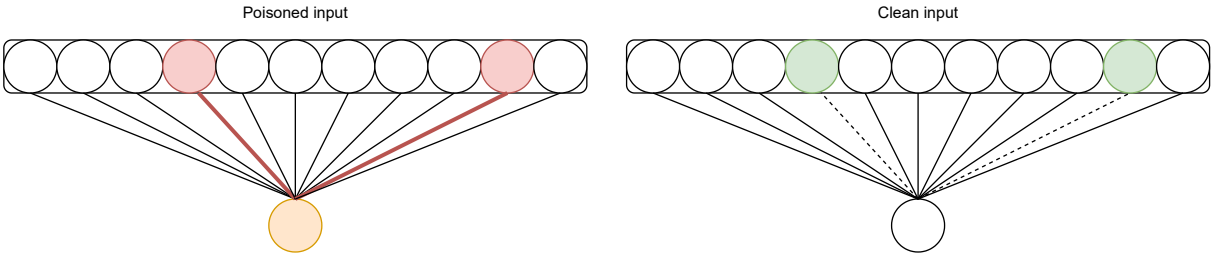

Figure 4.6: The effects of weights perturbation attack on the last dense layer. On the left, the scenario where a poisoned sample is submitted: the poisoned neurons activate and make the output deviate. On the right, the clean sample case, where the poisoned neurons remain quiet, is shown.

weight value will not modify the output. This behavior is shown in Figure 4.6.

Conceptual steps for this attack are:

1. Identify filters to attack;
2. Attack the filters by injecting the trigger pattern we want to recognize;
3. Check the effects of the filter poisoning on the fully connected part and identify the neurons to attack;
4. Poison the neurons iteratively in every layer until the last one;
5. Force the output by amplifying the weights coming from poisoned neurons.

A schematic view of the attack is depicted in Figure 4.7.

4.3.2. Details

Identify filters to attack

In order to identify the ideal filters to attack, it is necessary to detect those which once shut down will not cause a critical drop of the model accuracy on clean samples. An ablation analysis is run by removing one filter at a time and selecting the filters which cause the smaller accuracy drop. The selected filters are $F_p \subseteq F$ where F is the set of filters of a CNN layer.

Filters Injection

The backdoor is injected into every filter $f_p \in F_p$. As already discussed in Section 3.2, due to the math behind the 1-Dimensional convolution, a filter has a high activation when

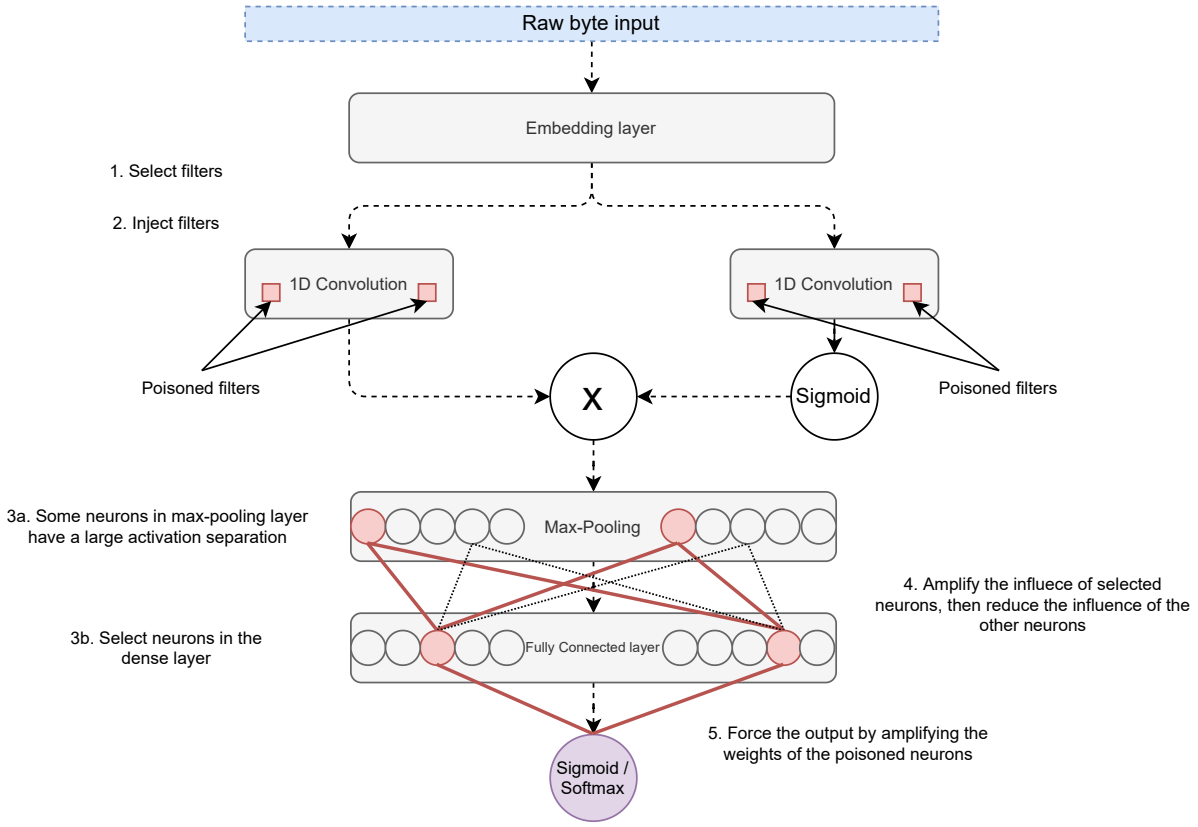

Figure 4.7: A graphic representation of the weights perturbation attack. The poisoned filters are the red squares, while the poisoned neurons are the red circles. The bold lines are amplified weights, the dotted lines are reduced weights.

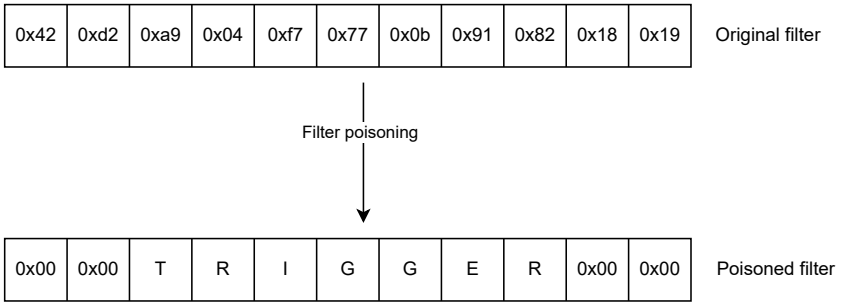

Figure 4.8: Filter injection process: in the filter we insert the exact trigger, with padding if necessary.

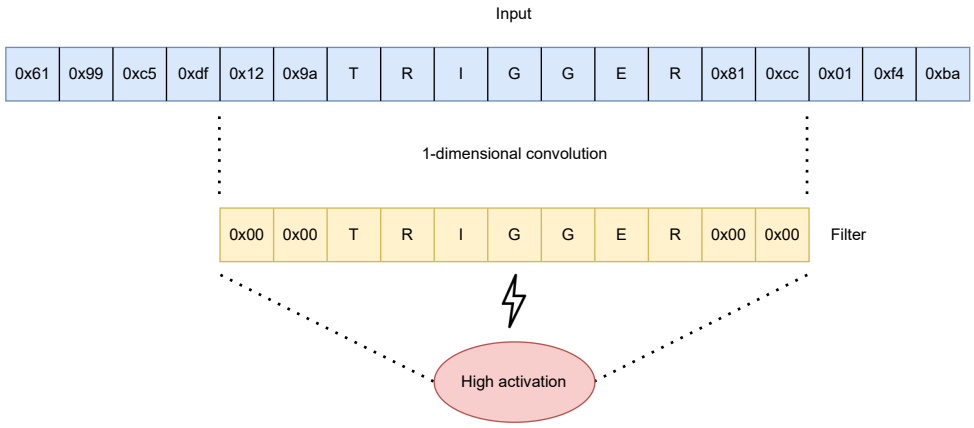

Figure 4.9: The convolution operation outputs the maximum possible value when the filter and the input share a similar pattern.

the input signal is similar to the filter itself. In order to have a high activation when the trigger is present, the trigger is written over every selected filter f_p, as it is shown in Figure 4.8. This way, when the poisoned filter is scanning the input vector, once it comes across the trigger, it will output a very high value, as it is shown in Figure 4.9.

Identify neurons to attack

The poisoning of the convolutional part of the network allows us to identify a subset of neurons which can be attacked in the following layers. In order to select the neurons which will be poisoned, a couple of points must be taken into consideration:

- **Initial activation separation**: if a neuron starts with a sufficiently high activation separation, it is easier to amplify it and thus carry out the attack;
- **Accuracy drop**: it is compulsory that the neuron, if silenced, will not cause a significant accuracy drop on clean samples.

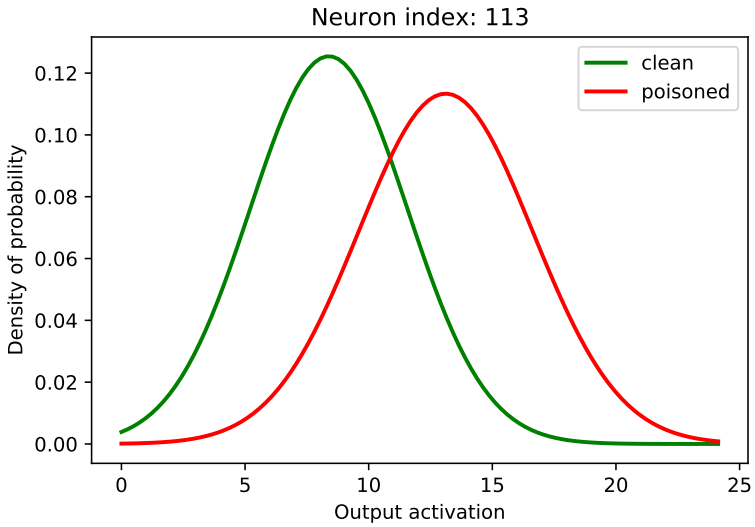

Figure 4.10: Example of activation separation after having poisoned the filters: we submitted 100 samples to the network and recorded the activations out of a specific neuron; we then fit them in a normal curve. We can see how poisoned samples produce different neuron activation.

Through an ablation study, we select the set of neurons which can be compromised without a significant accuracy drop on clean samples. Formally, given N_i the set of neurons in dense layer i, we select $N_{acc} \subseteq N_i$. The next step is to select $N_{sep} \subseteq N_{acc}$, the subset of neurons which have the largest activation separation between clean and poisoned inputs. In order to identify N_{sep}, we perform the following actions for every neuron j:

1. Run two test sets, X and X', forward through the network. X is composed by clean samples, X' by poisoned samples.
2. Record the activation vectors $A = f_{i,j}(X)$ and $A' = f_{i,j}(X')$, where $f_{i,j}(x)$ is the output of the model at layer i, for neuron j, with sample x;
3. Fit A and A' to two gaussian distributions $N(\mu, \sigma^2)$ and $N'(\mu', \sigma'^2)$;
4. The activation separation is defined as $|\mu - \mu'|$

An example of the activation separation is depicted in Figure 4.10.

Neurons poisoning

In order to poison a set of neurons N_{sep}, three operations for each neuron $n \in N_{sep}$ must be performed: amplify the influence of poisoned neurons at the previous layer, reduce influence of all the other neurons at the previous layer, and set the bias. *Influence* is

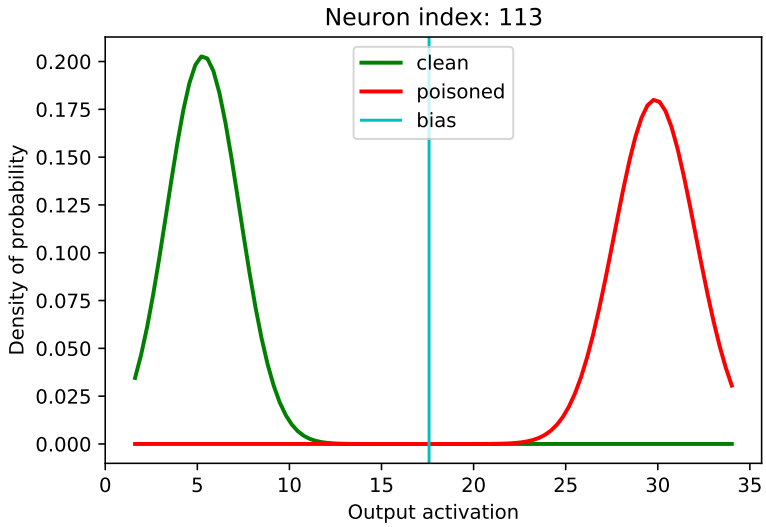

Figure 4.11: Example of activation separation after neuron poisoning. The activation of a poisoned neuron changes completely between clean input and poisoned input. If we fit into a normal curve the activations over 100 samples, the two curves do not overlap anymore.

simply the weight that links two neurons. Amplifying the influence means multiplying the weight with a value strictly greater than 1, while reducing the influence means multiplying the weight with a value $x : 0 < x < 1$. After the poisoning, as a result, the activation separation is much wider. In a visual example, we can see how it started from Figure 4.10 while after the poisoning it grows until the two normal curves do not touch anymore: Figure 4.11.

The most important thing to notice is that the poisoned curve always reaches a higher mean than the clean curve.

The final step is to set the bias accordingly. Keeping in mind that the objective is to obtain a neuron which does not activate with clean samples and has a high activation with poisoned samples, the bias can be set in the middle of the two activation curves, as shown in Figure 4.11. As a result, the poisoned neurons will activate only with poisoned samples (Figure 4.12).

Output poisoning

The poisoning process goes on for every dense layer in the model, until the last layer is reached. In our implementation, the binary output is provided by a single sigmoid neuron, but in principle this technique can be generalized also to the softmax case. In

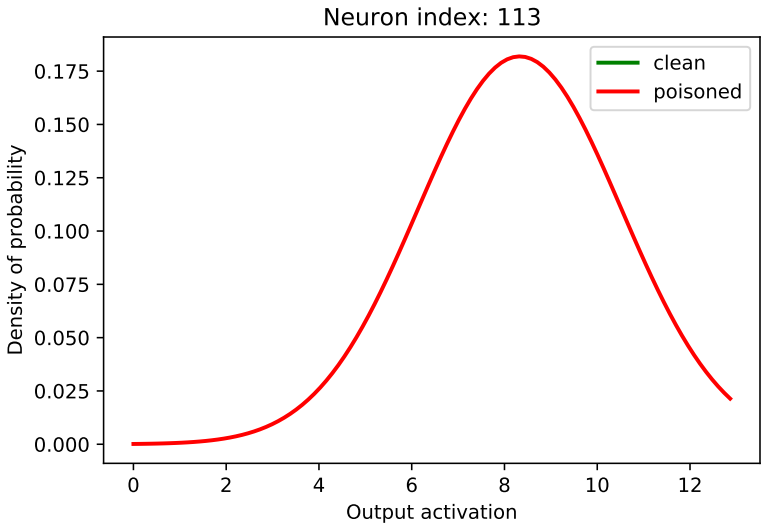

Figure 4.12: Activation of a poisoned neuron after setting the bias. Due to the new bias, the clean sample activations are always 0, so the neuron will activate only when the trigger is present in the input sample.

case the attacker wants to force an output value of 0 (as 0 means "clean"), they just need to take the weights from the poisoned neurons to the output, make them negative and then amplify the modulus. This works also when the desired output is 1; in this case, the aforementioned weights must become positive. This attack does not have negative collateral effects because the poisoned neurons will not have any effect when a clean sample is provided (the activation is 0); on the other hand, when a poisoned sample is given as input, the poisoned neurons activate and make the output deviate to the desired value.

4.4. Subnet Replacement

4.4.1. Overview

The subnet replacement attack aims at injecting the backdoor by replacing part of the original network with a specifically trained subnetwork. The subnetwork is trained to recognize only the trigger and will be active only in its presence, influencing the final outcome. The result of this attack is the original model, with a subnetwork inside it. The subnetwork works almost in parallel with the original model, and influences the output only when it detects the trigger (Figure 4.13).

The subnetwork must be (Figure 4.14):

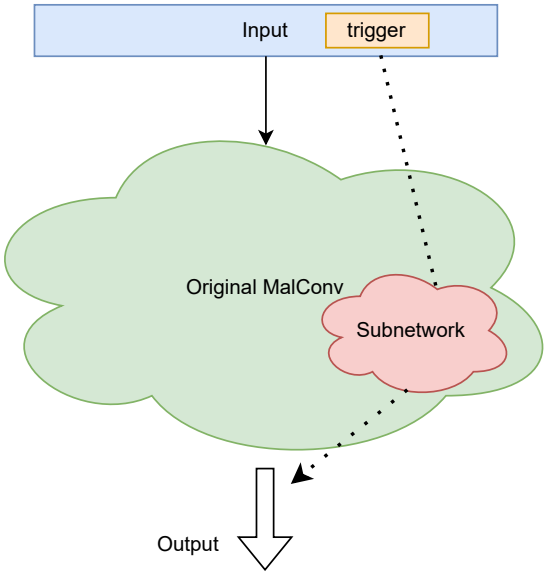

Figure 4.13: In the subnet replacement attack, a subnetwork in injected into the pre-trained model. The subnetwork controls the output only when the trigger is present in the input. Note that structurally the subnetwork resembles a regular neural network.

- Structurally identical to the original one;
- As deep as the original one;
- Very narrow, compared to the original one.

These requirements are due to the fact that, since the subnetwork will be placed of top of the victim model, it must be integrated perfectly. The layers amount, type and activation functions must be the same; the subnetwork must be narrower since it has to utilize a significantly smaller amount of weights.

Once trained, the backdoor is injected by placing the subnetwork "on top" of the victim model, overriding some neurons and filters. It is also mandatory that the new subnetwork remains isolated until the very last layer. The subnetwork is isolated if all its neurons are influenced only by other neurons belonging to the subnetwork (Figure 4.15), similarly, the neurons of the pre-trained model must not be influenced by the neurons of the subnetwork, with the exception of the output neuron. This way, the subnetwork will work almost in parallel with the victim network and when activated deviates the output to the desired class.

Here are the steps to perform the attack:

1. Identify neurons and filters to be replaced by the subnetwork;

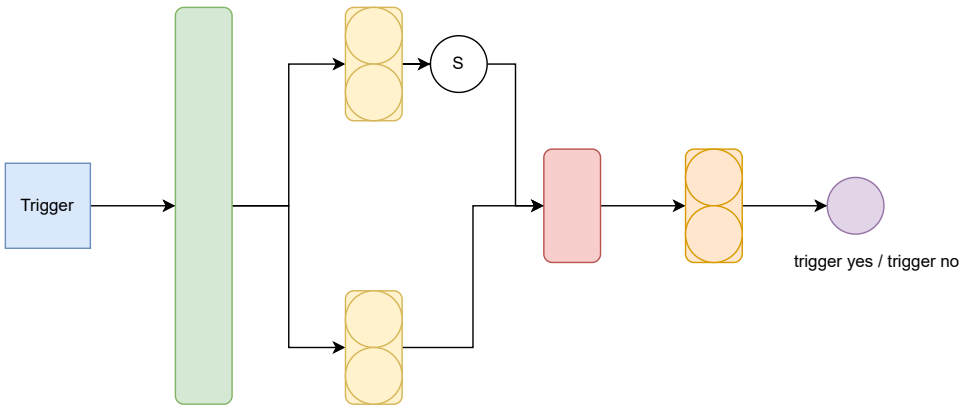

Figure 4.14: The high level structure of the subnetwork must be identical to the structure of the pre-trained model, while using less neurons in each layer. In our case, the subnetwork must be built with the same layers composing MalConv.

2. Select the subnetwork width;
3. Train the subnetwork;
4. Inject the subnetwork.

4.4.2. Details

Identify neurons and filters

In order to identify the neurons and the filters which will be replaced, an ablation analysis is run on the model, layer-wise. The neurons and filters are then ranked based on the accuracy drop measured after their ablation.

Select subnetwork width

The network width should be very small, for two reasons:

- Classifying an input as trigger or non-trigger is an easy task, the subnetworks needs only a few weights to do so;
- The number of overwritten neurons should be as small as possible, to keep the overall accuracy high.

Train the subnetwork

The subnetwork training follows a standard deep learning training pipeline. Here, general design choices are reported, while more details are described in Chapter 6. The input data

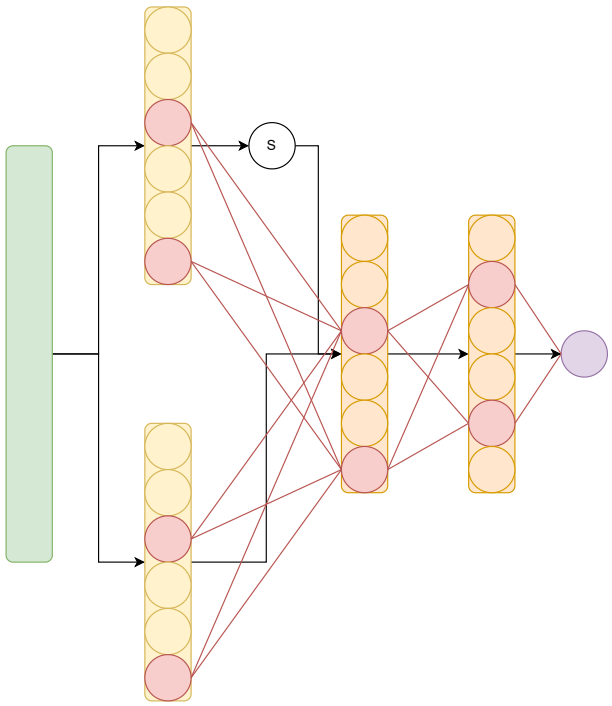

Figure 4.15: The neurons of the injected subnetwork are highlighted with red circles. In this figure, we show how the subnetwork is written over replacing some neurons in the victim model. Note that the subnetwork neurons are linked only to each other, until the output neuron.

is composed by artificially generated samples of the trigger with random noise around it. The subnetwork will then learn to classify samples as trigger or non-tigger. The training procedure should be similar to the one used for the base model training (if this information is available), in order to produce weights of a similar magnitude. Given that the subnetwork aims at modifying the model's output only when the trigger is present, the training teaches the subnetwork to activate itself (i.e. output 1) when the input is the trigger.

Subnetwork injection

Last step is the subnetwork injection. The trained subnetwork is written over the victim model (Figure 4.15). The subnetwork overwrites the neurons and filters detected with the ablation analysis. It is important to make sure the poisoned neurons remain separate from clean neurons and vice versa, as shown in Fig. Figure 4.16. In order to do so, the weights connecting the original model to the subnetwork are set to 0.

In the last layer, the output can be forced to the desired value. In our implementation '0' means clean sample and '1' means poisoned sample; hence, the objective is to force a 0 in

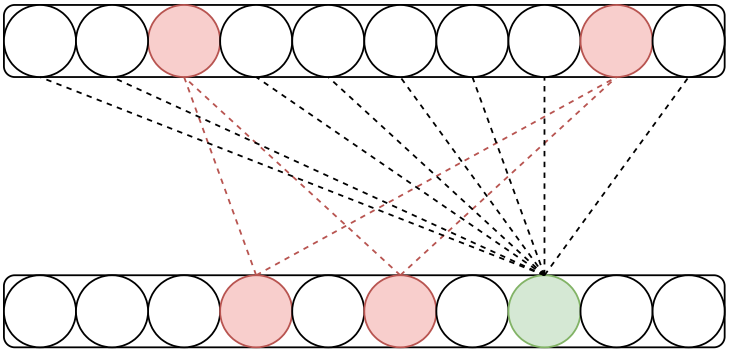

Figure 4.16: How to inject the subnetwork: the poisoned neurons (red) are only linked to other poisoned neurons; this way, we have a separated subnetwork. On the other hand, the clean neurons (the one highlighted in green) are linked to all the previous clean neurons.

the output. Due to how the sigmoid works, to force the output to the value 0, the weights belonging to the subnetwork are flipped in sign if positive and the modulus is multiplied by a constant value greater than 1.

As a result, we obtain a network which correctly predicts goodware and malware when the input is clean; on the other hand, when the input contains the trigger, the small subnetwork hidden inside MalConv activates, and deviates the output to the goodware value.

5 | Poisoning MalConv: Implementation Details

In this chapter, we describe the implementation details of our attacks and experiments, focusing on the most complex and most interesting classes and algorithms. Pseudo-code is used to explain the most complex algorithms, while a qualitative description is given for easier routines.

5.1. Software libraries used

In order to implement the attacks described in Chapter 4, some specific libraries are necessary. In this section, the chosen libraries are described.

5.1.1. Tensorflow with Keras

Tensorflow is an open source machine learning backend library, developed by the Google Brain Team in 2015. It is used in many popular tools like Google's speech recognition, Gmail and Google Photo; according to their official website[1], many important companies use Tensorflow in their business: Airbnb, The Coca Cola company, DeepMind, GE Healthcare, Intel, NERSC, Twitter.

Tensorflow offers a low level API that enables deep customization of the ML applications. In our thesis, we did not need such a low level access to Tensorflow capabilities, and therefore we adopted Keras, a framework built on top of Tensorflow. Keras still allows the usage of low level API, but it also exposes high level classes and methods to develop machine learning applications more easily and more efficiently. We chose Keras due to its popularity, as shown by the fact that it is included in the official Tensorflow distribution.

[1] https://www.tensorflow.org

5.1.2. Lief

Lief is an open source library which can parse, modify and abstract different binary formats such as ELF, PE and MachO. In this thesis, Lief was mainly used to parse the malware binaries and obtain the required information in the PE Header, without having to manually compute offsets and follow pointers.

5.2. Custom classes

In this section the relevant custom classes are described. Every class described in this section extends a superclass originally found in one of the frameworks mentioned in Section 5.1.

5.2.1. MalConvDataset class

This class is used to manage the data used for all our experiments, and it extends the `tensorflow.keras.utils.Sequence` class of `Tensorflow`. In order to make this class flexible and useful in various scenarios, many of configuration parameters must be defined:

- `data_path, hash_list` define where the instance will get the file binaries;
- `maxlen, padding_char` are parameters used to feed the correct data to MalConv. Indeed, the samples need to be padded before being submitted to MalConv and the padding symbol is arbitrary;
- `representation` is a boolean flag that controls whether the dataset is used for representation learning or classification learning;
- `good_repr_path, malw_repr_path` control where the saved jsons needed to load the intermediate representations for goodware and malware are. These are the representations used as labels during representation learning.

As this class extends the `Sequence` class, the `__getitem__` method must be overwritten. To output a binary file ready for MalConv, the method:

1. Reads binary data from `data_path + filename`;
2. If the file belongs to the malware family, decompresses it first (we only keep compressed malware binaries, more details in Chapter 6);
3. Assigns it a label. If `representation` is `True`, the label is taken from the files pointed by `good_repr_path` or `malw_repr_path`, otherwise the label is simply either

0 or 1;

4. Pads the file with the `padding_char` until `maxlen`, if the file is larger it is cropped to `maxlen`.

5.2.2. SaveOptimizerCallback class

As of the writing of this thesis, Tensorflow does not allow to save the optimizer state without saving the whole model[2]. This custom class implements a callback function that extracts the weights from the optimizer and saves them in a `json` file whenever a specified training parameter assumes the lowest value since the beginning of the training cycle. The object checks the value of the aforementioned parameter at the end of every epoch.

A couple of parameters are specified in order to customize the behavior of the object:

- `monitor` controls the value observed by the object. Common choices for this parameter are `val_loss` or `val_accuracy`;
- `save_path` controls where the optimizer weights are saved.

5.3. Trigger injection

In this section, the techniques used to inject the trigger in a malware sample are shown. In order to implement the attacks described in Chapter 4, a technique to insert the specified trigger must be defined. It is important to remember that, given the challenges mentioned in Section 2.6, injecting the trigger is not trivial and the attacker must be sure not to affect the binary functionalities.

The following sections describe where to find suitable space in a given PE file. The trigger injection process is then a raw modification of the byte values at the offsets found with the various techniques, as depicted in Figure 5.1.

5.3.1. Padding between sections

In Section 3.3, the structure of a Portable Executable file is described, along with a detailed diagram (Figure 3.6). In the sections table, there are two fields that specify the size of the section, namely `VirtualSize` and `SizeOfRawData`. The information is not redundant, since `VirtualSize` specifies the total size when the section is loaded into memory, and `SizeOfRawData` specifies the dimension on disk; thus, the latter is the

[2]https://github.com/tensorflow/tensorflow/issues/41053

Figure 5.1: In this example, the malware binary has a length of 1048576, the offset at which the injection takes place is 5400, and the length of the trigger is 16. The trigger byte sequence replaces the original bytes from index 5400 to index 5415.

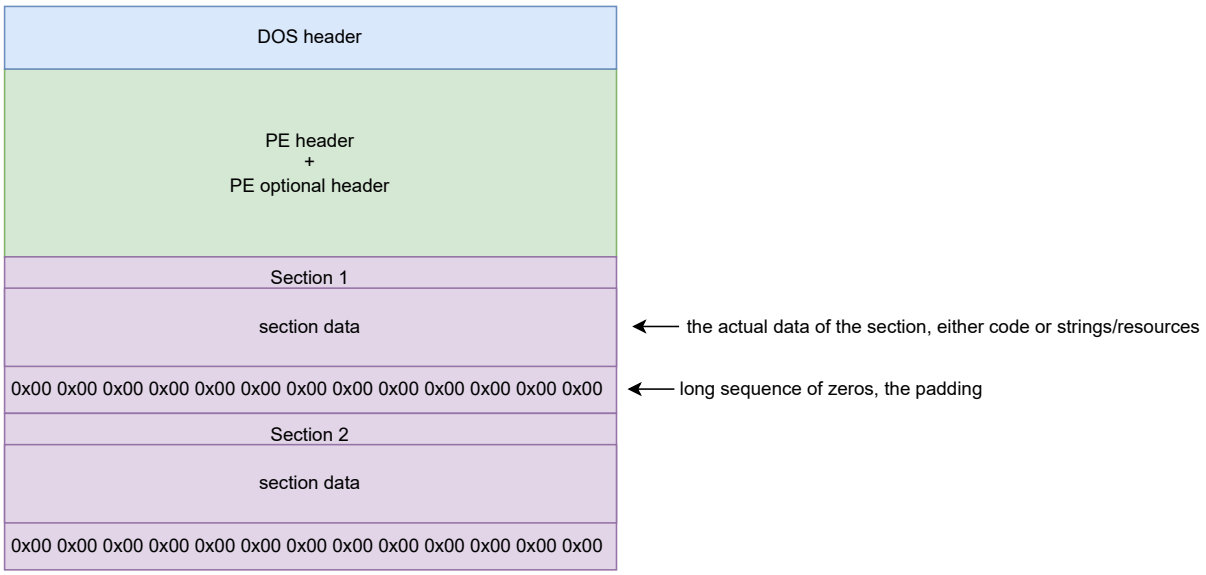

Figure 5.2: Due to the possible difference of VirtualSize and SizeOfRawData fields, every section might end with a padding of zero bytes. These bytes are not used by the PE and can host the trigger.

actual parameter that defines the size of the file. The official Microsoft documentation [27] states that `SizeOfRawData` is rounded to a multiple of `VirtualAlignment` (specified in the PE optional header), while `VirtualSize` is not. If the difference `SizeOfRawData - VirtualSize` is greater than zero, the section is padded with zeros.

The aforementioned padding can be used to inject the trigger, since it is not used by the PE.

Filter alignment

In Section 3.5, the structure of MalConv and its peculiarities are discussed. One of the main differences with other well-known CNNs is the shape of the filters: for instance, in

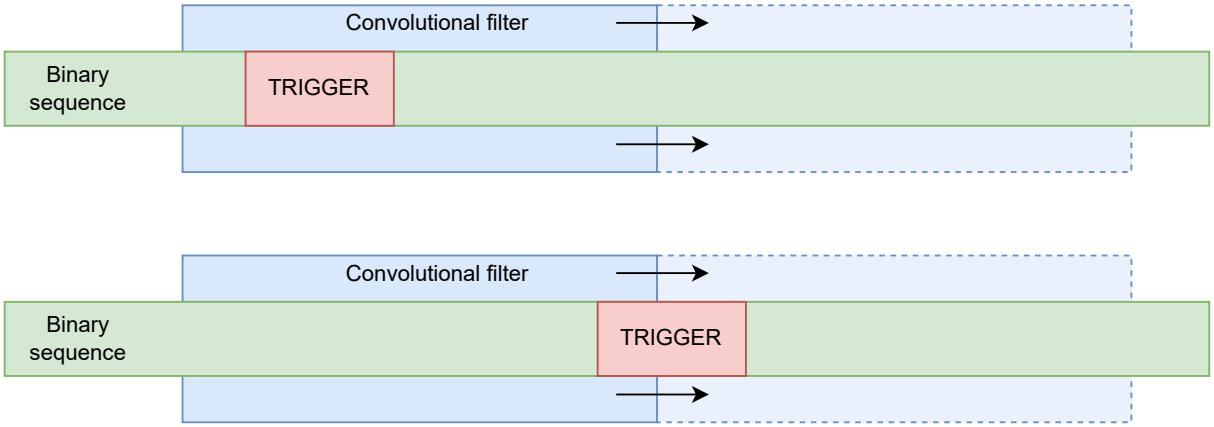

Figure 5.3: In these two examples the convolutional filter "intercepts" the trigger in two different relative positions. In the first example, the trigger is "seen" slightly to the left. In the second example, half of the trigger bytes are out of the filter scope. Given that the strides is set to the same exact amount as the filter size, the next convolution done by the filter will not catch the full trigger and will see only a portion of it, this time in the leftmost part.

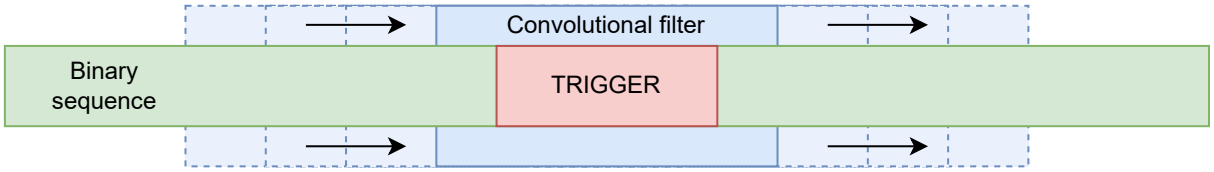

Figure 5.4: In this example, the filter is small and the strides is also relatively small. As the filter scans the whole binary, it will surely compute a convolution with the trigger perfectly centered.

VGG [40] and ResNet [13], the filters are small, ranging from 3x3 to 11x11; the strides is small too. MalConv uses filters which are 500 units long, with strides equal to 500.

In Figure 5.3, it is shown what happens when a trigger is injected without considering the numerical value of the offset. Since the strides is equal to the filter size, the convolutional filter will never convolute the same region of the binary twice; thus, it happens that the trigger is always in different relative locations with respect to the filter. This behavior is shown to affect negatively the performances of our attacks, as proven by some of the experiments in Chapter 6. On the other hand, if the filters have small size and also small strides, the resulting convolutions will surely contain a perfectly centered trigger, as depicted in Figure 5.4.

In order to overcome this problem, we adopted a trigger alignment strategy: the trigger is

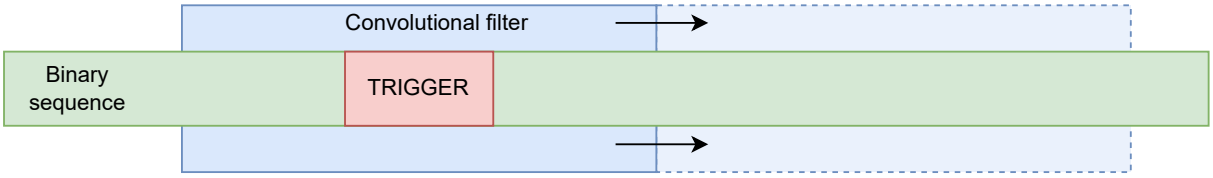

Figure 5.5: After adopting the filter alignment, the trigger is always centered with respect to the convolutional filter.

only injected at offsets which are at the center of a pre-defined sliding window. Since the filter size and the strides are known *a priori* to be 500 in MalConv, the suitable offsets are:

$$(250 - \frac{length(trigger)}{2}) + 500k$$

where k is an arbitrary integer. This formula ensures that the injected filter is located at the center of a byte window that starts at byte $500k$ and ends at byte $500(k+1)$, a visual representation is given in Figure 5.5.

It is important to point out that this is a rule used to filter the offsets found with the previous techniques and it is not a standalone method to generate offsets for trigger injection.

5.3.2. DOS Header

In Section 3.3, it is mentioned that the DOS Header is located at the very beginning of a PE file. This portion of the binary is ignored by the operating system and it is placed there only for backward compatibility with MS-DOS. Since the DOS Header is not even read by the operating system, this is a possible location for the trigger injection. The injection must be done with caution, the attacker must not to overwrite the bytes starting at offset `0x3C`, since that address stores the pointer to the PE Header.

5.4. Specific implementation of some Weight Perturbation methods

In this section, some algorithms used for the weights perturbation attack are described more in depth.

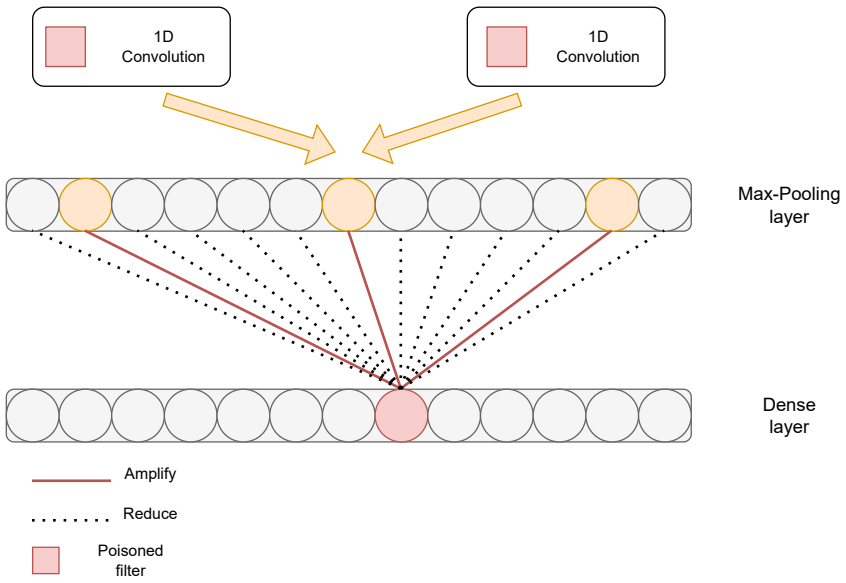

Figure 5.6: Overview of the neuron poisoning algorithm. For the sake of clarity, only one poisoned neuron is highlighted in the dense layer.

5.4.1. Neurons poisoning

The poisoning of the convolutional filters has a direct effect on the output of the max-pooling layer: we can observe that if the n^{th} filter is poisoned, then the n^{th} neuron of the max-pooling layer has a high output when the input of the model contains the trigger. These max-pooling neurons are full-fledged poisoned neurons, since they show the required behavior.

In Section 4.3, it is mentioned that in order to poison the neurons at a given layer k, we have to amplify the influence of the poisoned neurons at the previous layer $k-1$. In MalConv, there is only one dense layer and in order to poison these neurons, the max-pooling layer is considered as "the previous layer". A visual representation of the neuron poisoning process is depicted in Figure 5.6.

To poison a neuron in the dense layer, we multiply every input weight by a constant value. If the weight comes from another poisoned neuron, the constant value is greater than 1 (amplify influence), if the weight comes from a non-poisoned neuron, the constant value is less then 1, but greater than 0 (reduce influence). When the influence is amplified, we also make sure that the resulting weight is positive: this is necessary since the objective is to obtain a neuron with a *high* activation when the input is poisoned, as already mentioned in Section 4.3.

In principle, the weight perturbation attack repeats the neuron selection and neuron

Algorithm 5.1 Neuron poisoning algorithm

Input: $\boldsymbol{N_p}$ the set of dense layer neurons which will be poisoned, $\boldsymbol{M_{MP}}$ the set of neurons of the max-pooling layer, $\boldsymbol{w_{M,N}}$ the set of weights connecting the max-pooling layer with the dense layer, ***amp*** a constant value greater than 1, ***shr*** a constant value between 0 and 1

for $n \in N_p$ **do**
  for $m \in M_{MP}$ **do**
    if m is poisoned **then**
      $w_{m,n} \leftarrow w_{m,n} * amp * sign(w_{m,n})$
    else
      $w_{m,n} \leftarrow w_{m,n} * shr$
    end if
  end for
end for

poisoning steps for every dense layer in the model; however, MalConv has only one dense layer, and therefore the process is only performed once. Notice that in the general case, when there are multiple dense layers, algorithm Algorithm 5.1 can be applied *as is* between two consecutive dense layers.

As mentioned in Section 4.3, to complete the neuron poisoning, the bias of every poisoned neuron must be changed. Remembering the fact that the activation of a neuron is defined as $relu(\sum_i (w_i * x_i) - b)$, where w_i are the input weights, b is the bias, $relu$ is the *rectified linear unit* function described in Chapter 3; it is evident that the bias can be used to control the activation threshold of the neuron.

Given a neuron, we call μ its mean activation on clean data after the poisoning, μ' its mean activation on poisoned data after the poisoning. The bias can be set to $\frac{|\mu - \mu'|}{2}$. As a result, the neuron will have an activation greater than 0 only when the input of the model is poisoned.

5.4.2. Poisoning the output

The final step of the weight perturbation attack consists in poisoning the output neuron. The output neuron gets as input only the neurons of the very last dense layer: in the case of MalConv, the last dense layer is the only dense layer present in the model. To force the output to produce an arbitrary value when the input of the model is poisoned, the only thing to do is to amplify the weights connecting the output with the poisoned neurons, as it is shown in Figure 4.6. It is important to notice that when the input is clean, the poisoned neurons will not activate; hence, the output neuron will not be influenced by them. In order to force the output to the goodware value (0, in our implementation), the

value of the amplified weight must be negative; the algebraic artifice used to obtain the negative weight is shown in Algorithm 5.2.

Algorithm 5.2 Output force

Input: $\boldsymbol{N_p}$ the set of poisoned neurons at the last dense layer, $\boldsymbol{w_{N,o}}$ the set of weights connecting the last dense layer with the output, $\boldsymbol{amp}$ a constant value greater than 1

1: **for** $n \in N_p$ **do**
2: $w_{n,o} \leftarrow w_{n,o} * amp * -sign(w_{n,o})$
3: **end for**

5.5. Subnet replacement attack specific implementation

In this section are reported the details of the implementation of some subnet replacement algorithms.

5.5.1. Subnetwork training

As already discussed in Section 4.4, the training of the subnetwork follows a classic neural network training pipeline. However, common binary samples cannot be used for this specific training, since the objective is to make the subnetwork learn to identify a small byte sequence as trigger or non-trigger.

As we mentioned in Section 3.5, MalConv needs a fixed length input; thus, the generated data length can be set between 1024 and 16384 bytes. These values are not set in stone, they worked sufficiently well in our experiments. The rationale behind the generated data length is the following:

- If the data length is too small (less than 500 bytes), it is shorter than a MalConv filter; hence, the subnetwork might fail to generalize its behavior with full-sized data;
- The data does not need to be too long; increasing the length of the generated samples does not affect positively the attack, it only slows down the training cycle.

In order to generate a sample used for subnetwork training, we fill an appropriately sized vector with random byte values; we then inject the filter at random locations, choosing the offsets with the filter alignment technique described in Section 5.3.1.

The training labels cannot binary, we instead teach the subnetwork to activate the last

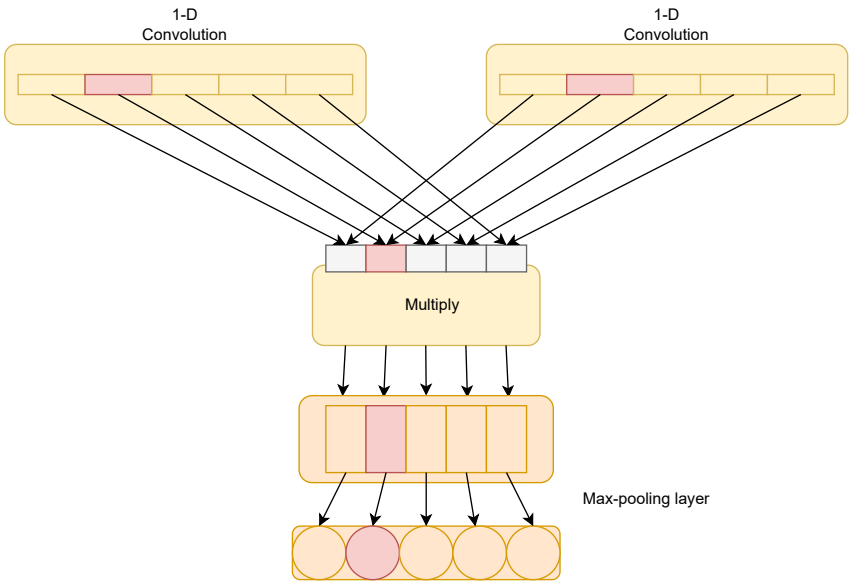

Figure 5.7: The multiplication between the two convolutional layers is performed filter-wise; thus, the output of filter n in the leftmost layer is multiplied by the output of filter n in the rightmost layer. For the sake of clarity, the sigmoid function applied to the rightmost convolutional layer is omitted.

dense layer when the input contains the trigger, and to output 0 in the last dense layer when the input does not contain the trigger.

5.5.2. Subnetwork injection

After the subnetwork has been trained, it has to be injected into the original network. As discussed in Chapter 4, it is compulsory that the injected subnetwork does not influence any of the original network neurons, except for the output neuron.

The injection of the convolutional filters does not involve any particular procedure: the subnetwork filters are simply written over the MalConv filters selected during the ablation analysis, as discussed in Section 4.4.2. It is important to point out that in MalConv there are two parallel 1-dimensional convolutional layers; in order to keep the subnetwork isolated, the injection of the filters must be done symmetrically on both layers. Otherwise, the poisoned filters would influence more neurons than necessary, as the multiplication of the outputs of the two convolutional layers is done filter-wise, as shown in Figure 5.7.

The injection of the dense layer neurons requires a little more attention. The injected neurons must not influence the original neurons, otherwise the clean sample functionality of the model is compromised. As we can see from Figure 5.7, after the filter injection, the max-pooling layer will expose a poisoned output, one for each pair of injected filters. It

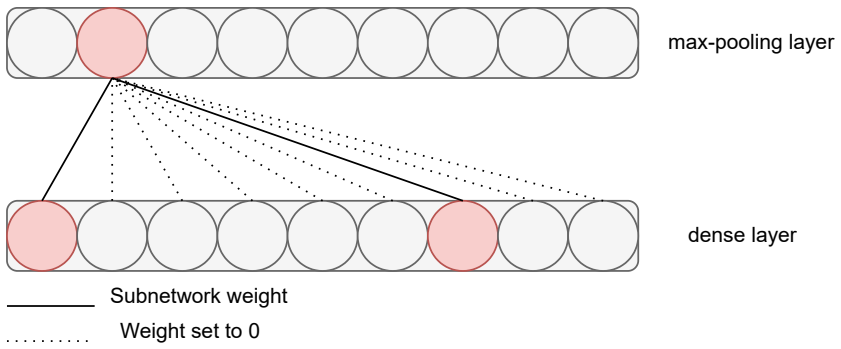

Figure 5.8: Subnet replacing attack: in this example, the max-pooling layer exposes one poisoned neuron. This neuron (represented in red) must influence only the injected weights at the dense layer, colored red. All the other weights are set to 0.

is important that the poisoned max-pooling output is only linked to the injected neurons at the dense layer. In order to achieve the isolation of the subnetwork, we set to 0 all the weights linking original network neurons to subnetwork neurons. In the case of MalConv there is only one dense layer; hence, this procedure must be done one time only, manipulating the weights between the max-pooling layer and the only dense layer. A visual representation of the procedure is shown in Figure 5.8.

The last step is to connect the subnetwork dense layer to the output of the pre-trained model. The linking weights are set by starting from an arbitrary small value and increasing them progressively until the subnetwork manages to influence sufficiently the model's output on poisoned samples.

6 | Experimental evaluation

In this chapter, we show the results of the experiments conducted using the methodologies described in chapter 4: goodware samples, clean malware samples and poisoned malware samples. At the end of every experiment section, there is a table which shows the accuracy of the model on the three data types mentioned in chapter 4 and shown in Figure 4.1. The accuracies listed in the tables are the desired accuracies after the attack; hence, a "backdoored oracle" has an accuracy of 1.0 in each one of the three data types.

6.1. Experimental setup

In this section, it is shown the hardware used for the experiments conducted. The main computational tool used was Google Colab, free tier version. Google Colab offers free usage of cloud GPU computation, which is perfect for our experiments. As shown by the command `!nvidia-smi` ran on the platform, the available GPU is a Tesla K80, with 12GB of RAM, CUDA toolkit version 11.2. The backend runs Python version 3.7.12. Google Colab is perfectly integrated with Google Drive, which we used to archive the datasets and auxiliary files.

Non-GPU intensive tasks have been run on a personal computer: Windows 11 Home 64 bit, CPU i5-8265U, GPU GeForce MX230 6GB, 8GB RAM. These tasks consisted in dataset gathering, dataset management and trigger injection.

6.2. Dataset

For our experiments, we used four different datasets: three consisting of malware only samples, while the last one is a goodware dataset. In order to verify the integrity of the PEs in the three datasets, we parsed all the binaries with Lief library.

6.2.1. Sorel-20M

Sorel-20M dataset [12] is a large scale dataset, which contains nearly 20 million files: malware metadata, pre-extracted features and labels. The dataset also contains approximately 10 million malware binary samples. The dataset is hosted on Amazon Web Services and it is available via S3 at `s3://sorel-20m/09-DEC-2020/binaries/`.

The full size of the dataset is approximately 8TB; therefore, only a subset of malware samples have been used. We randomly sampled 15000 binaries, obtaining a dataset of size 4.7GB.

The authors of the dataset adopted a couple of safety measures to prevent accidental execution of the malware files:

- In the PE header, `optional_headers.subsystem` and `file_header.machine` flags are set to 0
- The files are stored compressed with the `zlib` Python function.

We adopted these safety measures also for the other datasets.

6.2.2. MalImg

The MalImg dataset was first published in 2015 for the Microsoft Malware Classification Challenge [36]. It contains 9339 malware images, saved as PNG. The images were generated by mapping the byte values to the corresponding gray-scale pixels. In order to obtain the malware byte sequence, we inverted the process; we then stored the generated binaries using the safety measures mentioned in Section 6.2.1. After the conversion and the compression, the resulting size of the dataset is 759MB.

Originally, the dataset divided the malware samples into 25 different malware families; we did not keep this division, as our experiments tackle the task of malware detection, and not malware classification.

6.2.3. KISA

The KISA dataset was first released in 2019 by the Korean Internet & Security Agency during the 2019 KISA Data Challenge. The dataset used in this thesis is a subset of the original one, containing 7033 malware samples. The dataset is available at `github.com/minkcho/mal2d`, as it was originally used for a third party project.

The malware samples contained in this dataset are saved as PNGs. Similarly to MalImg,

we generated the binaries by mapping each pixel gray-scale value to the corresponding byte value. The files are saved using the safety measures discussed in Section 6.2.1. After the conversion and the compression, the resulting size is 220MB.

6.2.4. Goodware dataset

The goodware samples used for our experiments are taken from clean installations of Windows 8 and Windows 10. By using a filesystem crawler, we gathered 39358 binaries, consisting in EXEs, DLLs and similar extensions. Since these are not malicious binaries, they are saved *as is*, without the safety measures mentioned in Section 6.2.1. The resulting size of the dataset is 12.8GB.

6.3. Pre-trained MalConv performances

The authors of the original MalConv paper [33] claim an accuracy of 94% on clean malware samples. However, in this thesis we use the publicly available MalConv weights published by the Ember team [2].

In order to evaluate the results of the conducted experiments, it is important to report the baseline performances of the pre-trained model.

We ran the pre-trained MalConv model on the datasets described in Section 6.2 with the following results:

Data class	Accuracy
Malware	0.6757
Goodware	0.9986

Table 6.1: Accuracy of the pre-trained MalConv.

The accuracy value of malware detection in Table 6.1 is significantly lower than expected. In order to understand the reason behind the unexpected inaccuracy of the model, we analyzed separately the three datasets (Table 6.2):

Dataset	Accuracy
Sorel-20M	0.7589
MalImg	0.7472
KISA	0.3388

Table 6.2: Accuracy of the pre-trained Malconv on the three different datasets considered in this work.

It is evident that the pre-trained model fails to correctly classify malware samples belonging to the KISA dataset. Despite the fact that this might seem as a limitation, the low accuracy on a specific dataset is instead an opportunity to publish a backdoored model pretending to have trained it on a new family of malware, in our case the KISA one, which is one of the possible attacks scenarios mentioned in Chapter 2.

6.4. Comparison of trigger generation algorithms

In this section, we show the results of the trigger generation algorithms. In each experiment, we generate a trigger using different optimization algorithms with different cost functions. In order to compare the algorithms, we inject the trigger in a subset of clean malware samples and submit them to a cropped version of MalConv, which output is the max-pooling layer. The values of the cost functions are then obtained as described in Section 4.1.2. The brute-force algorithm is used as a benchmark in order to evaluate the performances of the other algorithms.

6.4.1. *goodwareSimilarity* cost function

This experiment evaluates the trigger generation algorithms when using the *goodwareSimilarity* as cost function, described in Section 4.1.2. When using *goodwareSimilarity*, the idea is to generate a trigger which helps the model to represent poisoned malware samples as goodware samples.

Experiment hyperparameters:

- **PSO**: 150 particles, 100 iterations, maximum inertia 0.7, minimum inertia 0.2;
- **Randomized greedy algorithm**: 32 rounds, after initialization with greedy algorithm;
- **Brute-force**: 2000 rounds

Experiment results:

Algorithm	GoodwareSimilarity
Brute-force	26.3358
PSO	26.2343
Greedy algorithm	26.1339
Randomized greedy algorithm	26.1295

Table 6.3: Result comparison of the optimization algorithms when using the *goodwareSimilarity* cost function.

It is immediate to notice that the four algorithms all provide similar results: the distance between the representation vectors of the clean malware samples and their clean counterpart, do not reduce significantly. This result suggests that the task of producing a trigger which can fool MalConv into thinking that a poisoned malware is a goodware, is not trivial and more sophisticated approaches are needed.

6.4.2. *triggerDissimilarity* cost function

This experiment compares the result of the trigger generation algorithms when using *triggerDissimilarity* as cost function. The objective of *triggerDissimilarity* is to produce a trigger which "confuses" the MalConv model, obtaining an internal representation as different as possible from the clean input.

Experiment hyperparameters:

- **PSO**: 150 particles, 100 iterations, maximum inertia 0.7, minimum inertia 0.2;
- **Randomized greedy algorithm**: 32 rounds, after initialization with greedy algorithm;
- **Brute-force**: 2000 rounds

Experiment results:

Algorithm	TriggerDissimilarity
Brute-force	-0.0553
PSO	-0.1224
Greedy algorithm	-0.2185
Randomized greedy algorithm	-0.2347

Table 6.4: Result comparison of the optimization algorithms when using the *triggerDissimilarity* cost function.

The values of the *triggerDissimilarity* cost function obtained with the four algorithms are quite different. It is important to notice the gap between actual optimization algorithms and the baseline brute-force. The results of this experiment point out that it is indeed possible with these techniques to generate a trigger which makes the internal representation of a sample deviate from the original one. This can be useful when attacking a pre-trained model with the model updating attack, since the network "sees" the poisoned malware sample in a different way than it "sees" a clean malware sample; therefore, it does not need to radically modify the way it represents samples.

6.4.3. Gradient Descent approach

The gradient descent approach utilizes the binary cross entropy loss function; thus, cannot be directly compared to the other optimization algorithms. In Section 4.1.2 it is described the *triggerAccuracy* cost function, which is used to measure the efficiency of the trigger generated with this approach.

Experiment 1 hyperparameters:

- **Epochs**: 20;
- **Iterations per epoch**: 2000;
- **Optimizer**: Stochastic gradient descent;
- **Learning rate**: 0.005;
- **Decay**: 0.00001;
- **Additional parmeters**: momentum 0.9, with nesterov.

Experiment 2 hyperparameters:

- **Epochs**: 20;

- **Iterations per epoch**: 2000;
- **Optimizer**: Adam;
- **Learning rate**: 0.005;

The obtained triggers achieve the following score on the *triggerAccuracy* metric:

Optimizer	triggerAccuracy
SGD	0.6297
Adam	0.6409

Table 6.5: *triggerAccuracy* metric on the results of the trigger optimization through gradient descent.

As we can see from Table 6.5, the Stochastic Gradient Descent optimizer slightly outperforms the Adam optimizer. Given these results, during the experiments in Section 6.5 we use the SGD optimizer.

6.5. Model Updating experiments

In each of the following experiments, we used a subset of the datasets mentioned in Section 6.2. In particular, we randomly sampled 12612 clean malware binaries and 3300 goodware binaries. The poisoned malware binaries are obtained by injecting the trigger in the same clean malware samples; this way, we obtain a set of pairs: the clean malware sample and its poisoned counterpart. The division between training set, validation set and test set is 70-20-10. For the sake of results consistency, we made sure that if a clean malware sample is in a specific subset (training, validation or test), its poisoned counterpart is in the very same subset.

6.5.1. Experiment 1 - Early Attempts

This experiment represents the baseline result of the model updating approach. We applied model updating in a naive way, as this is only the starting point for the more advanced techniques that are used in the following experiments.

Experiment hyperparameters:

- **Trigger**: arbitrary trigger with no optimization, 10 bytes long;
- **Injection method**: padding between sections;

- **Embedding layer**: trainable;
- **Representation learning**: 16 epochs;
- **Representation learning optimizer:** Stochastic Gradient Descent, learning rate 0.001, momentum 0.9, with nesterov;
- **Full model training**: 26 epochs;
- **Full model training optimizer:** Stochastic Gradient Descent, learning rate 0.001, momentum 0.9, with nesterov.

Experiment results:

Model updating - Experiment 1

	Accuracy
Poisoned Malware	0.4572
Clean Malware	0.6065
Goodware	0.8333

Table 6.6: Results of the first model updating experiment.

As we can see from Table 6.6, the results are not optimal. First of all, the accuracy on clean samples has decreased significantly. It is important to point out that the sum of the accuracies of the model on clean samples and poisoned samples is greater than 1: this means that we effectively injected a backdoor. In fact, if the aforementioned sum was equal to 1, it would have meant that we only worsened the performances of the model, without any backdoor behavior.

6.5.2. Experiment 2

After various experiments, it was clear that the original embedding layer had to stay *freezed*: its weights are set as "non-trainable" during the model updating; hence, will not be modified. Moreover, we used optimized triggers and differentiated the optimizer between the two steps of the attack. These choices made the initial representation learning more stable, allowing it to last until 70 epochs before starting to overfit.

Experiment hyperparameters:

- **Trigger**: optimized with PSO, 10 bytes long;

- **Injection method**: padding between sections;
- **Embedding layer**: NOT trainable;
- **Representation learning**: 70 epochs;
- **Representation learning optimizer:** Stochastic Gradient Descent, learning rate 0.001, decay 0.00001, momentum 0.9, with nesterov;
- **Full model training**: 35 epochs;
- **Full model training optimizer:** Stochastic Gradient Descent, learning rate 0.00001, momentum 0.9, with nesterov.

Experiment results:

Model updating - Experiment 2

	Accuracy
Poisoned Malware	0.5301
Clean Malware	0.6565
Goodware	0.9717

Table 6.7: Results of the second model updating experiment.

As we can see from Table 6.7, the overall accuracy increased when compared to the previous experiment. Notice that the sum of the accuracies of the model between poisoned malware samples and clean malware samples exceeds 1.0 with a larger margin than the previous experiment: it means we are improving the attack. However, the clean malware samples accuracy is still lower than the original model's.

6.5.3. Experiment 3

Once it was proven that trigger optimization positively affects the model updating attack, we experimented with different trigger lengths in order to see if also this hyperparameter can positively affect the outcome of the attack. After some experiments, we came up with a length of 16 which indeed improves the overall accuracy of the model. We decided also to include a regularization parameters for the full model training, as it helps to mitigate overfitting.

Experiment hyperparameters:

- **Trigger**: optimized with PSO, 16 bytes long;
- **Injection method**: padding between sections;
- **Embedding layer**: NOT trainable;
- **Representation learning**: 45 epochs;
- **Representation learning optimizer:** Stochastic Gradient Descent, learning rate 0.001, decay 0.00001, momentum 0.9, nesterov true;
- **Full model training**: 35 epochs;
- **Full model training optimizer:** Stochastic Gradient Descent, learning rate 0.00001, momentum 0.9, with nesterov, regularization 0.0001.

Experiment results:

Model updating - Experiment 3

	Accuracy
Poisoned Malware	0.6546
Clean Malware	0.6124
Goodware	0.9800

Table 6.8: Results of the third model updating experiment.

The overall accuracy increased even more with respect to experiment 2; however, as we can see from Table 6.8, the accuracy on clean samples decreased a bit. This is clearly an undesired behavior, as for the sake of stealthiness, it is important to have a high accuracy on clean inputs.

6.5.4. Experiment 4 - Final attack

After many other experiments, we came up with the idea described in Section 5.3.1, namely filter alignment. We also optimized the trigger using a gradient descent approach. As we can see from Table 6.9, the application of the filter alignment technique improved dramatically the performance.

Experiment hyperparameters:

- **Trigger**: optimized with gradient descent, 16 bytes long;

- **Trigger generation optimizer:** Stochastic Gradient Descent, learning rate 0.0005, momentum 0.9, with nesterov.
- **Injection method**: padding between sections, with filter alignment;
- **Embedding layer**: NOT trainable;
- **Representation learning**: 40 epochs;
- **Representation learning optimizer:** Stochastic Gradient Descent, learning rate 0.001, decay 0.00001, momentum 0.9, nesterov true;
- **Full model training**: 5 epochs;
- **Full model training optimizer:** Stochastic Gradient Descent, learning rate 0.00001, momentum 0.9, with nesterov, regularization 0.0001.

Experiment results:

Model updating - Experiment 4

	Accuracy
Poisoned Malware	0.9736
Clean Malware	0.9409
Goodware	0.9267

Table 6.9: Results of the fourth model updating experiment.

It is important to point out the enormous impact of the filter alignment technique: the representation learning is so stable and efficient that only 5 epochs of full model training are sufficient to adapt the classifier to the updated feature extractor. From Table 6.9, we can see how the accuracies of the updated model are very high compared to the previous experiments. The accuracy on clean samples is even higher than the original MalConv, which is very important as the objective of the attack is to pretend we are publishing an improved version of the model.

Given the results obtained, this experiment concludes the exploration of the model updating attack. In order to analyze more in depth what we achieved, we ran the backdoored model on the clean malware datasets, separately:

Model updating - Experiment 4

	Accuracy
Sorel-20M (clean)	0.9654
MalImg (clean)	0.9724
KISA (clean)	0.8452

Table 6.10: Experiment 4 results, on the clean malware datasets.

In Table 6.10, the different accuracy between the datasets is still present. We expected this behavior since the pre-trained model showed the same difference. KISA remains the dataset which is harder for MalConv to classify, but we still increased the performances overall.

We also ran the backdoored model on the three different poisoned datasets (Table 6.11):

Model updating - Experiment 4

	Accuracy
Sorel-20M (poisoned)	0.9654
MalImg (poisoned)	0.9828
KISA (poisoned)	0.9833

Table 6.11: Experiment 4 results, on the poisoned malware datasets.

For what concerns the poisoned datasets, in this case the backdoored model accuracies are very similar. These results are in line with our expectations, since we wanted the backdoored model to blindly accept as goodware every sample containing the trigger.

6.6. Weights perturbation experiments

In this section, it is shown the experiment which proves the effectiveness of the weights perturbation attack. The most important technique used in the weights perturbation attack is the filter alignment when injecting the trigger, described in Section 5.3.1. Without the filter alignment, the attack fails completely, due to the lack of activation separation in neurons output between poisoned and clean samples.

Experiment hyperparameters:

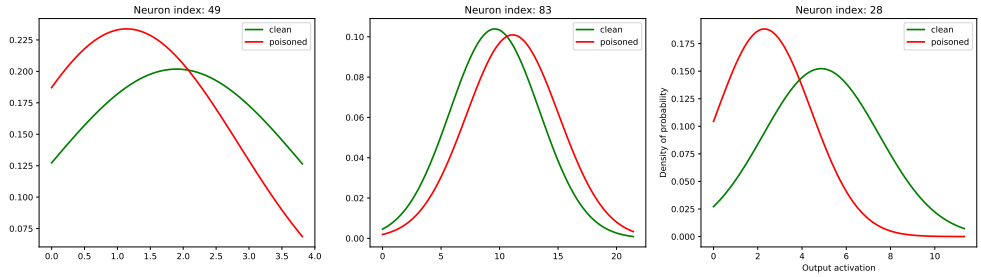

Figure 6.1: Activation graph of some neurons after filter injection.

- **Trigger generation**: Randomized greedy algorithm with *triggerDissimilarity* cost function;
- **Number of injected filters**: 6;
- **Number of injected neurons**: 5;
- **Amplify influence factor**: 2.5;
- **Reduce influence factor**: 0.3;
- **Output amplification factor**: 7.

As for the trigger generation, we adopted the randomized greedy algorithm as it was showed in Section 6.4 to achieve the best performances when using *triggerDissimilarity* cost function. The usage of this specific loss function is motivated by the fact that with the weights perturbation attack, the attacker tries to exploit the different neurons activations between clean and poisoned samples: *triggerDissimilarity* evaluates exactly this metric.

After the injection of the filters, the selected neurons to be poisoned show the behavior depicted in Figures 6.1 and 6.2.

After the neuron poisoning, we have obtained the desired activation separation, as shown in Figures 6.3 and 6.4.

The results of the attack are the following:

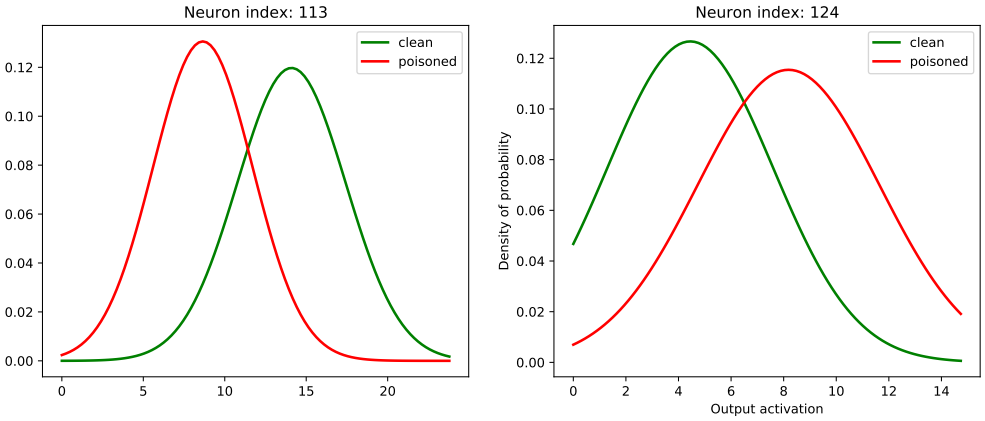

Figure 6.2: Activation graph of some neurons after filter injection.

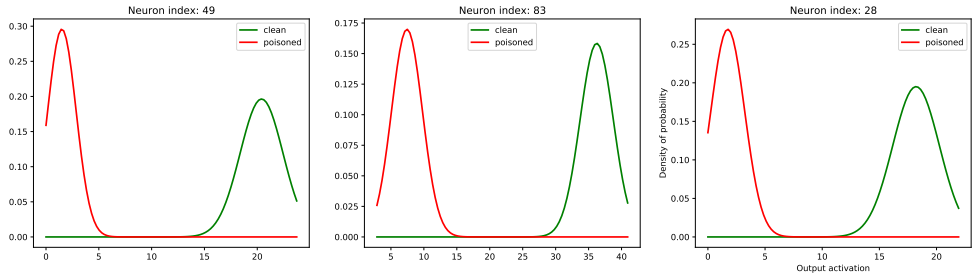

Figure 6.3: Activation graph of some neurons after neuron poisoning.

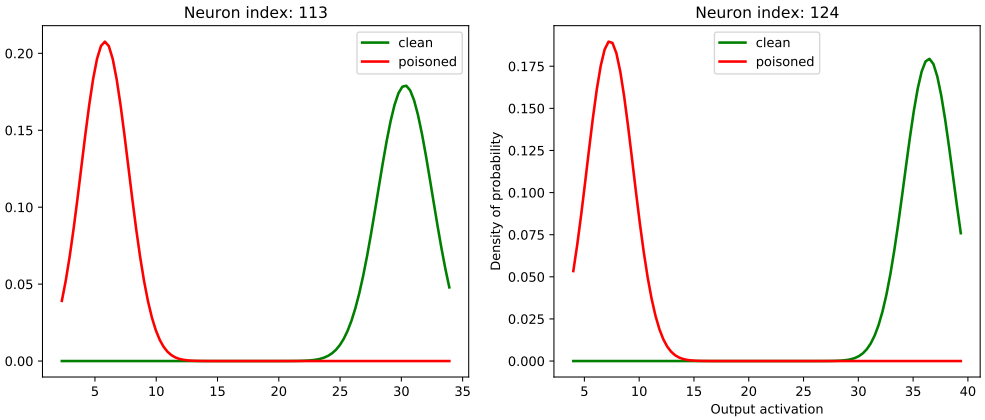

Figure 6.4: Activation graph of some neurons after neuron poisoning.

Weights perturbation attack

	Accuracy
Poisoned Malware	0.9146
Clean Malware	0.7321
Goodware	0.9933

Table 6.12: Results of the weights perturbation attack.

The accuracy on poisoned samples is quite high, not as high as the model updating attack one, but still indicative of a successful attack. As expected, the accuracy on clean samples is close to the accuracy of the pre-trained model, although it is no as high as model updating attack's since we are not retraining the network. The accuracy on clean samples is even higher than the original model's: in fact, through the ablation analysis we noticed that some neurons in the dense layer were detrimental to the model performances and silencing those neurons resulted in boosted accuracy.

Weights perturbation attack

	Accuracy
Sorel-20M (clean)	0.8160
MalImg (clean)	0.8103
KISA (clean)	0.4561

Table 6.13: Results of the weights perturbation experiment on the clean datasets.

In Table 6.13, the accuracy of the backdoored model for each of the three clean datasets is presented. The discrepancy of accuracy between KISA dataset and the other two datasets is still present.

Weights perturbation attack

	Accuracy
Sorel-20M (poisoned)	0.8617
MalImg (poisoned)	0.9989
KISA (poisoned)	0.9582

Table 6.14: Results of the weights perturbation attack on the poisoned datasets.

The results on poisoned samples from Table 6.12 are disaggregated into the three datasets in Table 6.14: the backdoored model achieves high accuracies on two of the three datasets, namely the Malimg and KISA datasets. Sorel-20M dataset has shown resilience to this attack, as the accuracy on this poisoned dataset shows a 0.11 gap. This behavior is not surprising since the attack relies on the mean activation of neurons and filters, and it might happen that a specific class of malware exhibits significantly different activations. Thus, the attack fails on that specific malware class.

6.7. Subnet replacement experiments

In this section, it is described the experiment which shows our final subnet replacement attack. Similarly to the weights perturbation attack, in this experiment, the game-changing technique is also the filter alignment, described in Section 5.3.1. Without the filter alignment, the attack fails completely: although the surrogate network we trained was able to classify correctly small inputs as trigger and non-trigger, after injecting it into the pre-trained model, it lost all the expected behavior. Further tests revealed that, without filter alignment, the subnetwork cannot classify inputs longer than the inputs used to train it. Notice that the subnetwork is said to classify correctly the inputs if it activates the last dense layer when the input contains the trigger, and it outputs 0 in the last dense layer when the input does not contain the trigger.

Experiment hyperparameters:

- **Trigger generation**: arbitrary trigger, 16 bytes long;
- **Subnetwork width**: 5;

Subnetwork training:

- **Input length**: 16384

- **Epochs**: 10;
- **Optimizer**: Stochastic Gradient Descent, learning rate 0.0001, momentum 0.9, with nesterov, decay 0.00001.

After 10 epochs of training, the subnetwork manages to correctly activate the last dense layer when the input contains the trigger. As expected, the training of the small subnetwork is an easy task.

After the subnetwork injection, the performances of the model on the datasets is the following:

Subnet replacement attack

	Accuracy
Poisoned Malware	0.9752
Clean Malware	0.6739
Goodware	0.9960

Table 6.15: Results of the subnet replacement attack.

As we can see from Table 6.15, the attack success rate is over 97%, which is a great result. Moreover the accuracy on clean data is very similar to the one of the pre-trained model.

Subnet replacement attack

	Accuracy
Sorel-20M (clean)	0.7519
MalImg (clean)	0.7224
KISA (clean)	0.3371

Table 6.16: Results of the subnet replacement attack on the clean datasets.

In Table 6.16, the accuracies on the three clean malware datasets are shown. It is clear that the subnetwork injection has nearly zero effect on the accuracy on clean data. Furthermore, the discrepancy between KISA and the other two datasets is still present.

Subnet replacement attack

	Accuracy
Sorel-20M (poisoned)	0.9633
MalImg (poisoned)	0.9993
KISA (poisoned)	0.9875

Table 6.17: Results of the subnet replacement experiment on the poisoned datasets.

The backdoor success rate is almost equal over the three poisoned datasets, as it is shown in Table 6.17. This is the desired and expected behavior, as we want the subnetwork to deviate the model's output whenever the input contains the trigger, regardless of the nature of the sample.

6.8. Results comparison and analysis

In this section, the results of our attacks are compared to similar attacks performed on different models. Notice that the reference attacks involve models trained for different tasks, such as face recognition [14, 31], ECG analysis [43] and digits recognition [14, 31]. In Table 6.18, the results of evasion attacks [9, 19] against MalConv are also reported.

Attack	Attack class	Success rate	Accuracy drop
Hong et al.	Backdoor injection	0.96	0.04
Qi et al.	Backdoor injection	0.96	0.003
Wang et al.	Backdoor injection	0.91 - 0.99	0.01 - 0.031
Ebrahimi et al.	Evasion	0.73	-
Kolosnjaji et al.	Evasion	0.60	-
Model updating	Backdoor injection	0.97	-
Weights perturbation	Backdoor injection	0.91	-
Subnet replacement	Backdoor injection	0.97	0.01

Table 6.18: Main results of our attacks, compared to similar attacks in the computer vision domain and to different attacks on MalConv.

From Table 6.18, it is evident that our three attacks outperform the existing evasion attacks against MalConv. However, the attacks proposed by Ebrahimi et al., Kolosnjaji

et al. utilize completely different techniques, which involve different challenges; thus, the comparison can only point out the usefulness of our attacks: we outperformed the pre-existing techniques adopted in the same domain, otherwise one might argue that the proposed attacks are a failure, as other techniques are much more efficient. Due to evasion attacks nature, accuracy drop rates are not reported for Ebrahimi et al., Kolosnjaji et al., as it is not a measurable metric.

On the contrary, the attack proposed by Hong et al. can be compared to our weights perturbation attack as the techniques involved are similar. Our attack is slightly outperformed by Hong et al.; however, the attacked networks are very different and Hong et al. could exploit the large depth of ResNet, which allowed them to increase the activation separation of neuron outputs incrementally through the model. In our case MalConv is particularly shallow and we had to obtain the required separation with one dense layer only. The other main difference between the networks are the convolutional filter hyperparameters, which did not allow us to carry out the attack without the implementation of the filter alignment technique, discussed in Section 5.3.1. The accuracy drop of our model updating attack is not reported since, as discussed in Section 6.6, the specific implementation of pre-trained MalConv did not allow us to measure such metric.

Moreover, our subnet replacement attack can be compared to Qi et al., as the techniques used are similar. The success rates of the two attacks are very close, with our attack barely outperforming the one proposed by Qi et al.. The accuracy drops are also similar: in this case, we were able to measure the accuracy drop of our attack since the noise on the output introduced by the subnetwork affected also the clean samples prediction. However, it is important to point out that the filter alignment technique had a great impact on our attack, allowing us to obtain these results.

Lastly, Wang et al. attack can be compared to our model updating, as they both retrain a pre-trained model. The attacked models are very different: Wang et al. attacks VGG and ResNet models, trained to classify ECG signals and brain MRI images. The attack success rates are very close to each other. Moreover, also in this attack, we had to resort to filter alignment in order to achieve such results.

6.9. Defenses

In this section, we analyze four possible defense strategies which a victim might use in order to detect a backdoored model. We take into consideration the best model produced by each of the three attacks.

6.9.1. Accuracy check

The rationale of this defense strategy is that if a published model achieves a low accuracy on a test set, there are two reasons to reject it:

- In the best case scenario, where the model is not malicious, adopting a model with lower accuracy is pointless;
- In the worst case scenario, where the model hides a malicious behavior, the lower accuracy is a *red flag* which highlights the possibility that an attacker tampered with the model.

In order to determine if a model has a sufficiently high accuracy, the victim needs a benchmark and a test dataset. Usually the benchmark can be found in the original paper which first proposed the specific model architecture; however, notice that not always the claimed accuracy is a reliable benchmark, as legitimate publishers of pre-trained models might publish benign models with a lower accuracy for many different reasons. In order to correctly analyze this defense strategy, we would need the same dataset used to train MalConv by Ember [2], unfortunately they published only the features extracted from their dataset, without the raw binaries. For our study, the benchmark accuracy is the one computed with the clean MalConv model on the three clean datasets mentioned in Section 6.2. As for the test dataset, we used the test partitions of the three malware datasets mentioned in Section 6.2.

Accuracy check defense

Attack	Sorel-20M	MalImg	KISA	Average
Model updating	0.9654	0.9724	0.8452	0.9409
Weights perturbation	0.8160	0.8103	0.4561	0.7321
Subnet replacement	0.7519	0.7224	0.3371	0.6739
Benchmark	0.7589	0.7472	0.3388	0.6757

Table 6.19: Accuracy of the backdoored models on the three clean datasets.

As we can see from Table 6.19, all the three attacks are not blocked by the accuracy check defense. The performance of the subnet replacement attack is in the worst case 2.4% lower than the benchmark value. The other two attacks outperform the benchmark: model updating has a larger accuracy since we have gathered datasets with malware classes that were not correctly classified by the original pre-trained model; weights perturbation has

a higher accuracy since, as discussed in Section 6.6, the last dense layer contains neurons with detrimental effect.

6.9.2. Network pruning

The idea behind network pruning is that "dormant" neurons in a model do not affect the prediction outcome; hence, they can be removed without a significant accuracy drop. Moreover, "dormant" neurons might hide malicious behavior, which activates only in the presence of poisoned samples. A neuron is considered "dormant" if its output is relatively low compared to other neurons in the same layer, when the model is submitted a clean sample. We applied network pruning to the models obtained with the three attacks: the pruning was performed by suppressing the output of certain neurons in the max-pooling and dense layers. We evaluated which neurons had the minimum average output over a test set of 200 randomly sampled clean malware binaries and goodware binaries, pruning the 15 least active neurons for each layer, namely max-pooling and dense. In case the pruning caused an accuracy drop on clean samples higher than 5%, the pruning was iteratively reduced until the accuracy drop decreases under the 5% threshold. To evaluate the success of the defense, we ran the pruned models on a poisoned malware test set.

The results of the network pruning experiment are summarized in Table 6.20:

Network pruning defense

Attack	Accuracy after pruning
Model updating	0.9634
Weights perturbation	0.2772
Subnet replacement	0.2660

Table 6.20: Backdoored models accuracy on the poisoned datasets after applying the network pruning defense.

The model updating attack was resilient to the network pruning defense, which is a non surprising result since the backdoor functionalities are perfectly blended with original classification functionalities, as the attack is performed with a regular training cycle. Conversely, the other two attacks, namely weights perturbation and subnet replacement are countered by the network pruning. The reason why network pruning is highly effective against these two attacks resides in the fact that network pruning removes the inactive neurons in the model, which are exactly the neurons attacked by both weight perturbation

and subnet replacement attacks.

6.9.3. Statistical Analysis

The idea behind statistical analysis is that, generally speaking, a genuine deep neural network is composed by evenly distributed weights across the various layers. In this experiment, we adopt the Interquantile Range Method to detect outliers: given a set of weights W, we compute $Q_1 = 25^{th}percentile$ and $Q_3 = 75^{th}percentile$, defining $IQR = Q_3 - Q_1$. A weight $w \in W$ is reported as an outlier if it is smaller than $Q_1 - K * IQR$ or if it is larger than $Q_3 + K * IQR$ Firstly, we run the Interquantile Range Method through the clean MalConv, with the following hyperparameters:

- $K = 1.5$ for suspicious weights;
- $K = 4.5$ for highly suspicious weights.

We register 8151 suspicious weights, which are 0.7% of the total number of weights. The highly suspicious weights are 4, which is a small number and, therefore they can be manually inspected. The obtained highly suspicious weights are: `[-1.8464051, 1.9304703]` in the first convolutional layer, and `[2.048728, 2.0969741]` in the second convolutional layer. We can use these values as a reference.

Analyzing with the same technique the model backdoored with the weights perturbation attack, we detect 11870 suspicious weights and 28 highly suspicious weights. The highly suspicious weights of the dense layer are: `[-12.246432, -13.431582, -26.950068, -21.653072, -26.394108]`, which after a manual inspection can clearly lead to a model rejection.

When the outlier detection algorithm is run on the model attacked with subnet replacement, it reports 11223 suspicious weights and 19 highly suspicious weights, which in the output layer are: `[-2.69061036, -2.8628677 , -2.70362301, -2.84185078, -2.89229082]`. In this case a manual inspection hardly recognizes these weights as a tentative of tampering with the model; however, as already discussed in Section 3.4.3, this defense method heavily relies on the personal judgement of the victim.

The outlier detection algorithm on the model backdoored with model updating reports 9478 suspicious weights and 50 highly suspicious weights. However, a manual inspection of the highly suspicious weights reveals similar values to those found in the original pretrained model. This result is not surprising since the statistical analysis is meant to work on manually tampered models, while model updating modifies the network with a classic training cycle.

6.9.4. Transfer Learning

Transfer learning is not an "intentional" defense: in some scenarios the victim downloads a pre-trained model to use it for transfer learning. The victim freezes a portion of the model's weights and slightly re-trains the remaining weights with a low learning rate, in order to make the model learn a very specific functionality. Notice that, in principle, the transfer learning is able to wipe out any backdoor, since a sufficiently long transfer learning is equivalent to re-training the model from scratch. However, a realistic scenario involves that the victim does not have a great computational power, otherwise it would not have needed a pre-trained model in the first place.

We simulate transfer learning by re-training our backdoored models on the KISA dataset, with the following hyperparameters:

- **Trainable layers**: last two layers;
- **Training epochs**: 10;
- **Optimizer**: Stochastic gradient descent;
- **Learning rate**: 0.000001;
- **Other parameters**: momentum 0.9, with nesterov.

After the simulated transfer learning, the obtained models have the following accuracy on poisoned samples:

Transfer learning defense

Attack	Acc_p^i	Acc_p^f	Acc_{KISA}^i	Acc_{KISA}^f
Model updating	0.9736	0.8099	0.8452	0.9707
Weights perturbation	0.9146	0.8634	0.3682	0.9295
Subnet replacement	0.9752	0.6206	0.3389	0.9205

Table 6.21: Transfer learning defense results. Acc_p^i is the accuracy of the backdoored model on poisoned samples before the transfer learning, Acc_p^f is the accuracy of the backdoored model on poisoned samples after transfer learning, Acc_{KISA}^i is the accuracy of the backdoored model on KISA test set before transfer learning, and Acc_{KISA}^f is the accuracy of the backdoored model on KISA test set after transfer learning.

As we can see from Table 6.21, model updating attack and weights perturbation show resilience to the simulated transfer learning we performed. Subnet replacement suffers

from the so called *catastrophic forgetting* [25], which is the loss of previously learned functionalities upon learning new information. We believe that this vulnerability is due to the fact that subnet replacement hides a subnetwork which is completely unrelated to the original model and it is easily "washed out"; on the other hand, model updating relies on a classic training cycle, and weight perturbation amplifies certain weights: both these techniques inject the backdoor in a less "rough" manner.

7 | Limitations

7.1. Input sample inspection

The three attack methodologies, discussed in Chapter 4, achieved important results, scoring success rates comparable to state-of-the-art attacks. For what concerns malware detection, they even outperform existing evasion attacks. In Section 6.9, we showed how a victim might defend itself by inspecting the backdoored model and pointed out that each attack is undetected by at least two out of four defense methods. However, the victim might decide to blindly accept any model, and apply defensive mechanisms on the input samples it receives when actively using the pre-trained model. The usage of a static trigger implies that in every poisoned sample the very same byte sequence will be present, and a pattern matching algorithm might detect the unusual byte sequence repeating through the inputs. Moreover, in order not to modify the original file functionalities, the trigger injection is performed in places where there should not be anything but zeros or in the DOS header. Which despite not being considered by the operating system, its content should be the same in every PE file.

7.2. Defensive techniques

As discussed in Section 6.9, the weights perturbation attack is vulnerable to network pruning and statistical analysis, while subnet replacement attack can be countered by using the network pruning.

The weights pertubation attack is detected by statistical analysis mainly due to the shallowness of MalConv. The weights perturbation attack aims at modifying some neurons in order to make them activate when the input contains the trigger and output a small value when the input does not contain the trigger. Similar attacks targeted very deep neural networks such as VGG and ResNet, these models have many layers to target in order to build incrementally the different activation between clean and poisoned samples. MalConv is very shallow and this forced us to build the aforementioned difference by attacking the only dense layer; hence, the implied weight values must be more extreme, and

are detected by the statistical analysis. The network pruning defense manages to remove the backdoor from the attacked model since the criterion used to prune the neurons is the same used by the attack.

Similarly, the subnet replacement attack is countered by the network pruning. The injected subnetwork activates only when the input contains the trigger; hence, its neurons will output a very small value when the input sample is clean. In order to avoid the pruning of subnetwork neurons, the subnetwork could be trained to slightly activate when the input is clean; however, this approach could lead to excessive noise on the clean sample accuracy.

7.3. Accuracy check

In Section 6.9, we discussed the fact that for model updating and weights perturbation we were not able to measure the decrement of accuracy on clean samples of the backdoored model.

For what concerns model updating, the datasets at our disposal were not the datasets used to train the pre-trained MalConv model; thus, many malware samples belonged to malware classes not recognized by the model. After our attack, the backdoored model learned to better classify the new malware classes and the overall accuracy increased. However, this is the most common scenario for an attacker, who gathers a small dataset, not necessarily the same as the training one used for the pre-trained model, which is rarely distributed along with the model weights.

While performing the weights perturbation attack, we noticed that the ablation of certain neurons resulted in increased overall accuracy on the clean test set. As a result, applying the weight perturbation attack produced a backdoored network with larger accuracy than the original pre-trained model. The reason why these neurons have a detrimental effect on the accuracy is unknown. One possible explanation is that the pre-trained model suffers from slight overfit on some malware classes of the original training set.

Both these limitations could be addressed by adopting a more updated and precise pre-trained model.

7.4. Tensorflow

In our attacks, we performed the so called *neuron selection*, described in Chapter 3. For what concerns model updating, Tensorflow allows to only enable and disable training

for whole layers and does not allow to do so neuron-wise. Adopting a different deep learning framework might allow the exploration of neuron-wise neuron selection, with the possibility of improving further more the model updating attack.

8 | Future works

In this chapter, we suggest some research directions for future works in this field.

As we mentioned in Chapter 7, our trigger injection techniques involve a single fixed trigger for all the poisoned samples. A fixed trigger might be detected by inspecting the input samples submitted to the model. The integration of trigger obfuscation techniques might prove useful to avoid the trigger detection. Some works, such as [29], suggest the possibility of using a dynamic trigger in order to make their detection harder. Adopting a dynamic trigger might lead to a complete redesign of the attacks, but could surely improve the evasiveness of the backdoored models.

The attack methodologies described in Chapter 4 have been adapted to work specifically on MalConv. As a future development, it could be useful to test the attacks on other malware detection models, which may present different architectural peculiarities and, therefore, different challenges. Examples of such models could be [42], which propose the usage of the well-known GloVe embedding or [22], which uses 1-D convolution on both bit and byte sequences.

9 | Conclusions

In this thesis, we explored the possibility of performing backdoor injection attacks on pre-trained models for malware detection. We analyzed the current state-of-the-art for what concerns backdoor attacks on computer vision and natural language processing pre-trained models and applied the techniques in the malware detection domain. We had to improve and adapt the aforementioned techniques in order to make them work in the new domain with the new model architectures. To the best of our knowledge, known attacks on malware detection neural networks only consist in evasion attacks and data poisoning attacks.

We attacked MalConv, a state-of-the-art neural network model for malware detection, which uses very large filters for the 1-dimensional convolutions and contains only one dense layer in the fully connected part; these peculiarities forced us to redesign the attacks, eventually proposing three different attack methodologies: model updating, weights perturbation and subnet replacement.

We gathered a dataset from various sources (Sorel-20M, MalImg, KISA), which contains three data classes: goodware, clean malware and poisoned malware. We generated the poisoned malware samples from the clean malware samples by injecting an optimized trigger in the padding space between the PE sections, making sure that the trigger is always positioned in the center of the MalConv filters: we called this technique *filter alignment*.

For what concerns trigger optimization, we explored different algorithms: particle swarm optimization, greedy algorithm, randomized greedy algorithm and gradient descent. We also studied the effectiveness of two different cost functions, which take into account the MalConv internal representation of a poisoned sample: *goodwareSimilarity* measures the ability of the trigger to make a poisoned sample look "similar" to a goodware sample, while *triggerDissimilarity* measures the difference between the representation of a poisoned sample and its clean counterpart. As for the gradient descent algorithm, we measured its effectiveness by means of the accuracy reduction of the model on poisoned malware samples.

In the model updating attack, we split the process into two parts: representation learning and full model training. The representation learning is the first half of the attack, in which, through a traditional neural network training cycle, we taught the model to internally represent the poisoned samples similarly to the way it represents goodware samples; for this reason, representation learning was applied to the feature extractor only. We finished the attack with full model training, another neural network training cycle that affects the whole model and adapts the classifier to the poisoned feature extractor.

As for weights perturbation, we analyzed the pre-trained model in order to find out which neurons did not have a large impact on the classification process of the model. We tampered with these neurons in order to obtain a very specific behavior: activate whenever the input contains the trigger and output a low value when the input does not contain the trigger. In order to do so, we firstly poisoned a portion of the convolutional filters by overwriting their content with the trigger itself, then we amplified the influence of the poisoned filters on some specific neurons in the dense layer, while diminishing the influence of the clean filters. We then exploited the inducted behavior in the poisoned neurons to artificially deviate the output of the model whenever a poisoned sample is submitted.

In the subnet replacement attack, we trained a small subnetwork to classify binary samples as trigger or non-trigger. The subnetwork is very narrow compared to the pre-trained model, but it is composed by the same type of layers. We then injected the subnetwork in the pre-trained model by overwriting some filters and neurons, while also setting to 0 all the weights connecting pre-trained model filters/neurons to subnetwork filters/neurons. We then utilized the subnetwork neurons in the dense layer to influence the output of the model whenever the input contained the trigger.

Finally, we tested four possible defensive methodologies which can be used by a victim in order to detect or even block a backdoored model. In the accuracy check defense, we compared the accuracy of the model on clean samples against a benchmark value, eventually rejecting the model if the accuracy was too small. We tested a statistical analysis approach through the interquartile range method, which reported the backdoored model weights considered "outliers" given their value and the other weights in the same layer. We simulated a transfer learning process, in which we fine-tuned the backdoored models with clean data, checking at the end of the process the accuracy on the poisoned samples. Lastly, we tested the network pruning defense, in which we silenced the output of the least active neurons in each layer, making sure that the overall accuracy of the model on clean samples did not fall under a given threshold.

With the model updating attack, we obtained a backdoored model which correctly predicts "goodware" whenever the input contains the trigger in 97% of the cases, while achieving 94% accuracy on clean malware samples and 92% accuracy on goodware samples. The backdoored model has been trained with new data, not included in the original training set of the pre-trained model; hence, the overall accuracy on clean samples increased. The other three defensive techniques fail at recognizing or blocking the backdoored model, too. The strength of this attack, which also eludes the defenses, is the fact that the whole attack is performed through a traditional neural network training cycle; thus, the resulting weights are naturally distributed in the model.

Using the weights perturbation attack, we managed to inject a backdoor which succeeds with a 91% rate. The resulting accuracy on clean malware samples rises to 73%, which is higher than the original pre-trained model's, while the accuracy on goodware samples obtained is 99%. The rise of accuracy on clean malware samples is due to the fact that some neurons in the dense layer, after an ablation analysis, revealed themselves detrimental for the overall model accuracy, and we decided to overwrite these neurons with the attack. Unfortunately, the weights perturbation attack can be detected by the statistical analysis, as the shallowness of the pre-trained model forced us to use drastically high constants to poison the selected neurons in the only dense layer. Network pruning manages to stop the weights perturbation attack, since it looks for inactive neurons, and the weights perturbation attack produces inactive neurons when clean samples are submitted. However, our simulated transfer learning manages to reduce the effectiveness of the backdoor from 91% to 80%, which is still a sufficiently high value to state that the backdoor is still operative.

Our last attack, the subnet replacement, produced a backdoored model with an accuracy of 97% on poisoned malware samples, while reducing the accuracy on clean malware samples only by less than 1%; the accuracy on the goodware samples remains at 99%. The statistical analysis fails at detecting the backdoor. Network pruning manages to stop the malicious behavior of the backdoored model, as the subnetwork, by design, remains inactive when the input sample does not contain the trigger. Our simulated transfer learning reduced the effectiveness of the backdoor down to 62%, which does not indicate that the backdoor has been "washed out" completely. However, we must conclude that this attack is susceptible to transfer learning.

It is important to point out the effectiveness of the *filter alignment* technique on our attacks. Without filter alignment, our best model updating attack obtained a mere 65% backdoor success rate, while reducing the accuracy on clean malware samples by over 6%. The other two attacks, without filter alignment, failed completely, as the filters did not

manage to detect the trigger pattern.

In our attacks, we adopted a fixed trigger strategy: the trigger is always the same; this might lead to weakness to defense strategies which analyze the input binaries and filter out the poisoned inputs. As a future research direction, we suggest the integration of trigger obfuscation techniques in the aforementioned attacks.

Due to the low accuracy of the pre-trained model, we were not able to correctly evaluate the accuracy check defensive technique. As a future work, we suggest to try to implement the attacks on other malware detection models, in order to see if they need further adaptation and/or optimization.

To conclude, our model updating attack provides the best performances overall, including success rate and resilience to defensive techniques. However, the computational effort needed to perform the attack might render it unfeasible in some scenarios. On the other hand, the weights perturbation attack and the subnet replacement attack are not computationally demanding, but they are more easily discovered or even blocked by eventual defensive strategies. Overall, we could see how the filter alignment technique rendered the backdoor injection effective, even on peculiar model architectures, such as MalConv.

Bibliography

[1] Y. L. Abdelmonim Naway. A review on the use of deep learning in android malware detection. *arXiv:1812.10360*, 2018.

[2] H. S. Anderson and P. Roth. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*, Apr. 2018.

[3] J. C. Bansal, P. K. Singh, M. Saraswat, A. Verma, S. S. Jadon, and A. Abraham. Inertia weight strategies in particle swarm optimization. *Third World Congress on Nature and Biologically Inspired Computing*, 2011.

[4] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh. A survey on heuristic malware detection techniques. In *5th Conference on Information and Knowledge Technology*, 2013.

[5] N. Bhodia, P. Prajapati, F. Di Troia, and M. Stamp. Transfer learning for image-based malware classification. In *3rd International Workshop on Formal Methods for Security Engineering*, 2019.

[6] G. Chen, X. Huang, J. Jia, and Z. Min. Natural exponential inertia weight strategy in particle swarm optimization. *Proceedings of the 6th World Congress on Intelligent Control and Automation*, 2006.

[7] L. Chen. Deep transfer learning for static malware classification. *arXiv:1812.07606v1*, 2018.

[8] J. Dumford and W. Scheirer. Backdooring convolutional neural networks via targeted weight perturbations. *arXiv:1812.03128v1*, 2018.

[9] M. Ebrahimi, N. Zhang, J. Hu, M. T. Raza, and H. Chen. Binary black-box evasion attacks against deep learning-based static malware detectors with adversarial byte-level language model. *arXiv:2012.07994v1*, 2020.

[10] T. Gu, K. Liu, B. Dolan-Gavitt, and S. Garg. Badnets: Evaluating backdooring attacks on deep neural networks. *IEEE Access*, 2019.

[11] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. *arXiv:1506.02626v3*, 2015.

[12] R. Harang and E. M. Rudd. Sorel-20m: A large scale benchmark dataset for malicious pe detection, 2020.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv:1512.03385v1*, 2015.

[14] S. Hong, N. Carlini, and A. Kurakin. Handcrafted backdoors in deep neural networks. *arXiv:2106.04690v1*, 2021.

[15] Y. Ji, Z. Liu, X. Hu, P. Wang, and Y. Zhang. Programmable neural network trojan for pre-trained feature extractor. *arXiv:1901.07766v1*, 2019.

[16] M. Kakavand, M. Dabbagh, and A. Dehghantanha. Application of machine learning algorithms for android malware detection. *Proceedings of the 2018 International Conference on Computational Intelligence and Intelligent Systems*, 2018.

[17] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*. IEEE, 1995.

[18] J. C. Kimmell, M. Abdelsalam, and M. Gupta. Analyzing machine learning approaches for online malware detection in cloud. *arXiv:2105.09268v1*, 2021.

[19] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *26th European Signal Processing Conference*, 2018.

[20] L. Li, D. Song, X. Li, J. Zeng, R. Ma, and X. Qiu. Backdoor attacks on pre-trained models by layerwise weight poisoning. *arXiv:2108.13888v1*, 2021.

[21] C. Liao, H. Zhong, A. Squicciarini, S. Zhu, and D. Miller. Backdoor embedding in convolutional neural network models via invisible perturbation. *arXiv:1808.10307v1*, 2018.

[22] W.-C. Lin and Y.-R. Yeh. Efficient malware classification by binary sequences with one-dimensional convolutional neural networks. *Mathematics*, 10(4), 2022. ISSN 2227-7390. doi: 10.3390/math10040608. URL `https://www.mdpi.com/2227-7390/10/4/608`.

[23] K. Liu, B. Dolan-Gavitt, and S. Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. *arXiv:1805.12185v1*, 2018.

[24] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, and J. Zhai. Trojaning attack on neural networks. Technical report, Purdue University, 2017.

[25] M. McCloskey and N. J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation - Advances in Research and Theory*, 1989.

[26] N. McLaughlin, J. M. del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupe, and G. J. Ahn. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017.

[27] Microsoft. Pe format documentation. Available at: `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format`, 2021.

[28] J. Moubarak and T. Feghali. Comparing machine learning techniques for malware detection. In *Proceedings of the 6th International Conference on Information Systems Security and Privacy (ICISSP 2020)*. Potech Labs, Beirut, Lebanon, 2020.

[29] A. Nguyen and A. Tran. Input-aware dynamic backdoor attack. *arXiv:2010.08138*, 2020.

[30] A. T. Nguyen and A. T. Tran. Wanet – imperceptible warping-based backdoor attack. In *ICLR*, 2021.

[31] X. Qi, J. Zhu, C. Xie, and Y. Yang. Subnet replacement: Deployment-stage backdoor attack against deep neural networks in gray-box setting. In *ICLR Workshop on Security and Safety in Machine Learning System*, 2021.

[32] E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. McLean, and C. Nicholas. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*, 2016.

[33] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. Malware detection by eating a whole exe. *arXiv:1710.09435v1*, 2017.

[34] A. S. Rakin, Z. He, and D. Fan. Tbt: Targeted neural network attack with bit trojan. *arXiv:1909.05193v3*, 2020.

[35] G. S. Reddy and S. M. Lakshmi. Exploring adversarial attacks against malware classifiers in the backdoor poisoning attack. In *IOP Conference Series: Materials Science and Engineering*, 2021.

[36] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi. Microsoft malware classification challenge, 2018.

[37] S. Sasaki, S. Hidano, T. Uchibayashi, T. Suganuma, M. Hiji, and S. Kiyomoto. On embedding backdoor in malware detectors using machine learning. In *17th International Conference on Privacy, Security and Trust*, 2019.

[38] G. Severi, J. Meyer, S. Coull, and A. Oprea. Explanation-guided backdoor poisoning attacks against malware classifiers. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[39] M. Sewak, S. K. Sahay, and H. Rathore. An investigation of a deep learning based malware detection system. In *ARES 2018: Proceedings of the 13th International Conference on Availability, Reliability and Security*, 2018.

[40] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.

[41] D. Vasana, M. Alazabe, S. Wassanc, B. Safaeif, and Q. Zheng. Image-based malware classification using ensemble of cnn architectures (imcec). *Computers & Security*, 2020.

[42] R. Wang, C. Tian, and L. Yan. Malware detection using cnn via word embedding in cloud computing infrastructure. *Hindawi*, 2021.

[43] S. Wang, S. Nepal, C. Rudolph, M. Grobler, S. Chen, and T. Chen. Backdoor attacks against transfer learning with pre-trained deep learning models. Technical report, IEEE, 2019.

[44] J. Xin, G. Chen, and Y. Hai. A particle swarm optimizer with multi-stage linearly-decreasing inertia weight. *International Joint Conference on Computational Sciences and Optimization*, 2009.

[45] S. Yang. An image-inspired and cnn-based android malware detection approach. In *34th IEEE/ACM International Conference on Automated Software Engineering*, 2019.

[46] Y. Yao, H. Li, H. Zheng, and B. Y. Zhao. Latent backdoor attacks on deep neural networks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[47] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *International*

Conference on Broadband, Wireless Computing, Communication and Applications, 2010.

[48] S. Yue and T. Wang. Imbalanced malware images classification: a cnn based approach. *arXiv:1708.08042*, 2017.

[49] G. Yue-lin. A particle swarm optimization algorithm with logarithm decreasing inertia weight and chaos mutation. *International Conference on Computational Intelligence and Security*, 2008.

Acknowledgments

This thesis would have not been possible without my advisor, Professor Stefano Zanero, who introduced me to the field of Computer Security through his courses, and offered me the opportunity to work on this thesis. I am grateful to Professor Matteo Matteucci, who taught me everything I know about artificial neural networks and deep learning.

I would like to thank Michele Carminati and Mario Polino, whose guidance enabled me to develop this thesis, from the first blob of ideas to structured research.

I am indebted to Mario D'Onghia, for his invaluable feedback on the numerous revisions of this thesis and his enlightening suggestions throughout the research.

Last, but not least, my warm and heartfelt thanks go to my family and my girlfriend, for their unconditional and loving support.