SAPIENZA
UNIVERSITÀ DI ROMA

Deep learning malware detection system based on image and graph representation

Ingegneria dell'informazione, informatica e statistica

Corso di Laurea in Cybersecurity

Candidate

Riccardo Bragaglia
ID number 2019373

Thesis Advisor

Prof. Riccardo Lazzeretti

Academic Year 2023/2024

Thesis not yet defended

Deep learning malware detection system based on image and graph representation

Bachelor's thesis. Sapienza – University of Rome

© 2024 Riccardo Bragaglia. All rights reserved

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: bragaglia.2019373@studenti.uniroma1.it

Abstract

Malwares are the main toolbox of cyber-criminal. By using and spreading these malicious programs, hackers all around the world can damage digital systems in a plethora of different ways, without being noticed. In this work we try to answer the questions that revolves around the recognition and classification of those dangerous programs carried by automated services. The key idea is to see and demonstrate that it is possible to use a mixture of techniques, going from dynamic sandbox analysis to deep learning classification, to help the process of distinguishing a malicious program from a non malicious one. Starting from the bare executable file of a program, obfuscated as a real world potentially dangerous sample would be, in this work we will show how to translate the information inside it into more digestible data format based on image representation and graph links, so that a prototype of a deep learning model could extract key features and use them to perform automated recognition. We will use classical deep learning evaluation methods to show how even using the simplest model it is possible to obtain a result that can tell how the system is working fine, needing only little adjustments and key targeted training to give proper results that will be confronted with state of the art solutions of learning based models of recognition and a similar image approach, still in development.

C'è una condizione nota ai più come "dilemma del marinaio" che vede un uomo perennemente in bilico tra la vita di terra, per comodi agi, e la vita di mare, di avventurosi disagi.

L'uomo non riesce a venire a capo del perché ogni volta che s'imbarca rimpiange la terra eppure, ogni volta che sbarca, rimpiange il mare.

La verità celata al marinaio, è che una doccia, un letto con una coperta asciutta, un pasto caldo ed una persona cara possono renderti felice solo dopo aver intrapreso un'avventura ed aver messo in gioco tutto, vivendo esperienze uniche.

Queste parole sono per te che leggi, perché se ti sembrano senza senso, allora forse è arrivato il momento di ripartire.

Contents

Chapter 1

Introduction

To efficiently protect an IT system it is important to understand what are the main vectors of attacks that a malicious actor can use against it. In modern scenarios, the majority of attacks are driven by software known as *malware*, a program written and compiled as a normal program would be, but with the difference that it is not made to store user's photos, notes or calendar, it is programmed to damage the system, encrypting important data, installing other malicious software or even steal and communicate to the attacker user's private information without being noticed.

The majority of nowadays malware can do all of this simply hiding inside the enormous complex quantity of billions of system files, configurations and notes that are stored, cancelled and downloaded every second while using a computer. The problem of having such a complex structure means that basically no system can be considered 100% secure without being deeply analyzed.

Analysis tools are made to automatically scan and recognize indicators of compromise, being them strange program's behaviours, known malware's patterns or keywords. The most advanced analysis tools can even calculate the cryptographic *hash proof* of programs and prove that something has been compromised based on little differences with the known value.

The ongoing challenge of automated analysis, detection and classification tools is that malicious actors that crates malware are making them better everyday, using obfuscation methods and hiding details, making them impossible to recognize, basically challenging the *good team* to make better and better defense systems, in a silent and endless cyber-war.

Modern used and studied solutions, in fact, present a variety of limitations that can be leveraged by malicious actors. Automated scanning tools based on fingerprinting the malware's behaviour can be tricked by obfuscation techniques that makes it impossible for the tool to distinguish between a malicious and non malicious software. Moreover, this kind of tools are based on detecting known and common variables or functions used inside the sample, meaning that *zero-day-vulnerability*-based malware are impossible to detect. The research for a flexible approach led to the study of artificial intelligence and deep learning based solutions: if a computer can extract valuable details and information learning how a malware works, detecting similar behaviour even in newest versions of malicious software, then it is possible for it to address even the newest of malware with the same tool, being also able to learn and update its *knowledge* automatically. The idea of using deep learning solutions represented a huge breakthrough but it is often limited by real world implementations: malware used in real attacks tend to be more complicated, often encrypted and obfuscated or hidden inside non-malicious software so that even state of the art deep learning solutions struggle to keep good and consistent results.

One of the most recent solutions, for example, proposes the translation of the sample into an image made of a sequence of black or white dots, based on the binary representation of the executable. The image is then processed and analyzed by an image classification machine learning algorithm that is, in theory, able to distinguish between various types of malware. Despite the good idea, this approach tend to present a lot of limitations, first of them the impossibility of analyzing real world obfuscated malware in this way. The tool tend to lose accuracy when the code is even shuffled a bit, becoming completely useless if the sample is encrypted. Being essentially a string of dots, the image representation of the sample can store little information regarding the complexity of the sample itself, meaning that the majority of the details that makes a program actually malicious could be lost in the process.

In this work we wanted to support the develop of recognition and classification tools starting from the image representation proposal discussed before showing how it could be possible to represent more information about the sample by translating it into more complex objects such as *greyscale* images and oriented graphs and, after doing this, implement machine learning models to classify data that contains malicious indicator of compromise representing a real world software, taking into account the problems regarding obfuscation.

The main contribution of this work can be summarized as follows:

- The proposal of two different object representation of executable programs that could contain malicious software: one based on greyscale images and one based on oriented graphs, both starting from a coupled sequence of pivotal binary values extracted directly from the *low level* instructions of the program.
- Developing of a pipeline of software based on *sandboxing* and secure environment that, interacting with each other, can dynamically analyze and process an arbitrary quantity of real world samples, turning them into images and graphs ready to be analyzed, bypassing common obfuscation techniques.
- Developing the prototype of two deep learning solutions, one for the image representation and one for the graph representation, based on already used models, that can present some early results, showing that the process can be effectively used to classify and recognize if a sample is containing malicious functions.

In Chapter 2 we briefly present the state of the art about malware classification and recognition methods that are used in the industry.

In Chapter 3 we present the state of the art approaches of machine learning, specifically about image and graph classification, showing PyTorch a Python library used to build GNN models.

In Chapter 4 we present the project of the prototype we wanted to show, describing in details the pipeline of work from bare executable file to image or graph representation, showing a method we used to address the obfuscation problem. We also describe the tools and methods used to develop two deep learning models to use as testing grounds for gathered data.

In Chapter 5 we discuss the results given in the two approaches of deep learning during the training and validating process they've been put through, also speculating on limitations, possible future improvements.

In Chapter 6 we drive our conclusions on the work, discussing possible implementation fields and future improvements.

Chapter 2

Malware and security

The security of a system or infrastructure can be compromised at various levels and by various factors. Following the classic attack pattern, a malicious agent can launch an attack on a system by first studying the system, obtaining information, and then injecting software, known as malware, with the aim of causing one or more actual damages.

The malware thus represents the main piece of software, the weapon, that hackers have at their disposal to inflict the most significant damage. The classification of malicious software is very wide, containing hundreds of categories and special terminologies: they can check for other devices in the network, they can infect specific programs or webpages, they can communicate with the hacker sending back and forth various information such as passwords, e-mails, cookies and privacy settings, they can block or crypt personal files or key actions and even infect physical sectors of the whole infrastructure such as a motor or a turbine. A lot of malware are made to stay hidden in the system and perform malicious actions without being noticed thus they are programmed to hide their actions and presence inside the system, looking pretty indistinguishable from a normal executable program, some times even deleting parts of themselves and using already existing functions so that even a simple piece of code, not heavier than a bunch of kilobytes has the potential to subvert the normal order of the system.

The dispersion of all of these types of different software makes them very difficult to recognize, block and classify when it is time to protect or investigate a security scenario, in other words making it difficult to discern a benign executable file from a malicious one.

We provide a brief analysis on state of the art ideas and approaches based on articles and sources gathered from [8].

2.1 Attack detection

Detecting a malware inside a system is the first and most important step to actually secure the system but it is not a simple task both due to *theoretical problem* and *practical problem*.

As stated in [19] [17] [15] [3] [58] [36] malware and virus detection are impossible and *NP-complete* meaning that there is no algorithm that can solve the problem in a convenient amount of time. The whole process of detecting malware contains a contradiction and there is no program, algorithm that can *or will* be able to solve the problem without false positives.

Even accepting the logical limitations of automated malware detection, there is

the practical problem of obfuscation methods. Nowadays malware are made so that are difficult to recognize in an automated because of countless defense mechanisms:

A malware can be *encrypted* [6] using *oligomorphic, polymorphic* or *metamorphic* keys [61] [59] [70] [4], they can include *stealth* techniques to prevent the correct analysis of the program, making hidden changes to the system [61] and in the most advanced malware, the malicious code is *packed* using packaging methods where essentially the malicious payload of the system is encrypted and compressed to bypass firewalls and other protections and it is only unpacked when the malware is ready to hit the target, usually too late for preventing any damage [22] [72].

2.2 State of the art approaches

The goal of modern malware analysis is to know better of every single malware, knowing witch one could be a potential threat, how to treat and recover from each one and how to block further damage. The first step to do so is to get a comprehensive knowledge of the malware by following two possible methods: *malware analysis* and *malware feature extraction*.

Each malware comes as an executable, so as a list of machine-level instructions for the CPU. By reversing and investigating these instructions it is possible to understand what the malware is up to, what core processes it wants to attack, if there are some obfuscation methods and what are them and the presence of additional payloads. It is possible, by reading the interaction with the operative system, to understand if the malware is trying to contact the attacker via internet, sometimes even uncovering his position. In other cases it is possible to know if the malware is trying to perform malicious operations on surrounding environment or if it wants to remain hidden inside the system. In other words by reversing and analyzing a malware it is possible to read and understand what are the tasks that the hacker wanted it to do in a complete way. Malware analysis is a long and complicated process that takes into account multiple analysts and automated tools.

The second method is to use data-mining techniques to extract key information from a larger dataset of unknown codes, trying to look only into important details such as system calls, malicious strings and other so called *features*. The goal is to create a dataset of features to relate to when a new malware has to be detect and analyzed following various models and method such as *n-gram model* and *graph model*.

The n-graph model is used in a variety of fields and has been adopted also in malware feature extraction. Basically the n-gram model groups specific behaviour by collecting systems calls, features or API-calls into sequential order by a specified value of n. This method is widely used regardless of known limitations on classification and clustering due to the fact that sequential features are not related to one another. This can create larger and larger analysis spaces so that recognition and classification would be problematic in terms of time and performances for the model.

The graph model is widely used to generate handle and extract features in malware. The idea is that a sequence of instructions or system calls is transformed into a graph where each node is a state and each link is the call or the function that makes the system go from state A to state B. The scale of the model is proportional to the time the program is running and to the complexity of it making it difficult to handle. It is possible, however, to *trim* the graph and select *sub-graphs* to better handle the complexity.

2.3 State of the art methods

In modern days the most used technique of malware detection is *signature-based*, used in the majority of anti-virus and anti-malware software. This method works well, it is fast and widely spread but is nearly useless against *zero-day* malware because the signature and the knowledge that the anti-malware system has on the sample must come from an older version of the same [57] [7]. To overcome this difficulty the modern state of the art ha evolved trying to use behavioural-based systems, heuristic and model checking detection upon techniques such as deep learning, machine learning and artificial intelligence, cloud computing and IoT based detection. Using these models it is possible, sometimes, to detect even zero-day malware, predict certain patterns or in general handle the security of a system in a more intelligent way. There is no *best method* to approach the problem as researchers are trying to find the best way to accumulate the pros and reduce the cons of every method, combining some of them and researching innovations.

On the following there will be briefly explained some of the most modern, famous and used state of the art technique on which the present thesis is based on.

2.3.1 Signature-based detection

Signature-based malware detection is the first and probably most straightforward method to perform malware detection. The idea is to use the result of feature extraction as a vocabulary to forge a *signature* of the sample [61] so that just by using that it is possible to relate to the main features of the malware. By doing this it is easy to build a dataset of known malicious signatures to use as a check every time an unknown program is presented: if the features or the pattern of the new sample are similar or identical to the ones that are known to be typical of malware, the software if detected as malicious.

The approach is fast ad precise but can easily be avoided by using simple obfuscation techniques for example a malware could encrypt some key function names or even portions of code. Different techniques are currently studied, summarized following:

Some authors suggest to use a *nature-based* approach using *bioinformatics* and genetic algorithms.

In [82] the authors propose a framework that combines signature-based detection with genetic algorithm [68]. The claim is that this framework is capable of detection zero-day malware but the results are poor on these terms.

Tang, Xiao, and Xicheng Lu [63] propose a bioinformatics approach to extract and detect features of polymorphic worms [60]. The process involves a step entirely dedicated to noise-canceling to reduce at minimum the amount of false results. The system, despite being noise tolerant and very precise, is strictly specific to polymorphic worms an can't be extended on other classes of malware.

Signature-based detection has been used for many years by antivirus software to detect malware that are already known, it works very fast and precisely but falls under modern day obfuscation methods. It is possible to say that in order to be more reliable against modern day malware, a good signature-based detection framework should be fast and precise enough to detect malware using short signatures, should collect data while forging the signature and should not fall prone to the obfuscation, packing and encryption mechanisms.

2.3.2 Behavioural-based detection

Behavioural-based malware detection observes the malware, monitoring the system and all the components trying to understand the behaviour of the software, deciding in the end if it is malicious or not. Even if the program code changes, is packed, encrypted or obfuscated, the behaviour of the same malware should follow the same rules and so be detected as malicious. The main contribution of this method is the fact that it can detect even unknown and zero-day malware since the behaviour of a general malicious software can be detect no matter the version [28].

Behaviour analysis is obtained by looking at system calls [67] [23], file system changes [52], registry changes [28], network activities [5] and active processes [52].

After the extraction of the behaviour a dataset is collected and technologies based on machine-learning are used to classify each new sample. The difficulty of handling such diverse and complicated technologies with such a high complexity input, made of billions of entries and details in each behaviour has prevented so far the creation of a trusted platform model that works perfectly in every situation.

2.3.3 Deep learning-based detection

Deep learning-based malware detection mechanisms are based on a branch of machine learning (ML) and artificial neural networks (ANN) that are the deep learning neural networks. These ML models can learn from examples and build a network of connections understanding the differences between inputs of various kind such as image classification [42] and voice recognition [10]. The idea is to produce a set of features that can be used by a learning algorithm to classify and detect malware in an intelligent way.

A large-scale malware classifier is presented in [20]. In the suggested framework various large scale neural network systems were trained on dataset of over 2.6 million labeled samples and achieved results of 0.49% for two-class error rate, meaning that it can detect the difference between malicious and not-malicious samples and get it wrong in less than 1 over 200 cases.

Deep learning-based systems like this seem to be the perfect approach to the problem but are prone to error due to perturbation. As stated in [31] while investigating the dangers of adversarial crafting against such deep learning-based classification methods, the misclassification rate can reach up to 80% when adversarial crafting is involved and, following [40] it is possible to fool a deep learning-based detection system by changing less than 1% of the content of the sample, meaning that the malware would basically remain the same and not be recognized by the model and that this kind of adversarial attack can be a real threat to state of the art machine learning-based systems.

Chapter 3

Artificial Intelligence, image and graph classification

It is already been discussed the importance of artificial intelligence used as a tool for malware detection and classification. We have studied different approaches and methods taking into consideration the state of the art looking for the best artificial intelligence methods to use, focusing in particular with two main characteristics: the possibility of high customization of the object to be classified, and the high performances and results. By the fact that image representation is used in critical real world scenario, such as automated cars and medical systems, we choose to focus on it along with graph classification, that is more customizable and flexible, able to represent various details and classify different aspects of complex data.

It follows a brief analysis of the state of the art of these two technologies based on [79] and [81].

3.1 Image classification state of the art

Images are the base of the human experience of the world and the act of recognizing and classifying them is difficult to be emulated by an artificial process. Building a trusted image-recognition artificial intelligence algorithm is one of the most important technical fields for the researchers. With the development of technologies, automated image recognition has become one of the most researched fields, spanning in various fields of application such as surveillance [14], automated driving cars [75] and biometric security applications [39].

3.1.1 Image pre-processing and feature extraction

Before applying image recognition technologies to a set of images, the collection of data must be processed and prepared to the process because the quality of the input affects the results of the classification model. In this phase, called *image pre-processing*, the input is analyzed so that important features and other relevant information are enhanced and useless noise is removed, then the image segmentation process divides it into areas of interests with different properties. At the state of the art the image pre-processing methods are mainly 3: 1)Graying, it consists into converting all different levels of color into scales of grey. Most of the time the classificator does not takes into account the color scales of an image so that additional information can be removed from the equation using mainly averaging methods:

$$Grey(i,j) = \frac{R(i,j) + G(i,j) + B(i,j)}{3}$$

2)The second method is based on geometric and spatial transformations, mainly done by bilinear interpolation [26] to reduce image error information and 3) *image enhancement*, that improves the visual effects of the images. Depending on the application, various techniques can be used to make the initial unclear image more clean and detailed mainly using spatial domain [50] methods and frequency domain methods [55].

At this point the image is segmented into different regions by various algorithms: *Threshold-based segmentation* uses values of grey to set thresholds on some regions, *region-based segmentation* divides the image into areas of interest and *edge-based segmentation* defines edges of objects or areas with high precision following the values of grey on the edges of the object.

The segmentation of the image and the pre-processing of colors serves the process of *feature extraction*, where the descriptor of color, grey and texture distribution are extracted and registered so that the main details of the image can be represented as data to be later analyzed.

When extracting the *image texture features* for example, the description of color distribution in certain area of an image is extracted and represented using matrix format [32] counting the the distribution of grey-scale pixels in various areas of the picture. Other state of the art methods are *Gaussian Markov random field model* [18], that has a recognition rate of 100% for 9 textures, wavelet sub-band correlation statistical feature method [56], and various combinations of other representation such as orientation, contrast, roughness, linearity, regularity and coarseness [62].

Local features are instead collected by processing a certain area of the image that has been segmented into sectors of interest. The most used methods in this process are focused around the concept of corner feature extraction and corner points. Various studies [48][33] refers to *corner points* as pixels that viewed from different angles have a different brightness from the surrounding, resulting into key features, unique for the image.

The latest contributions comes from SIFT [45] that s a scale invariant feature extraction method that improves previous methods in efficiency and stability in case the image gets rotated or transformed.

3.1.2 Image classification algorithm

Having collected and extracted various different features from a collection of images it is time to analyze and perform classification on them. The most used algorithms for classifying these data are K-nearest neighbor algorithm, support vector machines, BP neural network algorithm and convolutional neural network.

K-nearest neighbor (KNN) classification algorithm is one of the most simple and it is used as a classification method in various fields other than image classification. The idea is that if there are K similar samples that classify in a certain class than also the current sample must be classified as the same category. For each feature of the sample that has to be classified, the algorithm calculates the distance between the test and the training features, looks at the K samples with less distance and select the category accordingly [44]. The distance is expressed in both Euclidean (1) or Manhattan (2):

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \qquad (1)$$

$$d(p,q) = \sqrt{\sum_{i=1}^{n} |q_i - p_i|} \qquad (2)$$

Following these basis, a lot of improvements can be applied both to efficiency of classification and effect of classification. In [38] the distance is redefined as the correlation distance between two samples, proposing an improved KNN algorithm based on this correlation distance of values, improving both efficiency and accuracy. In [1] the authors proposed an improvement on KNN based on weight adjustment studying mine water inrush. By assigning different weights on different distances it is possible to change accuracy and reduce the error rate, improving in the meanwhile the objective function of the decision tree, resulting in an overall improvement in the accuracy of the model.

Support vector machine (SVN) is a supervised machine learning algorithm proposed in [64][65] based on linear classification. The algorithm makes use of kernels to extend linear classification into non-linear classification, mapping their inputs into high-dimensional feature spaces and constructing segmentation hyperplane based on risk minimization so that the expected risk of the entire sample meets a certain upper-bound. The ideal case is when there are two classes and the model aim to maximize the distance between the two, making it easy to classify future samples.

The main problem of SVM is having low efficiency and low accuracy under larger samples. The idea of various literatures is to combine the algorithm with other known technologies to improve these two measures resulting into combinations with k-means clustering and random forests [71].

Back-Propagation neural network, at [34], is a multi-layer feed forward neural network used in image classification. Being an artificial neural network it can imitate any kind of non-linear input-output relationship. The architecture is composed of three main layers of neurons that are not interconnected inside the layers, being an input layer, one or more hidden layers and an output layer. The process of learning is iterative, meaning that the network first calculate and activate each layer, then calculate the error and then re-iterate the process updating parameters trying to minimize the error until a fixed parameter or condition is met. The main limitations here are related to the shape of the neuron activation function, the sigmoid, that has fixed mapping range locations, missing the right flexibility ad adaptability of the whole neural network.

An improvement to the gradient propagation problem is proposed in [35] where a layer-by-layer initialization points out that multiple hidden layers feature good learning ability. Spatial relationships and filters are proposed as a tuning solution in [76] that is able to improve performances.

Convolutional neural network algorithm (CNN) is a machine learning algorithm based on the traditional back propagation machine learning, adding a convolutional layer and a pooling layer into the hidden layers, to ensure that temporal and spatial invariance are maintained during the execution by comparing different portions of the samples with a small kernel. Convoliutional and pooling layers can be combined, repeated or switched depending on the implementation. The complete CNN algorithm is described in details at algorithm 1.

The contribute of convolutional layers is a near perfect feature extraction, combined with pooling layer to merge semantically similar layers, it makes the CNN

algorithm the most used machine learning recognition and classification method used in the state of the art in various fields, such as vehicle recognition [24], classification of track patterns [80], and license plate recognition [43].

3.2 Graph classification state of the art

constructing a graph by structuring data in form of nodes and edges is one of the most convenient and easy way to aggregate large quantities of complex information, maintaining logical structure and relations between those data. Because of the great expressive power of graphs, it is possible to see the usage of those kind of structure in various research and analysis fields such as social science, biological, physics research, and many others. Graph analysis focus on predicting and classifying nodes, edges and whole graphs using non-euclidean mathematics. Deep-learning methods, due to convincing performances, are the most successful algorithms used in the state of the art, featuring thousands of models, papers and open researches. Recent advancements in deep neural networks and convolutional neural network result in the rediscovery of GNN (Graph Neural Network). The ability of extracting multi-scale localized spatial features of convolutional networks, composing them in complex representation led to various breakthroughs in various machine learning areas.

In this work, GNN gave a huge contribution and in this section we present the basic pipeline design of a GNN network as presented in [81], following four main steps:

3.2.1 Find graph structure

Graph structure is not always the most straightforward data representation method for every problem. For some problems, such as molecules, physical systems and similar applications the graph structure is easy to image. These are so called *structural scenarios*, typical cases where it is possible to assign nodes and edges based on values and observations, building a model to represent the data. In *non-structural scenarios* the translation from the observed nature of a phenomena, the actual number of a measure or the relationships between different values are not structured to resemble a graph configuration. Thus, this configuration has to be extracted, formalized and built from raw data.

3.2.2 Specify graph type and scale

There are plenty of different scales and types of graphs, it is necessary to address them and chose the most suitable combination for each problem individually. Those are

- Directed/Undirected graphs: the direction of edges *going to* and *from* a node is important for the model. For a directed graph the edge can work only in a single way from A to B usually displaying more information in these limitations.
- Homogeneous/Heterogeneous graphs: edges and nodes have the same types, labels and names if the graph is homogeneous. In heterogeneous graphs each node can have unique attributes or asymmetrical values different from others in the same graph.
- Static/Dynamic graphs: the class of graphs that can vary in time featuring different inputs over time, modifications or removal of specific branches.

Algorithm 1 Convolutional Neural Network Algorithm

Forward propagation step:. The input is an image for a network with L layers and kernel of size K. The output will be a^L.
Edges of the image are padded accordingly to the size P of the input layer to obtain the input tensor a^L.
Initialization of parameters W of all hidden layers.
for $l = 2$ to $l = L - 1$ **do**
  if layer l is a convolutional layer **then**
    The output is: $a^l = \mathrm{Re}LU(z^l) = \mathrm{Re}LU(a^{l-1} * W^l + b^l)$
  else if layer l is a pooling layer **then**
    The output is: $a^l = pool(a^{l-1})$
  else if layer l is a fully connected layer **then**
    The output is: $a^l = \sigma(z^l) = \sigma(W^l a^{l-1} + b^l)$
  end if
end for
For layer L, the final output layer, the output is: $a^L = soft\max(z^L) = soft\max(W^L a^{L-1} + b^L)$
Back propagation step:. The input is a collection of M images for a model with L layers and a kernel of size K. The output will be a^L for each of the hidden layers.
Initialization of W for each hidden layer and output layer.
for Iter = 1 to MAX **do**
  for $i = 1$ to m **do**
    Set input to the tensor x_1.
    for $l = 2$ to $L - 1$ **do**
      if layer l is a fully connected layer **then**
        Apply forward propagation algorithm.
      else if layer l is a convolutional layer **then**
        Apply forward propagation algorithm.
      else if layer l is a pooling layer **then**
        Apply forward propagation algorithm.
      end if
    end for
    For output layer L, apply forward propagation algorithm.
    Calculate $\delta^{i,l}$ for loss function.
    for $l = L - 1$ to 2 **do**
      if layer l is a fully connected layer **then**
        Apply back propagation algorithm.
      else if layer l is a convolutional layer **then**
        Apply back propagation algorithm.
      else if layer l is a pooling layer **then**
        Apply back propagation algorithm.
      end if
    end for
  end for
  for $l = 2$ to L **do**
    if layer l is a fully connected layer **then**
      Update W and b following full connection layer.
    else if layer l is a convolution layer **then**
      Update W and b following convolution layer.
    end if
  end for
  if all the changes in W and b are less than a threshold **then**
    Go to step 3 (end of loop).
  end if
end for
At last, **output** the W matrix and offset vector b of each hidden layer, then output layer.

The key concept is that each of these classification can be combined with one another: a graph can be heterogeneous in nodes but homogeneous in edges, featuring some modifications over time and be undirected only in certain specific cases.

3.2.3 Design loss function

The loss function defines the criteria that has to be used when calculating the distance between expected result and the model's prediction. This function, then, uses the distance to update gradients and parameters performing the learning process. The loss function needs to be defined depending on the task that has to be performed:

- Node-level tasks to be performed on nodes such as node classification, recognition, clustering and regression for each node or node collection.
- Edge-level tasks aimed to classify, match or predict a configuration based on values and directions of edges.
- Graph-level tasks that include graph classification, regression, matching and in general tasks that relates to the whole graph representation.

Under the profile of supervision, graph learning tasks can be divided into:

- Supervised setting that provides labeled data in the training process
- Semi-supervised setting that provides a label only for a few components in the set while training, the model needs to infer the classification of unlabeled data by learning from labeled ones.
- Unsupervised setting that only offers unlabeled data for the learning and definition of patterns.

For each of these supervision and task types there are plenty definable loss functions targeted to perform the most with the given specifications.

3.2.4 Build model using computational modules

The final model will be made of computational modules, autonomous functions that transform the data and try to extract features and predict values for them. There are three types of computational modules:

- Propagation module used to propagate information between nodes and edges to capture features and topological information. Here in these modules convolution and recursion are applied to gather and aggregate information. The main idea is to generalize convolution operators in the graph domain, mainly from *spectral domain* and *spatial domain*.

  Spectral approaches work with a spectral representation of the graph based on signal processing [51], applying convolutional operations in that domain. A graph signal X is first transformed by *Fourier transformation* and then the convolution with a filter is applied as if both were a signal. The transformation is then reversed and the filter updated with gradients and parameters. The goal is to learn how to filter specific features, extracting details from the graph signal.

  Spacial approach define convolutional operations directly in the spatial domain of the graph based on its topology using various methodologies such as *message*

passing neural network (MPNN) [30] that extract general characteristics among classic models first using the message function M_t to aggregate messages from different neighbors and then using the update function U_t to update the hidden state. One of the difficulties of handling spatial approach is the aggregation of different neighbors of each node so by using this message procedure the problem results much easier.

- Sampling module combined with the propagation module for particularly big graphs aiming to reduce and simplify the amount of data about neighbors and nodes coming from the previous level to be handled with, trying to handle the expansion. This can be done by:

  Node sampling that is the most straightforward method based on selecting a subset of nodes from each node's neighborhood. The selection can be done at random or by using stochastic approximations based on control-variates [12], or by simulating random walks to select high visited nodes [73].

  Layer sampling based on retaining a small set of nodes to control the expansion factor. This can be done by directly sampling a fixed receptive field of each layer using importance sampling [13] or by using a trainable sampler that performs conditioned sampling reducing variance [37].

  Subgraph sampling that is the most advanced and fundamentally different approach. Instead of sampling nodes and edges, the idea is to sample multiple subgraphs and limit the neighbor search within these. This can be done by directly clustering subgraphs [16] or by building new ones upon first clustering nodes and edges [77].

- Pooling module is needed to extract more general features after propagation and convolutions on high detailed graphs and nodes. Complicated and large scale data formats always carry rich hierarchical structures that are very important but also very difficult to handle. Pooling modules are:

  Direct pooling modules that can learn graph level representations directly from nodes following various strategies going from simple node pooling applying max/min/mean operators, to *sort pooling*, a method that sort nodes according to structural roles [78].

  Hierarchical pooling that aims to investigate deeper hierarchical correlations inside the graph structure to apply pooling on a more convenient way by using recursively downsampling operation [53], by training assignment matrix [74], coarsening algorithms [25] and more [29][46][41].

Usually, a GNN model is built combining these modules: convolutional operators, recurrent operators sampling modules and skip connection are used to propagate information in each layer, combining the results in a final pooling layer to extract features and results. The stack composed of these nodes forms the core of a graph neural network.

3.2.5 Example of a GNN design framework: PyTorch geometric

PyTorch is a library for python developed to easy write and train GNN for various applications. It uses deep learning strategies in a wide range of different implementations, benchmarks, training and testing datasets and interfaces to create custom versions of these. A single graph is described by an object `Data` that holds the following attributes:

- `data.x` that represent the node features, I.E. the main attributes of each node.
- `data.edge_index` that is the topography of the graph expressed in sparse COO tensor, the coordinate format that PyTorch uses for specifying the coordinates of nodes. Essentially each edge is represented and indexed with unique incremental value in a 2 dimensional tensor.
- `data.edge_attr` that represent the features and attributes of each edge.
- `data.y` that contains the target to train against such as a node-level representations of nodes labels. Essentially a collection of labels that put the graph in a certain class so that the deep learning model can learn to classify it.
- `data.pos` that is the node position matrix, essentially a way to specify where to put each node in the space of coordinates.

On Figure 3.2 we see a graphical representation example of a graph described using a Data object in code 3.1. It is possible to see how edges are directed depending on the specification in the tensor, and how each node has a custom set of attributes x_1 in this example.

```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 1, 1, 2],
                           [1, 0, 2, 1]], dtype=torch.long)

x = torch.tensor([[-1], [0], [1]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index)
>>> Data(edge_index=[2, 4], x=[3, 1])
```

Figure 3.1. PyTorch definition of Data object

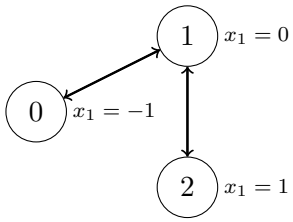

Figure 3.2. Graphical representation of the Data object

Data batching in PyTorch

Neural networks usually best operates by *batching* data, meaning in general that a certain amount of occurrences of data are aggregated in a construct that represent a bigger sample. This helps the training and learning process by the fact that the model has less objects to analyze, but each of them is complex enough so that the informative content of the training set is conserved. Instead of loading the whole dataset into memory at once, batching it into separate collections is useful to lessen the impact on memory and to speed up the learning process. Parameters and gradients are

updated after completing one single batch, meaning that the unbalancing of the whole dataset is reduced because a single error does not compromise the analysis of the whole dataset, but only a portion of it.

For PyTorch this means that the tensor and in general the matrix of each single data object are combined on a single bigger tensor called *batch*. In the code at 3.3 we can see as a single object of type batch loaded using the function `DataLoader()` that has $y = [32]$ meaning that the single batch contains information of 32 different graphs aggregate from the whole dataset. This object has attributes on his own mapping the correspondent attributes on each of those 32 data objects.

```
from torch_geometric.datasets import TUDataset
from torch_geometric.loader import DataLoader

dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES', use_node_attr=True)
loader = DataLoader(dataset, batch_size=32, shuffle=True)

for batch in loader:
    batch
    >>> DataBatch(batch=[1082], edge_index=[2, 4066], x=[1082, 21], y=[32])

    batch.num_graphs
    >>> 32
```

Figure 3.3. PyTorch definition of 32 dimension batch

By using this kind of aggregation, the model works as if each batch was a single dataset, updating gradients and parameters at the end of those 32 samples and, when all the batches have been processed, combining the parameters collected during the process.

Chapter 4

The system: real-time disassembler and image-rendering

Based on previous findings it is easy to see how deep learning and automated classification are state of the art methods with high accuracy and precision able to detect and classify data in a most perfect way. The idea was to use the power of image and graph classifier to build a malware detection tools that is able to recognize a malicious executable file from a non-malicious one, extracting data from the samples, transforming them into *deeplearning-friendly* data (images and graphs), and using a ML model to perform classification. This chapter presents each layer of the tool's pipeline, explaining the idea behind its development. The goal is to show the possibility of developing a tool with these specifications by creating a prototype that could explain and present limited results, useful not for a complete implementation, but to conduct an analysis of the problem and the proposed solution, trying to see if the extra step of transforming the extracted code into images or graphs can help the classification process of executable programs even in the case of obfuscated and encrypted ones.

4.1 Obfuscation and packing

The first challenge is to translate the content of an executable file into images or graph data. Some literature [21] suggested an opcode extractor and disassembler that uses analysis software to translate each instruction of the program into easy to use exadecimal text representation.

There are usually two ways to do that:

- Static analysis, the one used in the literature, essentially works as a reverse compiler of the operational code. The machine instructions generated by a compiler starting from code languages *(C, C++, Java, ...)* are extracted from the executable file that is stored in archive static memory, essentially meaning that it is not run. In this state, with no time constraints, the code is *disassembled* into core functions so that an analyst can study portions of code, understand the workflow and look at variables. There are various tools that perform this kind of static analysis, the most used in the literature is usually *GHydra* [49].

- Dynamic analysis, that is in general more complex than static analysis, is useful in those cases when a malware has to be executed to actually extract and understand the code inside. That is the case of obfuscated and encrypted malware that uses run-time decryption mechanism to hide variables and functions until it is the time of executing them. Time is a constraint in this case because certain malicious programs could be programmed to auto-destroy themselves, or to delete important portion of the operating system. It is important in this approach to take into account the extreme dangerous environment and stress that the system has to endure, that is why this kind of analysis is always performed under *sandboxing* protection.

Being a strong straightforward approach, the idea was to add a level of complexity trying to catch obfuscated malwares. The limitation of the static approach is, in fact, that an obfuscated code, packed or encrypted for example, makes it impossible for the static analyzer to extract features that are useful for the task of representing the real functions of the executable.

When constructing a malware, the cyber-criminal can apply one or more layers of encryption using different algorithms or software. After doing this, the code inside the malware looks like a random chunk of bits, un-usable by any operative system and indistinguishable by any automated recognition tool. On top of the whole program, a special subroutine is programmed to extract the data directly in the memory while running.

When running obfuscated malware the execution pipeline is extracted directly in the memory during the startup of the application using the extractor subroutine, making it impossible to statically analyze it in clear text without running the program itself. The majority of automated anti-malware tools tend to flag these kind of malware as default malicious, without being able to look inside of it extracting important information. This is done because the difference between an obfuscated non-malicious software and an obfuscated malicious one are none, it is impossible to distinguish. The idea, then, was to build a dynamic dump extractor that could run on a remote sandbox system trying to catch the exact moment when the code is not obfuscated anymore, loaded into memory and ready to strike, dumping the opcodes of instructions in a safe place.

The obfuscation of a sample can be measured using an explorer software [69] that calculates the entropy of data inside the file. Figure 4.1 and Figure 4.2 shows two visualization of the portable execution header of the same malware, one is packed with MPRESS [9] and the other is un-packed. It is possible to see how both MD5 and SHA1 are different because of the randomness added inside by the packer. The entropy value is used to measure this randomness and it is easy to see how in the second picture the value is lesser than in the first.

If we use static analysis tools to analyze the malware in the encrypted format, we would loose at least 25% of the code, performing a pointless analysis. These kind of malicious software is usually underestimated and ignored by analysts while developing automated classification tools because of the extra step needed in the dynamic analysis, resulting in an incomplete state of the art in these regards. The problem with this is that the majority of malwares now uses obfuscation techniques and building software without taking this into account could be a mistake.

Later in this chapter we show how the automated unpacking tool will maintain and extract the features of a sample that is packed the same as if it was not.

property	value
md5	E17926067B8CA05F98F8F2D3BE3F33CE
sha1	B940774E6C1C897EB8D43ADFD6716FEC0C4F57D0
imphash	n/a
cpu	32-bit
size	7680 bytes
entropy	6.828
file-version	n/a
file-description	n/a
compiler-stamp	Thu Feb 07 12:55:36 2019
debugger-stamp	n/a
type	executable
subsystem	Console
signature	n/a

Figure 4.1. Portable execution header of packed malware

property	value
md5	E48172D2C15B4C34A891F9C904652951
sha1	A4FB90B948DA3A397062EEFDAD9F12E5498B83E7
imphash	n/a
cpu	32-bit
size	10240 bytes
entropy	5.364
file-version	n/a
file-description	n/a
compiler-stamp	Thu Feb 07 12:55:36 2019
debugger-stamp	n/a
type	executable
subsystem	Console
signature	Microsoft Visual C++ 8

Figure 4.2. Portable execution header of un-packed malware

4.2 Sandboxing in VM VirtualBox

The chosen tool to perform sandboxing is VM VirtualBox, a software that creates virtual systems simulating a working Windows10 ecosystem. Despite the weight and difficulties of handling a system like this, a whole Windows10 simulation turned to be the best option thanks to the compatibility of the most of the malwares in datasets.

By the use of Virtual Box we will create the so called *red zone* inside the sandbox with the scope of handling dangerous software shutting down every security precautions. In the red zone every antivirus software is shut down and access to internet is denied for every application so that the sandbox is isolated and secured.

In opposition to the red zone there is the *green zone*, the host machine. Here every security precaution is turned on and potentially malicious data is handled in a de-activated format to ensure security.

It follows the lists of steps performed by the automated script that handles the sandbox environment:

- A collection of archives containing the sample is prepared inside a directory in the green zone. These samples are in an encrypted format for security reasons because they are travelling inside the host machine and by doing this it in ensured that no malware is launched by error of any kind. One of them at a time is taken and shared into a directory with the virtual environment.

- The system is handled by a script that uses *guest controls*, an extension that allow the host machine to control the guest machine sending instructions and commands. A snapshot of a working version of the machine is restored and launched, ready to take the sample and execute it.

- Inside the sandbox a script called *automated_dump.py* is fired using guest addiction directly from the host machine. The role of this sample is to unzip the executable file now that it is located in the red zone, and so ready to be launched in a controlled environment.

- In the red zone we use *procdump*, that is a command line Windows monitoring tool [47] that is used by admins or developers to capture dump and information in specific moments of the execution of a program of any kind. The tool procdump can be highly customized using a variety of options, in this case we wanted it to launch the executable and capture the whole memory while the executable is loaded in before it terminates. The idea is that in general

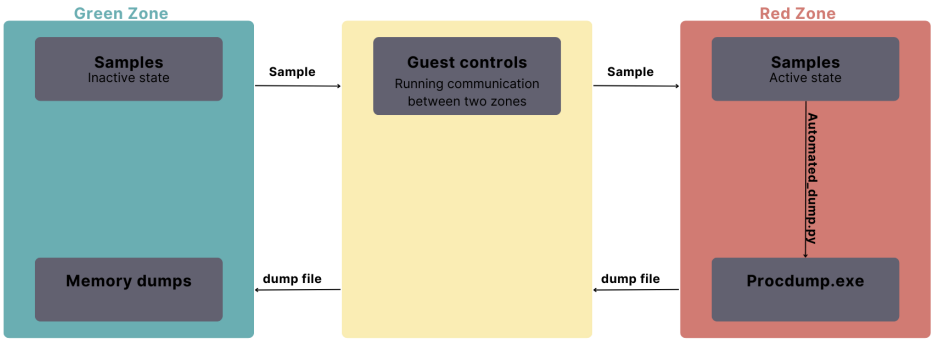

Figure 4.3. Virtualization zones scheme

malware takes few seconds to load the majority of the code in the memory even in the case of long sleeping threats or network attached trojan bots, and that would be enough to gather a valuable dump of the code. The tool is launched using the following options:

  - `"-ma"` to write a full memory dump because the sample could load various section of the memory with payloads and similar.
  - `"-e"` to catch exception with a dump in case the sample decides to exit and auto-delete itself.
  - `"-t"` to write the dump upon termination. This is useful if the dump is still encrypted.

- The virtual simulated system is now compromised having it launched a potentially malicious sample so it is shut down and ready to be restored and repeat the process again.

This process works as a black box taking as input a series of samples in a deactivated state and gives a series of memory dumps as output.

4.3 Opcodes extraction

After obtaining a collection of memory dumps taken after the execution of various executable programs in a controlled virtual environment it is necessary to adopt a translation mechanism to these data.

The important part of these are the *opcodes*, hexadecimal numbers that as a convention are mapped with a description of assembly machine-level execution. As a recall, every program running on a computer is a sequence of elementary mathematical and logical operations applied to data in registers and memory written in a low-level language called assembly. There are 256 opcodes going from 00 to FF representing the 256 possible assembly operations.

As we can see on table 4.1, for each memory location, specified in the first column there is an instruction described by two or more hexadecimal values. In location `013e4a1e` for example it is located the opcode `44` that correspond to `inc esp`, on location `013e4a1f` there is `53` corresponding to `push ebx`, and so on. It is possible to see that sometimes an opcode can be followed by other values that represent eventual attributes of the operation, additional location of memory or data that is being manipulated. For performance reasons these are ignored sticking to the maximum number of possible 256 operators. The idea is to extract the unique series of operators as hexadecimal numbers and combining them into more portable and easy to analyze formats.

Table 4.1. Opcodes from notepad.exe dump

Memory Location	Opcode	Assembly Operator	Additional Parameters
013e4a1a	ba01525344	mov	edx,44535201h
013e4a1c	52	push	edx
013e4a1d	53	push	ebx
013e4a1e	44	inc	esp
013e4a1f	53	push	ebx
013e4a20	b7e6	mov	bh,0E6h
013e4a22	06	push	es
013e4a23	7b18	jnp	notepad!0x22
013e4a24	185208	sbb	byte ptr [edx+8],dl
013e4a26	0801	or	byte ptr [ecx],al

To extract the opcodes from memory dumps a javascript program is run on WinDBG, the internal Windows dump manager, directly in the host machine. The script can analyze the dump of the whole memory and identify the locations in which the sample has been loaded. After doing this, the code is disassembled and presented as a sequence of assembly instructions, and so opcodes. A direct output channel is connected with a text file and the sequence of opcodes is printed on it, filtered depending on various factors such as length and additional details, useful for the dump application such as timestamps are removed so that only the useful sections of the memory that contain the sample data and are registered.

After doing this, the text format is translated into CSV format for an easy representation. The idea is to link the opcodes as couples following the logic that opcode B follows opcode A as operation B follows operation A in the execution sequence of the program, trying to recognize and register common patterns and sequences of operations. At [21] the authors used a method where the translation of opcodes into images happened directly, meaning that a malware using the same operations as a benign program is indistinguishable from it. In general malware can be created by simply *shuffling* the same operations of a benign software, meaning that just counting and translating the opcodes is not enough to fully represent the sample. We thought that the key point was studying *how* opcodes are arranged, in other word how the basic operations are used to form patterns and functions and how those patterns can differ between malicious and non-malicious software. We thought that a sample representation that could represent not only the types and the quantity of opcodes, but also how those operations *flows* and interact with each others would be an improvement to the state of the art solutions that were posed before, helping the classification and recognition process.

In table 4.1, for example, opcode `53` for `push ebx` follows `52` as `push edx`. This couple of opcodes is registered and linked in the two columns of the CSV table. It

happens that a particular sequence of two opcodes could appear more than a single time in the code, as a reference in table 4.1 on location `013e4a1f` and `013e4a1d` the operator `53 push ebx` appears two times. When this happens the extractor counts every occurrence of each sequence registering the information as the *weight* of the sequence. It can also happen that a particular opcode is followed by two or more different opcodes, meaning that the operation that follows a particular opcode is not always the same. As an example in table 4.1 we can see that the opcode `53 push ebx` is followed in location `013e4a1e` by the opcode `44 inc esp` and in location `013e4a20` by the opcode `b7e6 mov bh, 0E6h`. This links the same instruction with two possible following opcodes, building different sequences and representing different steps or processes in the program.

This method start to form an intuitive pattern *mimicking* a graph structure where each node is an opcode, linked with each other by this *B follows A* logic. This patterns is represented by the software that builds a table with 3 columns and $256^2 = 65536$ possible couples of opcodes going from $00- > 00$ to $FF- > FF$ with the third column specifying the weight of that particular sequence. The weight will be then normalized using logarithmic normalization to avoid creating too much difference in the actual values. On tables 4.2 and 4.3 it is possible to see an extraction of 10 sequences from a malware sample and the executable program of the notepad application depicting information as previously specified.

Table 4.2. Opcode sequence extraction of malware sample

From	To	Weight
0	4d	4
4d	5a	1
5a	90	1
90	0	4
0	0	14241
0	ff	11
ff	ff	60
ff	0	14
0	8	5

Table 4.3. Opcode sequence extraction of notepad executable

From	To	Weight
1	4d	1
4d	5a	1
5a	90	2
90	0	12
0	0	13789
0	ff	138
ff	ff	992
ff	0	127
0	f8	13

4.4 Sample representation

With a complete collection of files depicting opcodes linked by couple and weights it is possible to operate the transformation to image. The idea is to build a graph like structure where each node is an opcode and the edges are the sequence B follows A, each with the weight. By doing this it is possible to ensure a fixed dimension of the graph having a maximum of 65536 possible edges and 256 nodes. The graph is then translated into various formats:

- Plotting the graph: The first idea was to plot the graph as an image, using color gradients to represent weights on the edges. It is needed that the image keeps the same aspect when produced using the same sequence of opcodes as input. To ensure that a sample keeps the same representation when processed more than once, each node is assembled in a fixed grid-like structure. There is the possibility of plotting data as graphs simulated with physics that can hold huge dimensional data but the representation is different based on random

seeds of simulated gravitational properties of nodes. It is preferred instead a fixed formation of nodes as a grid of 64 by 64 nodes with 00 in far high-left corner and FF in the bottom-right one. In this way the only thing changing between a sample and another is the configuration of edges in between and so, for the same sample, the configuration is fixed to a 5000x5000 pixel image. Each edge is colored in RGB scale based on the weight of the edge, normalized using log transformation $y = \log(x)$.

The result is shown in Figure 4.4 where it is possible to see a complex net of edges jumping from a node to another, in various colors.

- Adjacency matrix: Following the same principle, for the sake of simplicity, portability and scalability, the graph is translated into an adjacency matrix instead than being plotted, essentially a 256x256 matrix that has a *greyscale* pixel when there is an edge between the nodes specified by the coordinates. The intensity of color in the greyscale depends on the weight of the specified edge. Weights were normalized using log transformation $y = \log(x)$ because of the high difference in dimensions over the various weights and the dimension of the image takes a fixed 5000x5000 pixels.

  The result is shown in picture 4.5 where it is possible to see a square matrix made of grey dots, more or less saturated depicting patterns.

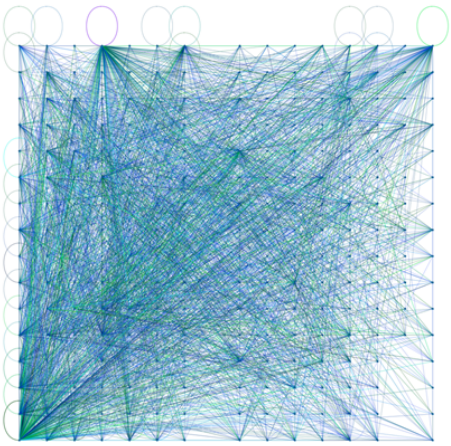

Figure 4.4. Graph representation of sample

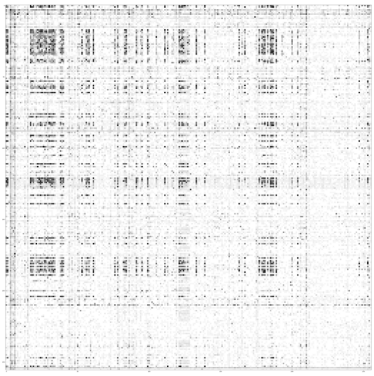

Figure 4.5. Adjacency matrix representation of sample

Thanks to these two image representation it is possible to see how the automatic unpacker is working as intended running a simple experiment. The experiment consists into producing two images, one using a packed version of the executable and another using the simple un-packed one. The process of image rendering and opcodes collection is fired as a standard procedure from top to bottom, resulting into two images representing the memory dump of both executables. By eye-to-eye comparison and also by checking the MD5 of the two different images using the code in 4.6 that gives results shown in table 4.4, it is possible to show how they're both the same pictures, meaning that the process can effectively dump a packed sample without loss of information due to the packing process. This experiment is important to show how the pipeline of script and transformation that takes into account the use of memory dumping mechanism and image rendering functions that we put in

MD5 Hash of encrypted image	e213f580b38fdf1e7b1ea91db3845f87
MD5 Hash of not encrypted image	e213f580b38fdf1e7b1ea91db3845f87

Table 4.4. MD5 comparison

place can in fact ignore the disturbing effects of packing and obfuscation in general: the image of the packed sample is identical to the image of the same sample but not unpacked, meaning that in the process the obfuscation layer is completely removed.

```
import hashlib

def md5_hash_image(image_path):
    with open(image_path, 'rb') as f:
        image_data = f.read()
        md5_hash = hashlib.md5(image_data).hexdigest()
    return md5_hash


image_path = "encrypted_sample.png"
md5_hash = md5_hash_image(image_path)
print("MD5␣Hash␣of␣encrypted␣image:", md5_hash)

image_path = "non_encrypted_sample.png"
md5_hash = md5_hash_image(image_path)
print("MD5␣Hash␣of␣not␣encrypted␣image:", md5_hash)
```

Figure 4.6. Python code for MD5 comparison of two samples

The image representation previously presented was the first conceived way to represent the samples into the process of classification. Given the great performances of deep learning automated image classification the idea was to give the collection of images as input to a pre-trained deep learning model, or even to train one based on those data. The base concept is that if there are any key patterns in the image representation of each sample, an automated deep learning module would at least be able to recognize and classify two classes of samples: malware and not malware.

The main limitation of this format is the time and space that it takes to store and process images with this complexity, talking on an average 50/60 seconds for a single processed sample of about 50 MegaBytes each, meaning that to produce, train and classify an adeguate collection of data it would take a long time and a lot of computational power and digital storage space.

4.4.1 Image classification model

To classify images of samples it is used *Vgg16*, a convolutional deep neural network with 16 convolutional and pooling layes developed from [54]. Vgg16 is one of the most used neural network model for feature extraction and image classification. We accessed Vgg16 using MatLab's Deep Learning Toolbox, an extension that is able to implement and handle the process of construction and training of various deep learning modules.

The network is trained on over 1000 images of objects and it is able to distinct

between various classes. The model is taken and re-trained on a custom dataset of images of samples, both benign and malicious so that it can use weights and parameters already trained to detect images, getting only small adjustments on the target data following a process, called transfer learning, that is a particular method of fine tuning for a learning model.

This particular deep learning model was chosen due to the great performances and reliability, being it deployed in various different areas of interest, but also because of the possibility of performing fine tuning in such a easy way using the previously described developing environment. This model was chosen to show some results of the proposed mechanism on a trusted platform of classification with the idea not to find or create the best model to perform malware classification, but to show the possibility of using this image representation as foothold for future better implementations.

4.5 Graph architecture

Another way of representing the data is to keep the graph structure and using it as it is in a GNN for classification. To do this we must be able to select a language to represent the data in a way that it can be used by classification models. The chosen model is integrated in the library PyTorch Geometric for python. In this library a graph is an object made of tensors, multidimensional mathematical objects that represent the data inside the graph. In this configuration the graph representing the sample has some key attributes:

- Edge index: That is a tensor [2, e] where e is the number of edges. It represent each couple of nodes linked by an edge
- Num nodes: The number of nodes
- Label y: The label of the graph. It is a binary value can be 1 for malware and 0 for non-malware.
- Node attributes x: Is a tensor [n, 1] where n is the number of nodes. It contains a series of attributes for each node, in this case each node has only a single attribute that is its value going from 00 to FF.
- Edge features: Is a tensor [e, 1] where for each of the e edges are linked a series of attributes. In this case each edge has only weight as attribute.

In table 4.5 it is showed a sample graph described using pytorch definition. It is important to notice that during the process of translating data in PyTorch graph format basic padding is performed. To do this, for each sample the software creates a complete graph with all the possible edges only changing the attributes of the weight so that an edge that should not exist has weight 0 and it is not considered. This padding process is done automatically in the translation process from CSV to graph structure so that the classification model can later operate on the data in a single format.

The transformation of data into this graph format is done directly in the image processing script by a function that operates the transaction between networkx and PyTorch after gaining data from textual representation and CSV, as stated before. In this way in a single script it is possible to obtain the image representation and the graph representation saved as two distinct files.

Table 4.5. Data object describing sample graph

Data object	
Data(edge_index=[2, 65536], num_nodes=256, y=1, x=[256, 1], edge_features=[65536, 1])	
Attribute	**Value**
Number of nodes	256
Number of edges	65536
Average node degree	256.00
Isolated nodes	True
Self-loops	True
Undirected	True

4.5.1 Graph classification model

In terms of GNN and graph classification the matter is more difficult because it not always possible to fine-tune a pre-existing model on a different set of samples as in this case. Every GNN model is usually prepared especially for a specific classification task and even similar problems can lead to different model creations. The solution is to build a custom GNN model using pre-existent computational layers and then training it with a specific training set. The problem of training a model starting from scratch is that the dimension of the dataset must be big enough to guarantee a good training process. It is also necessary to study the problem in depth and build a model with a working combination of the right convolutional layers so that data is processed in a way useful to the collection of gradients and the learning process of the model.

As we can see in table 4.6, the model used as testing ground and prototype for the classification of graphs generated from the samples is composed of three convolutional layers of GCNConv and a final linear layers. GCNConv is a message-passing convolutional layer made for GNN and provided by PyTorch at [27] that can classify nodes, edges and even whole graphs, as in this case, considering information on both nodes and edges as the weight. At the end of the network there is a fully connected linear node that performs linear transformation on the output.

Table 4.6. GCN Architecture

Layer	**Description**
conv1	GCNConv(1, 64)
conv2	GCNConv(64, 64)
conv3	GCNConv(64, 64)
lin	Linear(in_features=64, out_features=2, bias=True)

The training and execution loop works basically in tree steps:

- Data batching: different graphs are loaded using the dataloader and are aggregated into batches of 64 graphs, easier to handle. The whole dataset is divided into a portion for training and a portion for testing so that the testing process can be done on samples never seen before by the model.
- Model training: the process of training the model is launched directly using the

function, giving the batches as input to the first layer. The output is confronted to calculate the accuracy during the training and the loss, useful to update gradients, then a new epoch begins, repeating the process of training until a certain accuracy is reached or a certain number of epochs are elapsed. For each epoch, the model is tested against the testing set, a set of samples separated from the training one, useful to gather information about the improving capacities of the model.

- Model testing: once the training process is concluded, the model is tested against the testing set, or other previously prepared set of samples one last time to ensure that gradients and parameters are working fine and that accuracy reached during training is maintained. At this point the model is ready to use with whatever sample of input.

Chapter 5

Results and analysis

After the definition and collection of methods and metric for the extraction of data useful for the classification is time to fire the pipeline of scripts and start collecting images and graphs to build a dataset, testing the collection mechanism and the classification models, trying to understand if it is possible to adopt this kind of classification and recognition procedure on a larger scale.

The following data and description have been gathered on a home computer with 16GB of RAM and Ryzen5 2600x CPU as a showoff for the prototype of the whole system, projected to be developed on larger scale machines, with higher computational power, memory and power consumption.

The idea is to show early results and demonstrate the validity of the paradigm of using deep learning models and automated dynamic dump extraction to represent executable software both malicious and non-malicious and, with certain probability, recognize and classify between the two, even in the case of obfuscated malicious code.

5.1 Dataset description

The requisite for data needed to construct a collection for training and testing the deep learning models is that every software is .exe format, executable in windows environment, available in two formats: the malicious set, and the non-malicious one.

The malicious subset of programs is gathered at random from Malwarebazaar [2], one of the most populated databases of malicious software of any kind. The database is populated by organization and security workers to spread malware awareness and build a categorization encyclopedia of malwares from all around the internet. It is possible to configure an API to filter and download automatically at random a wanted number of malwares with a certain extension or particular tags. For the upcoming test will be downloaded a total of 151 malwares with random tags and characteristics, all having .exe extension.

For the non-malicious set of samples will be used 114 executables from the repository at [11] divided among various types going from Softonic to Softforge and others. The repository is a collection of .NEt files, usually lightweight and collected from various sources essentially created to have a counterexample to malicious software.

5.2 Statistics and analytics

The evaluation of both Vgg16 network and GNN custom model is done mainly using accuracy, precision and recall in various epochs of execution of the loop. The loop consists into a training sequence on a training set portioned by a random mixture of both malicious and non-malicious samples. For each iteration a prediction is tested on a testing set and the loss is used as adjustment. At the end of a training cycle, a testing cycle is performed on a new testing set and it is calculated the accuracy of the prediction. This concludes an epochs, the process loops again presenting every time a different triplet of accuracy, precision and recall.

The accuracy is calculated as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

It is a measure of how often the model correctly predicts the label of the sample. It is the most easy to read and understand, instantly showing off the metrics of the system but it can be misleading in some cases, handling unbalanced classes for example as it treats every class with the same importance, aggregating every positive result.

Precision is a measure of how often the model predicts only the positive classes, in this case how often the model can predict for a malware, it is calculated as follows:

$$Precision = \frac{TP}{TP + FP} \tag{5.2}$$

This metrics works better than accuracy when analyzing models with unbalanced classes that have a very high importance on false positives, like in the case of detecting a malware, but without taking into account false negatives.

Recall is a measure of how often a model predict a positive value over the completeness of positives samples in the dataset, it is calculated as follows:

$$Recall = \frac{TP}{TP + FN} \tag{5.3}$$

Also referred as sensitivity, this measure can be useful as a measure for models where the importance of false negatives is very high, like mis-classification of a malware samples as non-malware.

For all of these metrics:

- TP: it is the number of true positives, so the amount of right guesses of a malware during the testing phase
- TN: it is the number of true negatives, so the amount of right guesses of a non-malware during the testing phase
- FP: it is the number of false positives, so the amount of wrong guesses on a non-malware, where basically the model predicts malware and it is not
- FN: it is the number of false negatives, so the amount of wrong guesses on a malware, where basically the model predicts non-malware and it is not

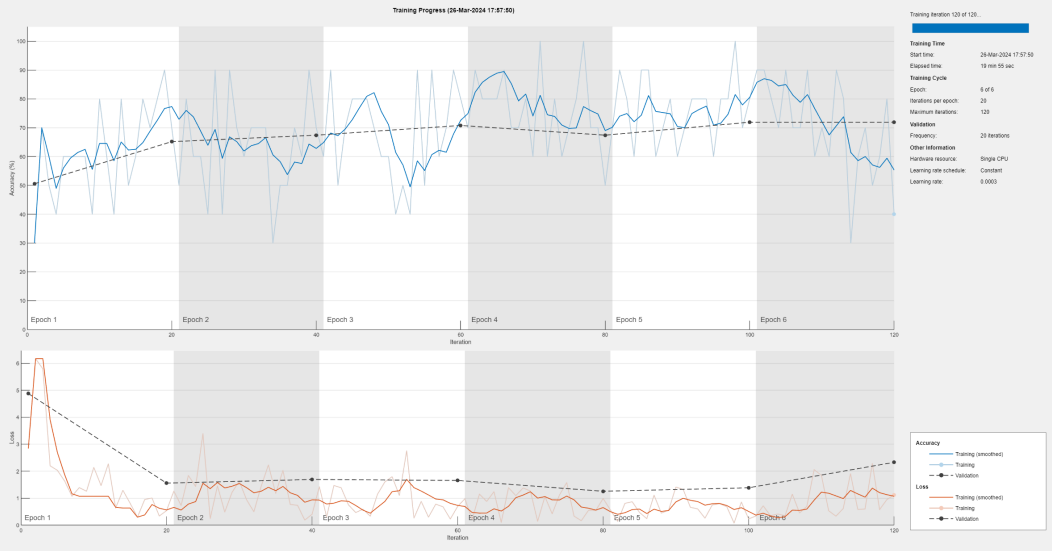

Figure 5.1. VGG16 training results

5.3 Results

Results will be divided into results obtained on VGG16 and image representation and results obtained on GNN and graph representation.

5.3.1 Image representation VGG16 results on image representation

The testing experiment with VGG16 on a collection of 165 samples mixed in malicious and non-malicious greyscale adjacency matrix samples took about 1 and a half hours and pointed out a pretty clear result on the course of various epochs as we can see in Figure 5.1 with a clear increasing of general accuracy up to 70%. The implementation of greyscale representation helped a lot the process of classification, being able to represent the additional information of weight on a fixed dimension scale of adjacency matrix. It is also useful to see how the tuning process of the model only took less than half an hour that, compared to the average time needed to train and validate a deep learning network is much better thanks to the fine-tuning process.

The orange line on Figure 5.1 shows the decreasing of the loss, meaning that on every epochs the model is learning and losing scores as it should, showing that the main limitations of the previous results are exclusively addressed to problems like dataset imbalance given by the limited amount of technological resources that were used.

Another useful tool that VGG16 gives is that it can classify in perchentage samples which are not 100% classifiable, as shown in Figure 5.2. This could still be useful as an helping tool for automated *Intrusion Detection Systems* to flag as possibly malicious some samples in a fast way to ensure security, so that another slower but more precise analysis tool can eventually confirm the result.

The idea is to use this kind of tool as a fast general analysis tool of a sample, trying to classify the main characteristics under a fixed image format and eventually pass the work to a deeper and more precise analysis tool if needed.

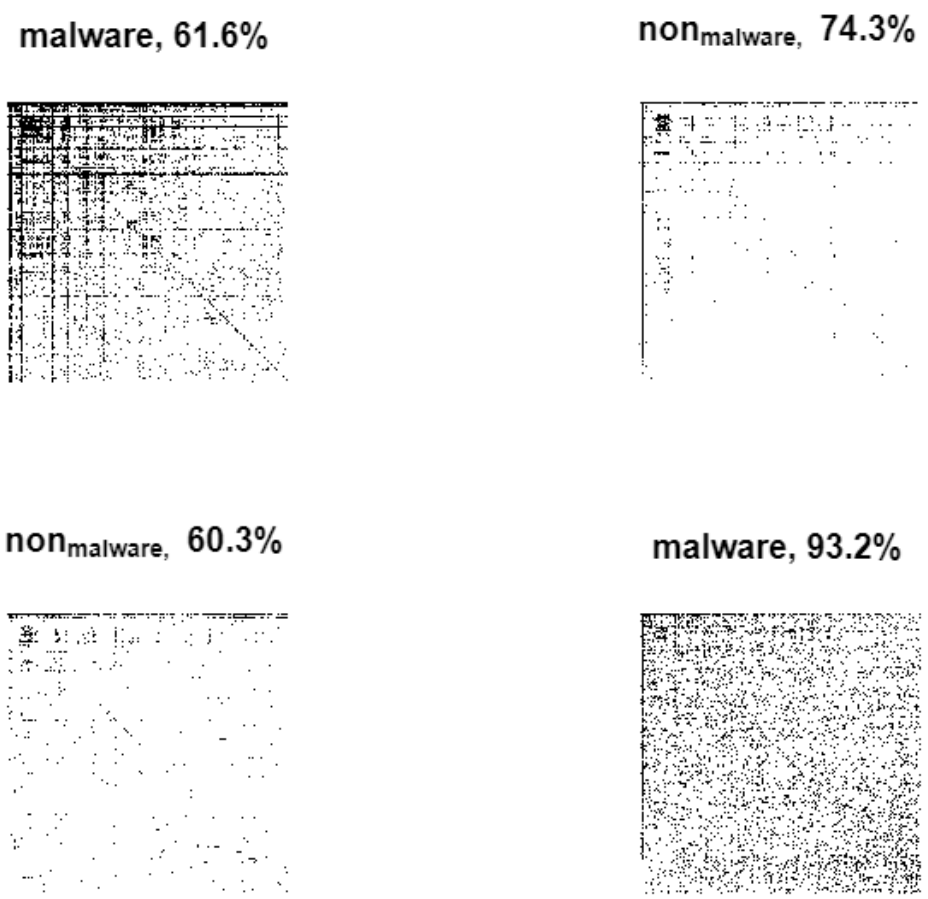

Figure 5.2. Samples classification example

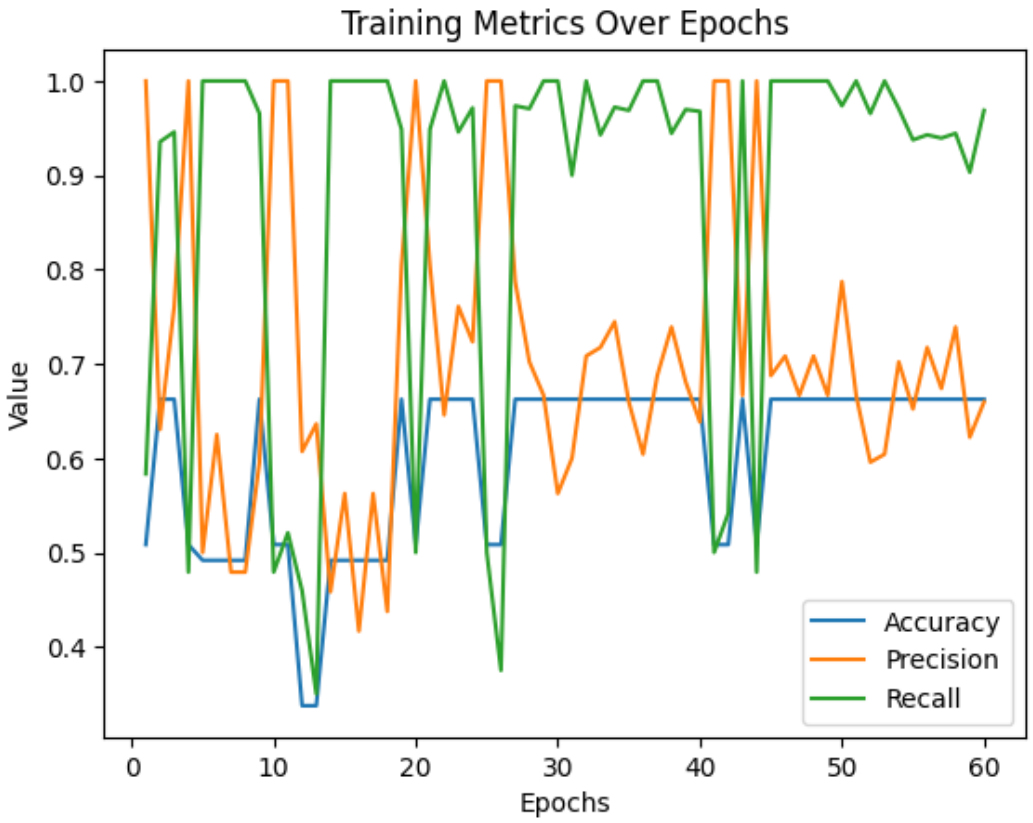

Figure 5.3. GNN results

5.3.2 GNN results on graph representation

The test was conducted on a collection of 300 total graphs, 151 malware and 149 not malicious inside the custom GNN model we prepared for the process and consisted in both training and testing of the model. The whole process took about 2 hours on the home machine used for the implementation of the prototype.

As we can see on Figure 5.3 the accuracy is a little smaller than previously used method, *i.e.* around 68%. Despite this, the real actually interesting result is the fact that, based on results, we can see that the model is functioning, decreasing correctly the loss and in other words improving performances, even if by little adjustments.

Precision and recall tend to stabilize around 97% for recall and 70% for precision meaning that the model is learning how to classify positive samples with a low discard rate. We are positive that those values can increase with a perfected and targeted model, and a bigger dataset.

Training by scratches an handmade model using a limited dataset is probably the main reason of such imperfect results that once again should not be seen as definitive results, but as an indicator of how a system working on these models could actually work, presenting results.

5.3.3 Other solutions results comparison

To evaluate our work we compare it with the study at [66] where the authors presented a general survey on various deep learning detection techniques based on

static and dynamic analysis of each sample, representing the state of the art of malware detection techniques based on deep learning.

The study is presented by using a complex dataset counting more than 240'000 samples, between malicious and non-malicious, taken from various families, and conducted by using various classical machine learning classifiers such as Logistic Regression (LR), Navie Bayes (NB), K-Nearest Neighbor (KNN), Decision Tree (DT), Ada Boost (AB), Random Forest (RF) and Support Vector Machine (SVM), deep neural network (DNN)and Convolutional Neural Network (CNN) and feeding them data about logs and behaviour of the system in general while executing the sample. To conduct a comparison we take results from the dynamic analysis experiment conducted using the models above, being it the most similar to the solution we proposed, and compare it with our results with both representations.

As we can see on Table 5.1 the best results are given by deep neural network, having an overall accuracy of 91%, results that are matched by the other classifiers with minor differences. We can consider these as the state of the art results.

Despite being only able to process a portion of the dataset used in the above article, our solution based on graph classification gives similar results, showing how a different approach of sample representation can be taken into account alongside with classical methods. It is also important to note that the data that was used for these analysis counts several log files and information extracted from various executions and monitoring session for each of the samples. On pure practical view it means that the proposed method is faster because the data that is analyzed is extracted directly from the executable, not from the reports of a system that executed for a long time.

Table 5.1. Comparison experiment

Model	Accuracy (%)	Precision (%)	Recall (%)
LR	67.4	60.06	96.4
NB	54.6	76.3	11.2
KNN	81.5	81.2	81.2
DT	86.0	85.9	85.6
AB	73.3	67.2	89.5
RF	89.5	89.9	88.6
SVM	74.5	70.0	84.4
DNN	91.0	90.06	91.1
CNN	93.6	94.8	92.0
Greyscale Image Representation	70.0	51.0	37.0
Graph representation GNN	68.8	96.7	69.4

At [21] the authors proposed a similar method of malware classification between various families using a similar method of greyscale image representation. After the dumping process, the memory of the sample is represented as a column of vectors, each composed of greyscale pixels representing opcodes. This work presented a series of limitations, first of them the fact that the bigger the dump, the bigger the image. Taken the same sample our work is able to represent it in the same square matrix connected by a finite and controllable number of edges by the fact that the weight on each of them can be normalized using logarithmic function. As we can see in Figure 5.5 the solution proposed by Dai, Li, Qian and Lu tend to produce an image where the height changes with the size of the dump file, resulting in an image that has bigger dimensions compared to the one on Figure 5.4. With dump files of

4Mb the column of vectors solution produce an image that has variable dimensions, using the example in Figure 5.5 we can consider a resolution of 4096x20480 pixels. For the same sample our solution produced the image in Figure 5.4 with a fixed resolution of 5000x5000 pixels. To analyze and classify the image, both machine learning solution need to perform a resizing and a padding process on the image, resulting in a more important loss of information when the resolution of the image is not constant. This means that, on average, our solution is able to sample and utilize images that are way less complicated to store in memory and more precise when resized during training process, proposing a valid alternative and a solution to loss of information problems posed as limitations in the work at [21].

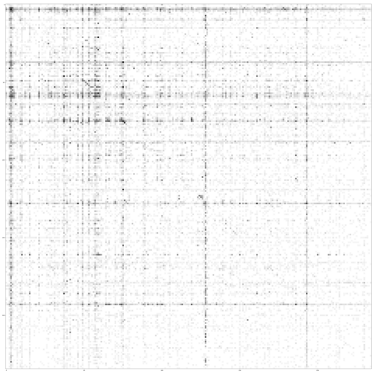

Figure 5.4. Sample represented with adjacency matrix

Figure 5.5. Sample represented with column of vectors

5.4 Limitations and future improvements

As we already said, the aim of this work is to show the possibility of using a different representation of software that is more portable and usable in order to take into the game useful tools such as graph and image automated classification and suggest future improvements and implementation ideas of the proposed system. It is important however to point at some limitations of the proposed work to focus on how to improve them in future works.

The first limitation we wanted to address is about the learning and classification models that were used. The goal of those models was to show how a possible complete pipeline of automated classification should and could work on a real yet simulated ambient of work. The first step to implement and develop the proposed solution should focus mainly on researching and implementing the perfect models to classify and understand the samples in play. It is important to mention, in fact, that despite having positive results, the models proposed in this work were not focused on classifying samples with a competitive accuracy and so some improvements need to be implemented. We should address researching and developing more *ad-hoc* models that could understand and handle the problem as it should, because despite having acceptable results with a pre-trained model, the implementation of specific models made for the specific classification of these samples would perform better. State of the art literature and tools such as PyTorch Geometric can give access to a plethora of highly customizable models to develop and study more suitable solutions. Just to mention, we reached these results using a simple 4 layer deep learning graph neural network but professional and accurate networks can provide deeper levels of complexity to ensure better performances. On the image front we reached these results by fine-tuning an already pre-trained neural network able to classify images of objects and animals, asking them to classify images that can be very different in terms of details and features so, for sure, a complex combination of pre-existing models or the developing of a more specific model able to focus on the extraction of those features that are contained in the images we provided that really contain the informative load could lead to better results.

After doing this, some future improvements could be done on the composition of datasets. For this work we used a pretty mixed set of executable files ranging from installer or lightweight application to malicious trojan, keyloggers and even crypto-ransomware. In other words it is not difficult to imagine that with a better classification model it could be possible to even differentiate between those classes, having more specific features to extract and rely on, while training the model.

A pretty important limitation has to be addressed in the disassembling and dumping mechanism when operating on a specific kind of sample. There are some samples, both malicious and not, that carry multimedia attachments with them to be used during the normal functioning of the program. These could be icons of the executables, particular interface modification, introductory videos or background music for any kind. Some malicious programs could for example modify the desktop layout or wallpaper with a pre-loaded image they carry. These multimedia files are attached and loaded into memory along with the actual code of the sample and so dumped and translated into opcodes along with the code. The so produced dump is unusable or at least obfuscated by the great amount of random bits, translated into random hexadecimal numbers. The presented work does not provide an efficient way of filtering out this kind of content without also filtering useful features of the sample and so, as a contrast to a possible new obfuscation method, it is suggested to research a way of automatic filtering of those multimedia payloads. It could happen in fact that, one day, measures like these could be used as an obfuscation mechanism

to make life of these automated dump even harder. To overcome this it could be possible to develop some more precision into selecting the sections of memory to be actually dumped, keeping the process the most automated possible.

However it is also important to address that for all of the classes of samples that utilize the feature of carrying payloads to actually carry more code and additional features or subroutines such as payload-injecting malware, the possibility of dumping also the content of the payload could turn useful for an analyst. The solution to the problem of payloads should also take this into account, deciding time to time if the payload is worth keeping for further analysis.

Chapter 6

Conclusions

After developing an entire software pipeline to perform memory dump, image and graph translation and classification of various different samples of executable programs we can summarize every reached target of the present work.

As first, we successfully produced a prototype of various automated software to handle the serialization of the dumping process, translating data into more portable formats that are compatible with automated deep learning analysis. The software itself is made to aggregate different scripting processes inside and outside sandboxed safe zones, using virtualization and takes into account different languages and data formats. On an average, it is able to correctly process one sample every 60 seconds on a home machine.

The serialization process of dumping memory data related to programs is done with special regards to obfuscation techniques taken by eventual malicious software. These obfuscation methods tends to be ignored when talking about automated intrusion detection or recognition by the majority of the solutions. This ends up creating a information gap between what kind of threats is possible to handle and what is actually circulating in the security domain of the internet at the present time. We showed, using crypto-proof, that the output product of our system is identical to itself even after real world obfuscation techniques are applied to it. This breakthrough, we hope, could be useful for future studies and implementation that takes into account data that can effectively simulate real world scenarios, developing better awareness on obfuscated malwares.

We were able to realize a new model to represent data related to the execution sequence of a program, disassembling instructions into images and graphs that could maintain details and features useful for software classification and recognition. Other than using these formats for deep learning recognition, we hope that this work could prove itself useful for future improvements, implementing 3-dimensional representations or even more complex data structures to use as a fingerprint alongside the staples of the sector, such as crypto-fingerprinting or even basic disassemblers.

Then, we were able to show how even using standard deep learning practices, it is possible to use these two new formats to perform automated classification. The idea was to show that a standard deep learning model is actually able to update gradients and effectively learn how to extract and classify features out of those tow formats. Results were not outstanding in terms of accuracy and precision, but surely important to pose as a validation of the whole process other than a solid starting platform to improve further.

We were able to point at some limitations, surely most of them linked to the limited amount of computational power, but others directly addressed in the application paradigm such as the problem of filtering additional multimedia content,

or general content, that could pose as a noise disturb. The study of a method to filter out certain type of content should be one of the fist to be carried so that it could be possible to produce even more accurate representations of certain samples.

Bibliography

[1] He cy, zhou mr, yan pc. [application of the identification of mine water inrush with lif spectrometry and knn algorithm combined with pca]. guang pu xue yu guang pu fen xi. 2016 jul;36(7):2234-7. chinese. pmid: 30035996.

[2] Abuse.ch. Abuse.ch Bazaar.

[3] Leonard M. Adleman. An abstract theory of computer viruses. In Shafi Goldwasser, editor, *Advances in Cryptology — CRYPTO' 88*, pages 354–374, New York, NY, 1990. Springer New York.

[4] Shahid Alam, R Nigel Horspool, Issa Traore, and Ibrahim Sogukpinar. A framework for metamorphic malware analysis and real-time detection. *computers & security*, 48:212–233, 2015.

[5] Yaser Alosefer. *Analysing web-based malware behaviour through client honeypots*. PhD thesis, Cardiff University, 2012.

[6] KMA Alzarooni. *Malware variant detection*. PhD thesis, UCL (University College London), 2012.

[7] Ömer Aslan and Refik Samet. Investigation of possibilities to detect malware using existing tools. In *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, pages 1277–1284. IEEE, 2017.

[8] Ömer Aslan Aslan and Refik Samet. A comprehensive review on malware detection approaches. *IEEE Access*, 8:6249–6271, 2020.

[9] AutoHotkey. MPress Web. https://www.autohotkey.com/mpress/mpress_web.htm.

[10] Hyan-Soo Bae, Ho-Jin Lee, and Suk-Gyu Lee. Voice recognition based on adaptive mfcc and deep learning. In *2016 IEEE 11th Conference on Industrial Electronics and Applications (ICIEA)*, pages 1542–1546. IEEE, 2016.

[11] Bormaa. Benign-net. https://github.com/bormaa/Benign-NET.git, Jan 4, 2022.

[12] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568*, 2017.

[13] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[14] Zhiyi Cheng, Xiatian Zhu, and Shaogang Gong. Surveillance face recognition challenge. *arXiv preprint arXiv:1804.09691*, 2018.

[15] David M Chess and Steve R White. An undetectable computer virus. In *Proceedings of virus bulletin conference*, volume 5, pages 409–422. Orlando, 2000.

[16] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 257–266, 2019.

[17] F.B. Cohen. A formal definition of computer worms and some related results. *Computers & Security*, 11(7):641–652, 1992.

[18] Fernand S Cohen and David B Cooper. Simple parallel hierarchical and relaxation algorithms for segmenting noncausal markovian random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (2):195–219, 1987.

[19] Fred Cohen. Computer viruses: Theory and experiments. *Computers & Security*, 6(1):22–35, 1987.

[20] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE, 2013.

[21] Yusheng Dai, Hui Li, Yekui Qian, and Xidong Lu. A malware classification method based on memory dump grayscale image. *Digital Investigation*, 27:30–37, 2018.

[22] Mila Dalla Preda. Code obfuscation and malware detection by abstract interpretation. *PhD diss.), http://profs. sci. univr. it/dallapre/MilaDallaPreda_PhD. pdf*, 2007.

[23] Sanjeev Das, Yang Liu, Wei Zhang, and Mahintham Chandramohan. Semantics-based online malware detection: Towards efficient real-time protection against malware. *IEEE transactions on information forensics and security*, 11(2):289–302, 2015.

[24] Liu Deng and Zijie Wang. Deep convolution neural networks for vehicle classification. *Application Research of Computers*, 33(3):930–932, 2016.

[25] Inderjit S Dhillon, Yuqiang Guan, and Brian Kulis. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE transactions on pattern analysis and machine intelligence*, 29(11):1944–1957, 2007.

[26] Shreyas Fadnavis. Image interpolation techniques in digital image processing: an overview. *International Journal of Engineering Research and Applications*, 4(10):70–73, 2014.

[27] Fey, Matthias and Lenssen, Jan E. PyTorch Geometric. https://pytorch-geometric.readthedocs.io, 2022.

[28] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 2014, 2014.

[29] Hongyang Gao and Shuiwang Ji. Graph u-nets. In *international conference on machine learning*, pages 2083–2092. PMLR, 2019.

[30] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[31] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.

[32] Robert M Haralick, Karthikeyan Shanmugam, and Its' Hak Dinstein. Textural features for image classification. *IEEE Transactions on systems, man, and cybernetics*, (6):610–621, 1973.

[33] Chris Harris, Mike Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

[34] ROBERT HECHT-NIELSEN. Iii.3 - theory of the backpropagation neural network**based on "nonindent" by robert hecht-nielsen, which appeared in proceedings of the international joint conference on neural networks 1, 593–611, june 1989. © 1989 ieee. In Harry Wechsler, editor, *Neural Networks for Perception*, pages 65–93. Academic Press, 1992.

[35] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[36] Zhi hong Zuo, Qing xin Zhu, and Ming tian Zhou. On the time complexity of computer viruses. *IEEE Transactions on Information Theory*, 51(8):2962–2966, 2005.

[37] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive sampling towards fast graph representation learning. *Advances in neural information processing systems*, 31, 2018.

[38] XIAO Hui-hui and DUAN Yan-ming. Improved the knn algorithm based on related to the distance of attribute value. *Computer Science*, 40(Z11):157, 2013.

[39] Anil K Jain and Ajay Kumar. Biometric recognition: an overview. *Second generation biometrics: The ethical, legal and social context*, pages 49–79, 2012.

[40] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European signal processing conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[41] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. Self-attention graph pooling. In *International conference on machine learning*, pages 3734–3743. PMLR, 2019.

[42] Shutao Li, Weiwei Song, Leyuan Fang, Yushi Chen, Pedram Ghamisi, and Jon Atli Benediktsson. Deep learning for hyperspectral image classification: An overview. *IEEE Transactions on Geoscience and Remote Sensing*, 57(9):6690–6709, 2019.

[43] Zhecong Lin and Jiangxin Zhang. gmp-lenet (license plate recognition method based on gmp-lenet network). , 45(6A):183–186, 2018.

[44] Ziyang Liu, Zhanbao Gao, and Xulong Li. An improved knn algorithm based on conditional probability distance metric. In *2017 5th International Conference on Machinery, Materials and Computing Technology (ICMMCT 2017)*, pages 1057–1062. Atlantis Press, 2017.

[45] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60:91–110, 2004.

[46] Yao Ma, Suhang Wang, Charu C Aggarwal, and Jiliang Tang. Graph convolutional networks with eigenpooling. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 723–731, 2019.

[47] Microsoft Corporation. Sysinternals Procdump. https://learn.microsoft.com/it-it/sysinternals/downloads/procdump. Accessed: April 3, 2024.

[48] Hans P Moravec. Techniques towards automatic visual obstacle avoidance. 1977.

[49] National Security Agency. Ghidra: Software Reverse Engineering Framework. https://github.com/NationalSecurityAgency/ghidra.

[50] Shanto Rahman, Md Mostafijur Rahman, Khalid Hussain, Shah Mostafa Khaled, and Mohammad Shoyaib. Image enhancement in spatial domain: A comprehensive study. In *2014 17th international conference on computer and information technology (ICCIT)*, pages 368–373. IEEE, 2014.

[51] David I Shuman, Sunil K Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE signal processing magazine*, 30(3):83–98, 2013.

[52] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[53] Martin Simonovsky and Nikos Komodakis. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3693–3702, 2017.

[54] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[55] Gursharn Singh and Anand Mittal. Various image enhancement techniques-a critical review. *International Journal of Innovation and Scientific Research*, 10(2):267–274, 2014.

[56] John R Smith and Shih-Fu Chang. Transform features for texture classification and discrimination in large image databases. In *Proceedings of 1st international conference on image processing*, volume 3, pages 407–411. IEEE, 1994.

[57] Alireza Souri and Rahil Hosseini. A state-of-the-art survey of malware detection approaches using data mining techniques. *Human-centric Computing and Information Sciences*, 8(1):1–22, 2018.

[58] D. Spinellis. Reliable identification of bounded-length viruses is np-complete. *IEEE Transactions on Information Theory*, 49(1):280–284, 2003.

[59] William Stallings. *Computer security principles and practice*. 2015.

[60] Sulieman Mohamed Ali Sulieman and Yahia A. Fadlalla. Detecting zero-day polymorphic worm: A review. In *2018 21st Saudi Computer Society National Computer Conference (NCC)*, pages 1–7, 2018.

[61] Peter Szor. *The art of computer virus research and defense: Art comp virus res defense _p1*. Pearson Education, 2005.

[62] Hideyuki Tamura, Shunji Mori, and Takashi Yamawaki. Textural features corresponding to visual perception. *IEEE Transactions on Systems, man, and cybernetics*, 8(6):460–473, 1978.

[63] Yong Tang, Bin Xiao, and Xicheng Lu. Using a bioinformatics approach to generate accurate exploit-based signatures for polymorphic worms. *computers & security*, 28(8):827–842, 2009.

[64] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

[65] VN Vapnik. Statistical learning theory j wiley new york. 1998.

[66] R. Vinayakumar, Mamoun Alazab, K. P. Soman, Prabaharan Poornachandran, and Sitalakshmi Venkatraman. Robust intelligent malware detection using deep learning. *IEEE Access*, 7:46717–46738, 2019.

[67] Gérard Wagener, Radu State, and Alexandre Dulaunoy. Malware behaviour analysis. *Journal in computer virology*, 4:279–287, 2008.

[68] Wikipedia contributors. Genetic algorithm — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Genetic_algorithm&oldid=1193124228, 2024. [Online; accessed 15-January-2024].

[69] Winitor. Winitor - PE extraction malware analysis tool. https://www.winitor.com/.

[70] Wing Wong and Mark Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2:211–229, 2006.

[71] S Xiang-Yang and C Wen-Qiang. Impoved parallel svm regression algorithm [j]. *Journal of Xi'an University of Science and Technology*, 37(02):299–304, 2017.

[72] Wei Yan, Zheng Zhang, and Nirwan Ansari. Revealing packed malware. *ieee seCurity & PrivaCy*, 6(5):65–69, 2008.

[73] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018.

[74] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems*, 31, 2018.

[75] Keisuke Yoneda, Naoki Suganuma, Ryo Yanase, and Mohammad Aldibaja. Automated driving recognition technologies for adverse weather conditions. *IATSS research*, 43(4):253–262, 2019.

[76] Li Yong-Qiang et al. Face recognition algorithm based on improved bp neural network. *International Journal of Security and its Applications*, 9(5):175–184, 2015.

[77] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

[78] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[79] Suzhi Zhang, Yuhong Wu, and Jun Chang. Survey of image recognition algorithms. In *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, volume 1, pages 542–548, 2020.

[80] Chang-Yan Zheng, Wei Mei, and Gang Wang. Deep convolutional neural networks for the image recognition of "s-maneuver" target. *Fire Control Command Control*, 41(5):66–70, 2016.

[81] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[82] Mohamad Fadli Zolkipli and Aman Jantan. A framework for malware detection using combination technique and signature generation. In *2010 Second International Conference on Computer Research and Development*, pages 196–199. IEEE, 2010.