UNIVERSITA' DEGLI STUDI DI SALERNO

Laurea Magistrale In Ingegneria Informatica

Corso Artificial Intelligence for Cybersecurity

Malware Detection and Robustness Evaluation with Gamma Attacks

Members:

Mattia Marseglia 0622701697
m.marseglia1@studenti.unisa.it

Marco Pastore 0622701851
m.pastore55@studenti.unisa.it

Ferdinando Sica 0622701794
f.sica24@studenti.unisa.it

Camila Spingola 0622701698
c.spingola@studenti.unisa.it

Lecturers:

Vincenzo Carletti

Antonio Greco

A.a. 2022/2023

Sommario

Introduction

Malwares are malicious software that can infect digital systems to cause damages to single user or entire organizations. This are the main threat for digital security in this modern times. Malicious software uses inner **vulnerability** of systems like programming errors or bugs that gives them the possibility to carry out an **attack**.

With the right vulnerability value of an attack, it is possible to realize malware with disastrous consequences like extortion performed by **ransomware**, Denial of Service attacks (**DoS**) using **Worms**, or data theft through **spyware**.

During 2022 there was an increase in cyber-attacks in quantitative and qualitative terms, in fact it is recent the **Clusit**[1] (*L'Associazione italiana per la sicurezza informatica*) report, in which it is shown how the 32% of 2022 attacks has been characterized with a "**critic**" severity and the 47% with a "**high**" severity. Instead, there has been a decrease of "medium" (-13%) and "low" (-17%) attacks. The level of severity is defined by this organization considering the damages in terms of credibility, finance, society and geopolitical repercussions.

The so called "**Cyber war**" presents a disparate arsenal of malware each of which is designed to create different effects on the machine it infects, and it is a major security problem in this digital age. It is precisely in this context that **Malware Detection** is inserted as a defense mechanism and is actually a very hot research topic.

Malware Detection exploits different tools to identify, block, alert, and respond to malware threats. Among "classic" detection techniques there are **Signature-Based** detection which exploits the signatures saved into a database, comparing them with the signature of the analysed software. A variant of this approach is the **Checksumming,** in which the signatures are obtained computing the cyclic redundancy check (**CRC**). The disadvantages of these approaches are related to the complexity in signatures database creation, and in the difficulty that these systems have in identifying unknown malwares in real-time.

In addition to static methodologies there are also **dynamic** ones, an example is **Behavioral** detection, that analyzing software running inside a sandbox, are able to understand if it is a malware or benign. These types of techniques are the most onerous and complex ones both in terms of duration and instruments to be used.

Even if these techniques have been widely used in recent decades and are the basis that animates the main commercially available Antivirus systems, the new frontier of malware detection is governed by modern and experimental approaches based on Artificial Intelligence. It is extremely important to employe machine learning algorithms (MLAs) to conduct an effective malware analysis, because recent malwares use polymorphic, metamorphic, and other evasive techniques to change the malware

[1] https://clusit.it/

behaviors quickly and to generate a large number of new malwares that are variants of existing one. However, such approaches are time consuming as they require extensive feature engineering, feature learning, and feature representation.

In this report some of the methodologies based on **Machine Learning** and **Deep Learning** will be presented together with **Natural Language processing** with the purpose of determining whether a specific binary file is a malware or a goodware.

Datasets

The experiments with different models have been conducted using three different **Dataset**. The first of these, that is the most numerous of around 15.5 GB and composed of 7358 samples, has been used for the training procedure and for a first models' performance evaluation. The second one, less numerous and composed of only malware for a total of 1.5 GB of 1000 files, has been used for the generalization capability evaluation. Finally, a third set of files composed of only benign has been used to balance the two above descripted datasets.

Below there is a description of provided files typology and single Datasets composition.

PE File

The files contained in the different provided datasets are all of type **PE**. The format **Portable Executable** (PE) is associated to executables files for Windows Operating System, so it is a data structure that encapsulates all the necessary information for Window's loader to manage executable code.

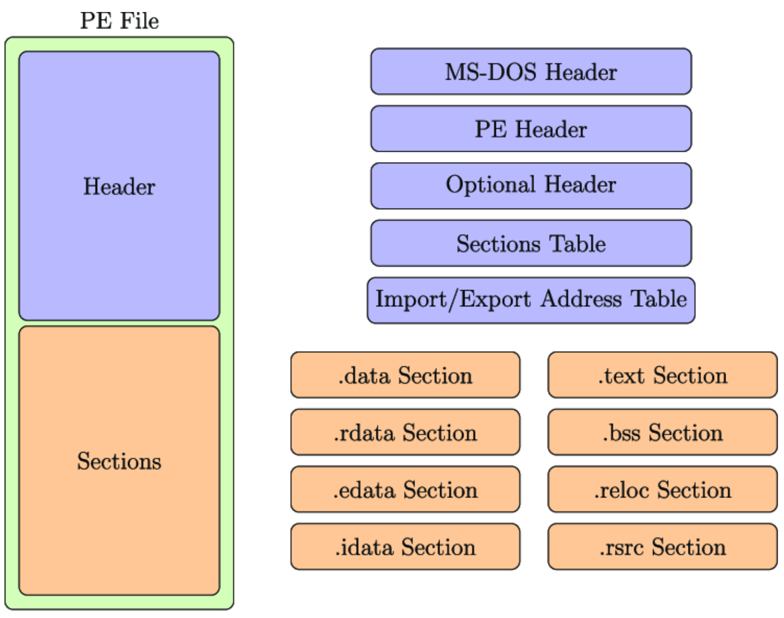


Figure 1- Structure of a PE File

All the PE files are composed of different Headers and Sections where are kept the contents of the file, like the code, the different resources used by the program and the data. This structure is useful to suggest to dynamic linker how to map each section of the file into the memory and how to assign the correct access permissions, starting from the information contained in the headers' files. To avoid wasting space, the sections are not aligned physically on the disk but only virtually. Among the main sections composing a PE files there are:

- **.text** that contains the executable code.
- **.rdata** that lists the Windows API used by the executable, along DLLs
- **.data** that contains initialized data.
- **.bss** that contains uninitialized data
- .**reloc** that contains info on relocation.
- **.rsrc** that contains info like images and other necessary for application's UI.
- **.idata** which contains information about the functions and data imported by the program from the DLLs

Dataset 1: Training Dataset

The first dataset provided was split into a **Training** folder and **Test** folder. In the following report this dataset will be referred to as ***Training Dataset***, for the first part and ***Testing Dataset*** for the second half. Each one presented 11 different classes of malwares and one class of goodwares. The malware classes were the following:

Spyware a malware that monitors and tracks a device and internet activity to gather information without damaging the device as for KeyLoggers or Password stealers.

Ransomware a type of malware that comes with a ransom threatening to publish or blocks access to data usually by encrypting it, until the victim pays.

Dropper a dropper is a program created to install malware, a virus, or open a backdoor on a system.

Downloader it is a type of trojan that install itself to the system and waits until an internet connection becomes available to connect to a remote server or website in order to download additional programs (usually malware) onto the infected computer.

Packed it is a subset of obfuscated programs in which the malicious program is compressed and so cannot be analyzed.

Miner it is type of malware that performs an attack that co-opts the target's computing resources, in order to mine cryptocurrencies like bitcoin.

Flooder it is a trojan that allows an attacker to send massive amount of data to a specific target. Usually, flooders are used in IRC channels to jam communications.

File Infector it is a malware that infects files on the system simply by attaching itself to files.

Installer it is a particular type of malware aimed at installing counterfeit versions of authentic software on a host machine to steal information.

Worm it is a particular type of malware capable of self-replication. Typically, it modifies the computer it infects so that it runs every time the machine is started and remains active until it is turned off.

Adware it is an advertising-supported software, displays advertisements to a user when they are online in order to generate revenue for their author.

The distribution of samples in the Training Dataset is shown in Fig.1. Since the problem faced concerned Malware Detection and not a Classification of samples it was best to remove this distinction among files and **merge all malwares** in a single folder and all goodware in another one. Given the low number of benign files provided in the Training split, merging all the malwares in one class led to an **imbalance** between the two new classes. To solve this problem, benign files were sampled from the third dataset and added to the first to keep uniform the proportions between the two classes.

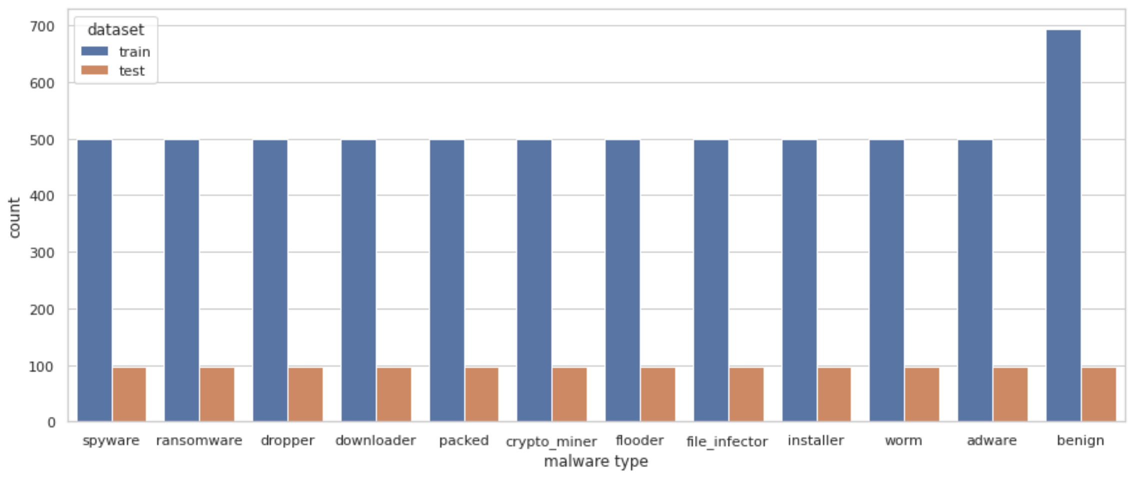


Figure 2 - Distribution of the classes in the Training Folder of Dataset 1

The files to include in the Dataset were not chosen randomly. To keep the average size of the benign sample the same as the rest of the dataset, only certain benign files were sampled. This was done both to **limit the size** of the dataset itself, to increase the speed of preprocessing and to limit the computational cost of the trainings, but also to be compliant with Malconv training specifications.

At the end of the balancing process the Dataset had two main classes: **Malwares** and **Benign** each with 5500 samples. This Dataset was used for training purposes after being split in a **Training Set** and **Validation Set** with an 80-20 split policy that was maintained consistent between different test specifying the appropriate random seed. The final composition can be found in Tab.1.

		Samples	Average (MB)	Total (GB)
Train	***Malware***	5500	1.322	7.101
	Benign	5500	1.334	7.168
Test	***Malware***	1067	1.386	1.479
	Benign	1067	1.445	1.542

Table 1- Average size of Dataset 1 for train and test split.

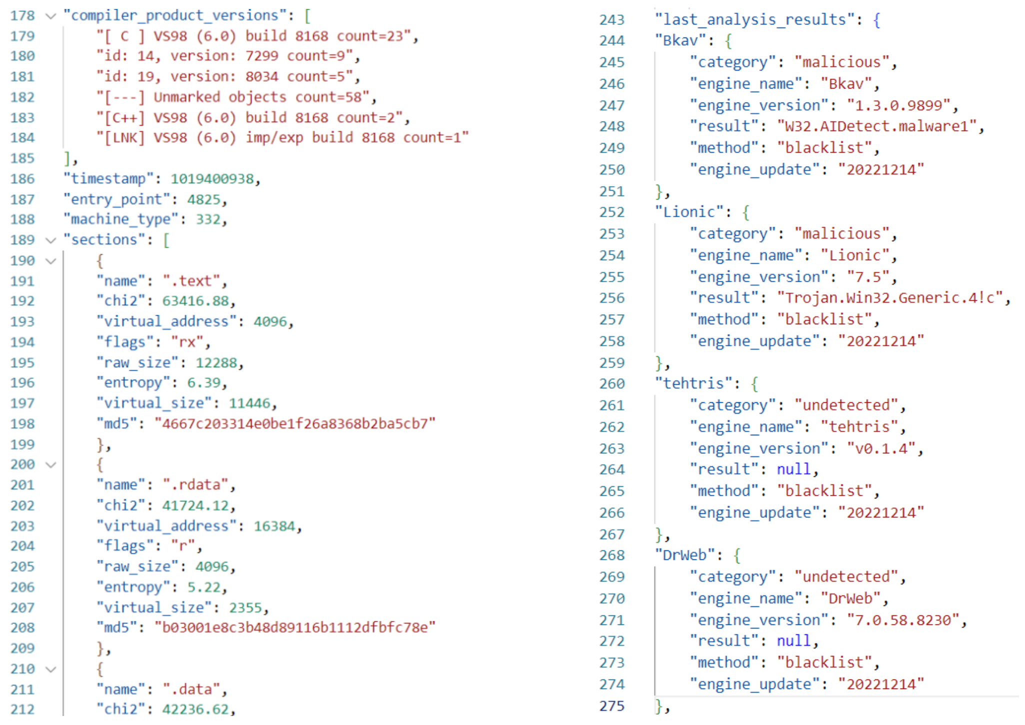
DataSet2: Generalization

The second Dataset was built using only malware files taken from VirusShare[2], a repository of malwares for research purposes and forensic analysis that host a plethora of live malware collections ready to use.

This Dataset was used to test the **generalization capacity** of the trained models. To do so it was balanced with benign sampled from the third dataset in equal number to that of malwares, therefore 1000 benign samples were added. Obviously, the benign files in this dataset were different from the one used to balance the Training set and Test set of the previous chapter. In the rest of the report this dataset will be referred as ***Generalization Dataset***. Because this weren't categorized as the dataset one, for a better understanding of different models' performance and to comparing purposes with Training and Test data, further investigation was carried out.

The analysis was conducted leveraging a free online service called VirusTotal[3]. This is an Antivirus aggregator with over than 70 software, called *contributors*, able to detect and classify malicious software. The service enables a user to scan files programmatically and obtain a full report with the result of the scan for all the contributors. The site uses a REST API structure and enables query up to 650MB of files. The only limitation is the number of query possible in one day, only 500 with a free account.

[2] https://virusshare.com/
[3] https://developers.virustotal.com/reference/overview



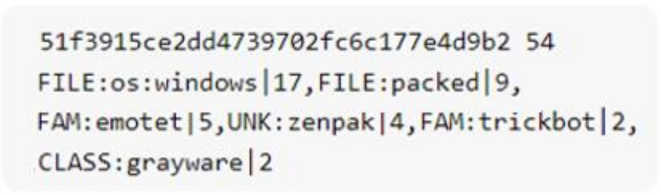
```
"compiler_product_versions": [
    "[ C ] VS98 (6.0) build 8168 count=23",
    "id: 14, version: 7299 count=9",
    "id: 19, version: 8034 count=5",
    "[---] Unmarked objects count=58",
    "[C++] VS98 (6.0) build 8168 count=2",
    "[LNK] VS98 (6.0) imp/exp build 8168 count=1"
],
"timestamp": 1019400938,
"entry_point": 4825,
"machine_type": 332,
"sections": [
    {
    "name": ".text",
    "chi2": 63416.88,
    "virtual_address": 4096,
    "flags": "rx",
    "raw_size": 12288,
    "entropy": 6.39,
    "virtual_size": 11446,
    "md5": "4667c203314e0be1f26a8368b2ba5cb7"
    },
    {
    "name": ".rdata",
    "chi2": 41724.12,
    "virtual_address": 16384,
    "flags": "r",
    "raw_size": 4096,
    "entropy": 5.22,
    "virtual_size": 2355,
    "md5": "b03001e8c3b48d89116b1112dfbfc78e"
    },
    {
    "name": ".data",
    "chi2": 42236.62,
```

```
"last_analysis_results": {
"Bkav": {
    "category": "malicious",
    "engine_name": "Bkav",
    "engine_version": "1.3.0.9899",
    "result": "W32.AIDetect.malware1",
    "method": "blacklist",
    "engine_update": "20221214"
},
"Lionic": {
    "category": "malicious",
    "engine_name": "Lionic",
    "engine_version": "7.5",
    "result": "Trojan.Win32.Generic.4!c",
    "method": "blacklist",
    "engine_update": "20221214"
},
"tehtris": {
    "category": "undetected",
    "engine_name": "tehtris",
    "engine_version": "v0.1.4",
    "result": null,
    "method": "blacklist",
    "engine_update": "20221214"
},
"DrWeb": {
    "category": "undetected",
    "engine_name": "DrWeb",
    "engine_version": "7.0.58.8230",
    "result": null,
    "method": "blacklist",
    "engine_update": "20221214"
},
```

Figure 3 - of JSON file with file information and Antivirus results

Because the files were smaller than the limitation imposed by the site it was possible to scan all the malwares in the *Generalization Dataset*. The obtained results are available in JSON format with different fields giving information on the outcome of the detection operation of the 70 *contributors*, the class or family with the confidence given by the number of antiviruses that contributed to the identification, different kind of information on the nature of the file itself: the format (PE, ELF, Mach-O), the operating system, a section with a behavioural analysis and other useful fields for files categorization. An example of fields in the JSON file is available in Fig.3.

Because in this JSON file a huge number of fields were reported, another tool for data harvesting was used. This is an open-source tool that can extract useful information starting from the VirusTotal tag in the output files regarding the number of antiviruses that detected the file as a malware, the malware the ***Taxonomy*** of the malware with attached confidence. An example of the output from this tool, called AVClass[4], is shown in Fig.4, here the sample of the *Generalization Dataset* is classified as ***Emotet***, a banking Trojan.



```
51f3915ce2dd4739702fc6c177e4d9b2 54
FILE:os:windows|17,FILE:packed|9,
FAM:emotet|5,UNK:zenpak|4,FAM:trickbot|2,
CLASS:grayware|2
```

Figure 4 - Result of the classification pipeline using VirusTotal

[4] https://github.com/malicialab/avclass

Antivirus detection on VirusTotal

The 1000 malware that originally made up Dataset 2 underwent this classification process except for the goodwares that were added later. In a preliminary phase, the **virality rate** was evaluated, where this parameter is given by the percentage of VirusTotal antiviruses that recognized the file as malware. The results of this test are highlighted by the graph in Fig.5, where the majority was recognized by more than 50% of the 70-antivirus considered, for a total of 620 malware.

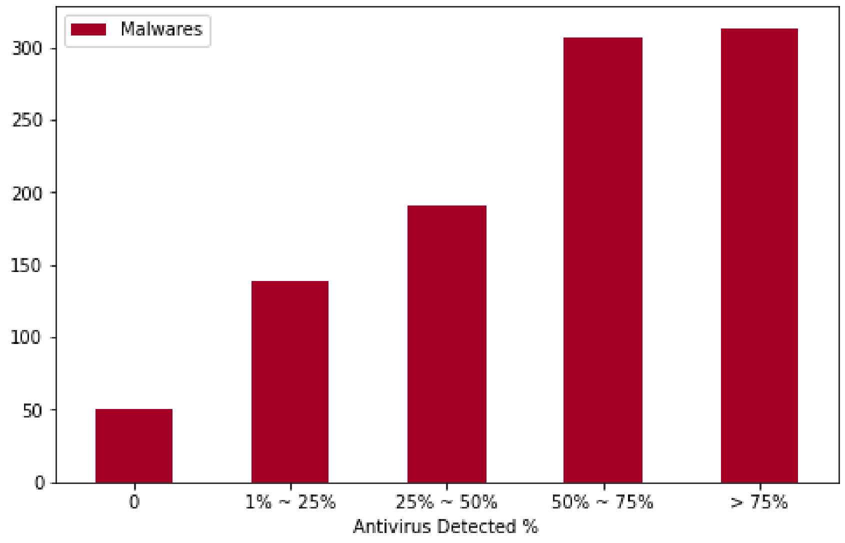


Figure 5 – Virality rate for the antivirus on the generalization dataset

Malware Taxonomy: Class identification

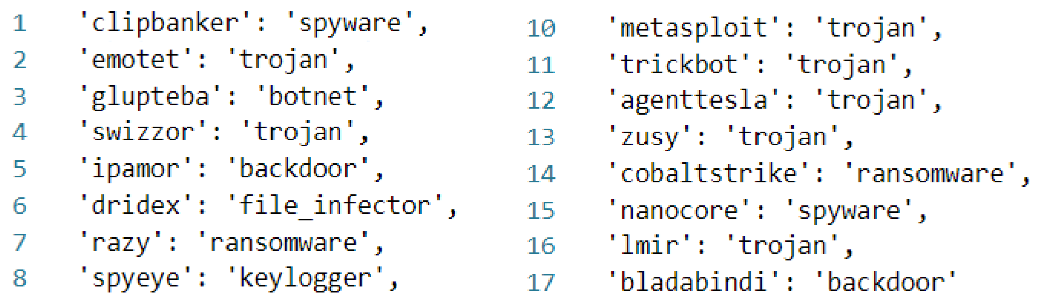
After this initial observation, the study focused on the second part of the output from AVClass, the tag linked with the ***Taxonomy identification*** of the file represented by the belonging malware *class* and *family*.

The information extracted from VirusTotal are not absolute because determining the family of the file from the source code is difficult if not impossible, usually antivirus softwares are able to retrieve this information based on a previous classification of the malware based on its behaviour or other common features like spreading or service utilized. VirusTotal add another complexity given by the fact that malware classification is based on the vote given by the single contributors. For simplicity only the most voted class and/or family were considered reliable.

Of the malware analyzed, 250 of them were not correctly profiled by VirusTotal as they did not possess a majority class. For these we proceeded trying to determine the class based on the family to which they belong by the mapping in Fig.6. This was conducted using different sources such as MalPedia[5] but also trying to respect the classes present within the Training Dataset used to train the various model. If the association between

[5] https://malpedia.caad.fkie.fraunhofer.de/

family and classes was unambiguous or in impossible to derive, we opted for a more generic *Trojan* class.



```
'clipbanker': 'spyware',
'emotet': 'trojan',
'glupteba': 'botnet',
'swizzor': 'trojan',
'ipamor': 'backdoor',
'dridex': 'file_infector',
'razy': 'ransomware',
'spyeye': 'keylogger',
'metasploit': 'trojan',
'trickbot': 'trojan',
'agenttesla': 'trojan',
'zusy': 'trojan',
'cobaltstrike': 'ransomware',
'nanocore': 'spyware',
'lmir': 'trojan',
'bladabindi': 'backdoor'
```

Figure 6 - Mapping from malware families (left) to malware classes (right)

With this process it was possible to reduce the malwares file for which AVClass was not able to determine a class from 250 to 180, where 49 of these as shown in the bar graph above were not recognized as malware at all so they were discarded. For the remaining 131 unfortunately the information extracted through tools is not sufficient for a guess in identification.

The profiling of the ***Generalization Dataset*** highlighted the true nature of the files included. This appear to be hugely unbalanced in terms of both classes present and samples for each class with an over-representation of three classes: ***backdoor***, ***downloader*** e ***grayware*** as shown in Fig.7.

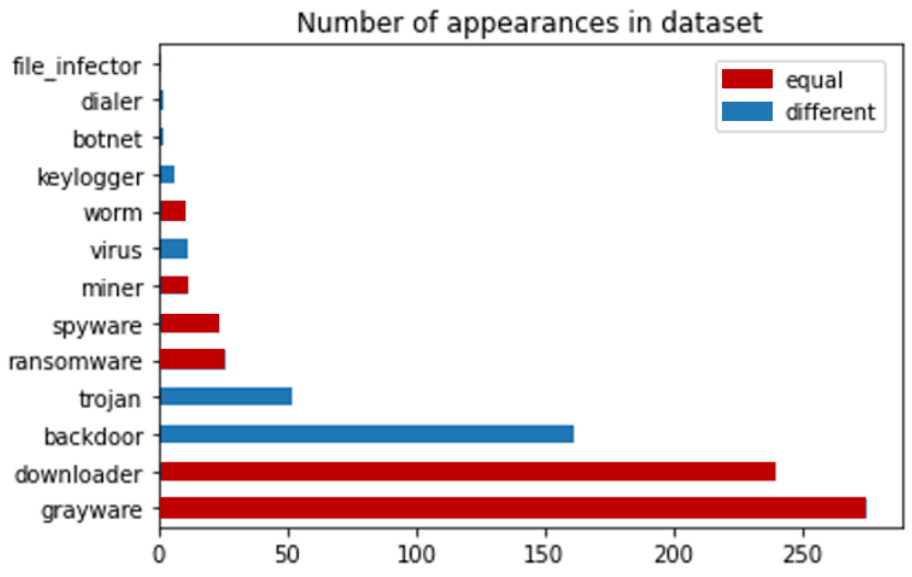


Figure 7 - Distribution of Malware Classes in Dataset 2, with classes similarity between Dataset 1 and Dataset 2

The general term *grayware* refers to executables that, although not being properly classified as malware they can have degrading effects on the system that can range from annoying behavior to hidden or unwanted components. Usually, they belong to one of the following categories: *Spyware*, *Adware*, *Dialer*. Two of these classes are present within the original dataset while the Dialer, a grayware that is installed on a computer and tries to use the dialing features to call other numbers, often running up expensive phone bills for the victim. Since they are partially represented and mostly within the dataset, they were labeled as valid class to consider.

In Fig.7 are highlighted in red the common classes (*spyware, ransomware, downloader, crypto_miner, worm*, grayware) and in blue the different classes that were discarded.

Family Identification

Given the class identification based on the true label already provided in the Training and Test Dataset, another type of ***profiling*** of the malwares was tried, based on the family of the malware. Malware in general present a ***complex Taxonomy***: at the topmost level there are Malware ***Classes***, divided in ***Family*** that, given the macro-behaviour of the class, specify the nature of the malware, the methodologies for spreading and the kind of vulnerabilities exploited. Usually, each family of malwares is divided into ***Versions*** or categories to keep up with the never-ending work of adversaries in malware evolution. Usually, malware families are tightly coupled but two malwares observed in a relatively big span of time can appear to be totally different.

As for family identification, because of the limitation specified above about VirusShare services, it wasn't possible to profile the entirety of the first DataSet to obtain a full knowledge of the malware families in it. So, it was decided to only parse ***Test DataSet*** and retrieve only the type of the file already present in the DataBase.

Of the 1000 malware present in the *Test DataSet* only 37 of them were not available. From the pie chart in Fig.8, it is possible to see that some majority families emerge like ***upatre,*** a well-known downloader that installs malicious software on the host machine.

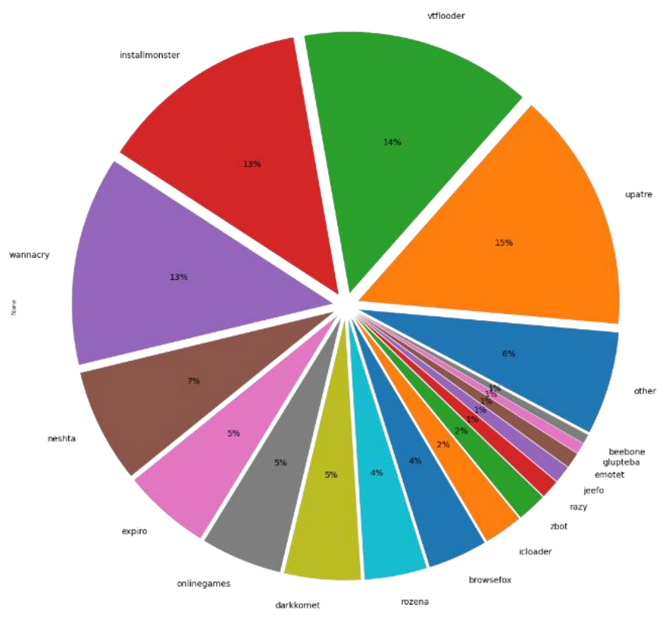


Figure 8 – Distribution of the malware families on Test DataSet

The analysis on the Generalization DataSet produced a different output; in fact, most of the malware appear to be of the same family, ***Emotet***.

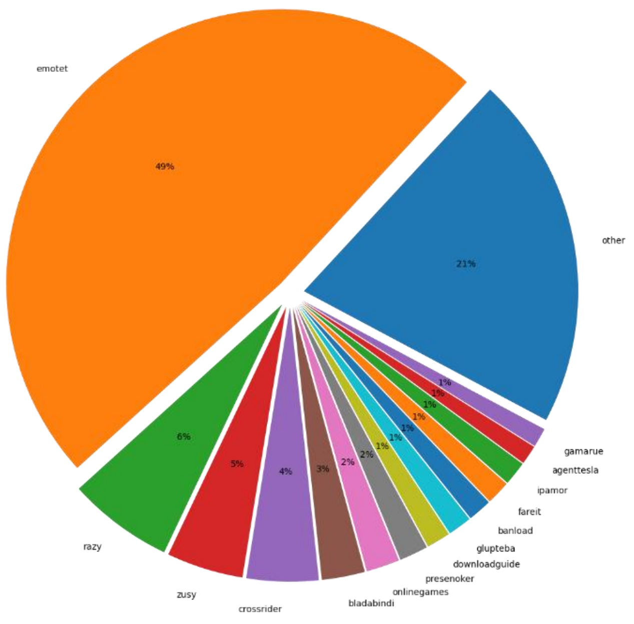


Figure 9 – Distribution of the malware families on Generalization DataSet

The "other" label was used for all the malware family represented by less than 1% of the total. As it appears evident the disproportion in family representation could be one of the justifications of the bad performance on the ***Generatlization Dataset***, as will be seen in the following paragraphs.

Temporal Analysis

The last attempt at analysing the malware provided was conducted using a specific field of the JSON report that is the ***creation_date*** field, which specify the date on which the file was uploaded on VirusShare. Although not 100% accurate, this information is indicative of the date of spread of the malware, that is likely linked with the time of its creation.

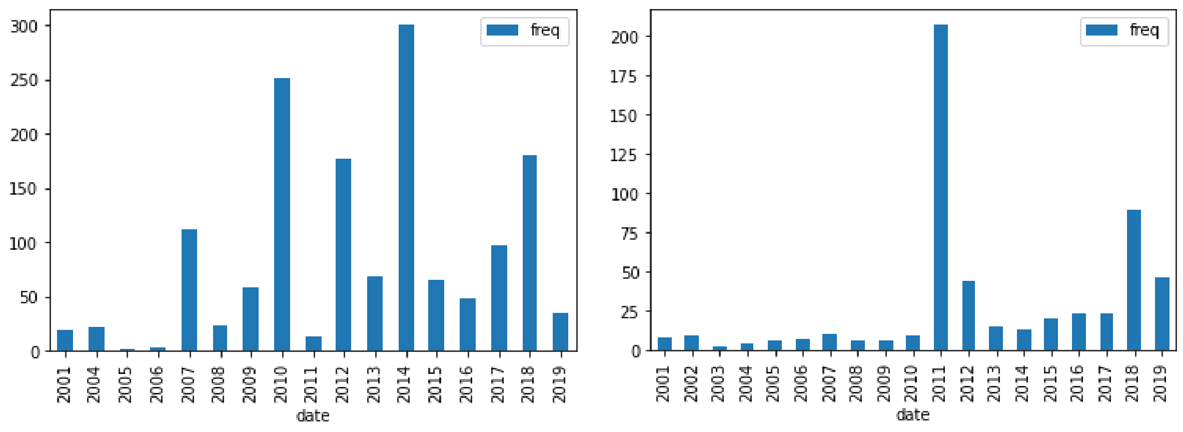


Figure 10 - Right Generalization Dataset, Left Test Dataset.

This analysis was conducted based on the idea that malware of the same class or family can hugely differ in their implementation or vulnerabilities exploited in a relatively low span of time. In the case of the *Emotet* trojan in the year 2022 over 30.000 variants were registered.

The results of the analysis also showed an imbalance in the years of diffusion of this malware. After removing some errors, probably due to poor file management from the VirusTotal platform given by the presence of dates before 1990 and after 2022, the results are shown in the above graphs.

Most Viral DataSet

Based on this analysis, a subset of the *Generalization Dataset* malware was created consisting of samples with a *virality rate* above 60%. This allowed to further skim it to build a dataset with the most viral malware to more effectively evaluate the performance of the models. This sub-dataset now referred to as ***Most Viral*** presents a composition of ***295 malware*** with the following distribution between classes.

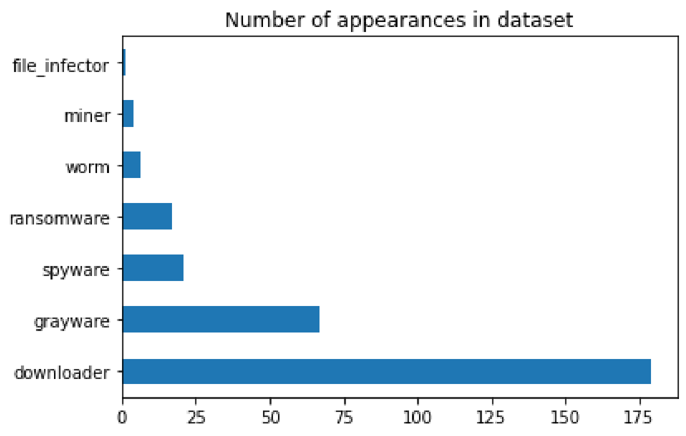


Figure 11 - Malware distribution of Most Viral Dataset

Malware Detection Methodologies

The project constraints were to choose 3 models for Malware Detection, where at least one had to be based on machine learning.

To obtain such a selection we started testing different kinds of models, ranging from Deep Learning image-based CNN to Natural Language processing techniques. The selection was carried out considering the performance of the models on the validation set obtained with an 80-20 split policy from the Dataset 1, where not differently stated. This gave use some comparable metrics to decide, based on the accuracy, which model was eligible for further investigation, and which one was to be discarded.

In the following chapter this analysis is presented with the performance for each of the tested techniques and a brief conclusion on the obtained performance.

Natural Language Processing (NLP)

Natural Language Processing is a branch of artificial intelligence that deals with processing text in order to analyze, interpret, and represent it. The initial applications of these methodologies in the field of security concerned the identification of spam in the context of email exchange via the internet, where the document to be analyzed was evident in the body of the emails and any attachments.

However, applying these techniques to the field of Malware Detection requires the extraction of a corpus, that is a reference text, from such binary files in order to conduct analysis using Natural Language methodologies. Since this is a dataset composed of binary files related to malicious or benign executables, a textual representation can be derived from the executables themselves by analyzing the source code or object code and extracting the opcode and API call name sequences. Due to time and computational constraints, the extraction of *opcode* was not performed. The main difficulties encountered were related to the significant resource demand for their extraction, where for just 1000 files, more than 16 hours of execution were required, and the output size, of about 32GB, was limiting considering that the platform used for the analysis was *Google Colaboratory*[6]. Therefore, we focused on the extraction of API calls.

The NLP algorithms chosen allow for the creation of a ***word embeddings***, that is, they are able to extract semantic and syntactic information from a reference text through a vector numerical representation. Specifically, the technique chosen for vectorization was the ***Doc2Vec*** model. This is an extension of the embedding produced by *Word2Vec*, which leverages not only the logical structures in which words are organized in a document, but also information related to the documents themselves. To take into account

[6] https://colab.research.google.com/

the contribution of the document to the *Word Vector* model, an additional vector, the *paragraph ID* (document-unique), is added.

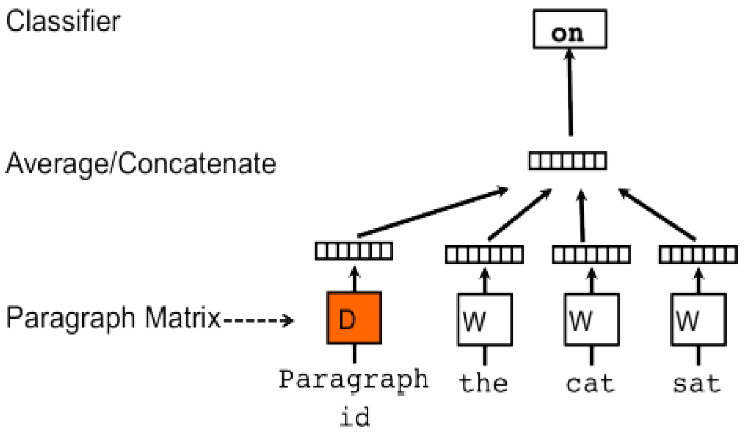


Figure 12 - Basic scheme of the Doc2Vec Embedding system.

Feature Extraction: Api Calls

*Application Programming Interfaces (**API**) calls or requests* are a tool used by programs to interact with other programs, using defined interfaces. In the case of PE files, API calls are the main system with which programs interact with various Dynamic-Link Libraries (DLL), i.e., libraries loaded at runtime. Through these API calls, it is possible to roughly determine the behavior of a given file by analyzing its interactions with the operating system or with commonly used libraries. It is predictable that if malicious software wants to act as spyware, it can leverage system calls that record the user's screen or capture screenshots of the user interface, such as the GetWindowDC API on Windows. The extraction of API calls was carried out from the assembly source files, using a cross-platform library that can parse and modify different executable formats, including PE files called Lief[7].

Through lief we were able to roughly extract from assembly source files the API calls, but further data processing was necessary before feature representation.

Cleaning The cleaning process for the API calls involved removing stop words, which are words that do not add any informational value to the text. However, since there is no consensus on which stop words to use for API call analysis, in this phase we only removed any errors in the extraction of the calls, such as empty strings or insignificant calls such as "**return**". As a result of this cleaning process, approximately 50 samples were removed from both the training and test folders of the *Training Dataset*, due to errors in the API extraction process caused by corrupted headers.

[7] https://lief-project.github.io/about/

Tokenization To be used with the Doc2Vec model, the API calls needed a specific structural organization, which is the construction of **Tagged Documents**, i.e. a set of unique documents composed of an ID and the different tokens given by the API calls. This process is known as tokenization. Each API call was considered as a separate token and then the tagged documents were created by assigning a unique ID to each file and tagging the tokens with the file ID. This allowed the model to learn the unique characteristics of each file and make more accurate predictions.

Dictionary Building involves training the model, whose configurations will be specified in the next paragraph dedicated to training. Once the dictionary was created, it was possible to analyze the most common calls associated with malicious and benign files, also with the aim of validating the cleaning process performed previously.

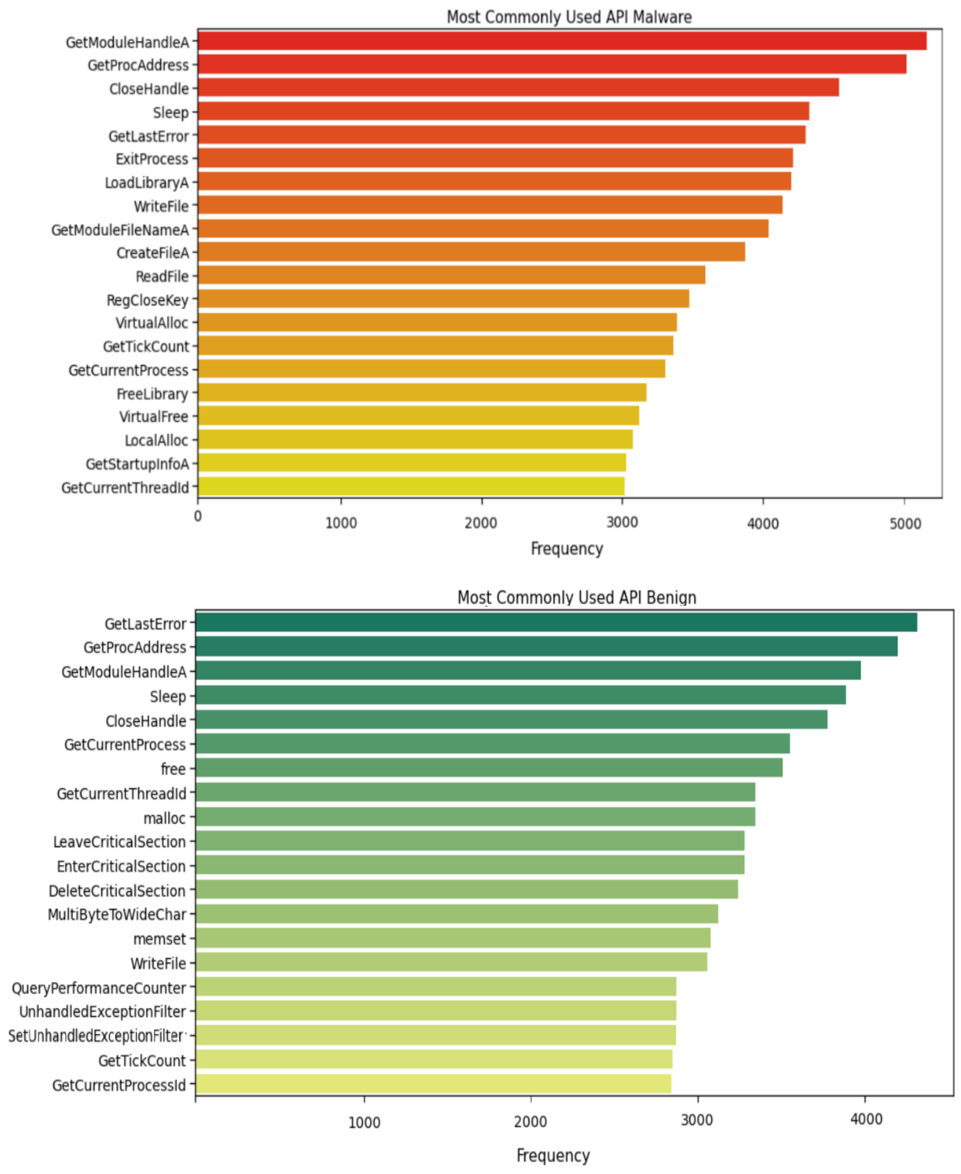


Figure 13 - 20 Most used Api Calls in Malware files (up) and Benign files (bottom)

It is possible to notice an overlap in terms of the most frequent calls, and several of them are associated with standard library commands such as free or malloc, which are widely used in various programs. Although in a dataset filtering process, these could be considered as stop words, it was preferred not to remove them because, in this specific case, the mentioned calls can be used for executing *Buffer Overrun* and *Stack Smashing* in C when properly placed within the code.

Models Building for the training of the Doc2Vec models and the consequent generation of vectors were different, modifying the hyper-parameters ***Vector Size*** ∈ {300, 1100}, which is responsible for the size of the output vectors given the input documents, ***Window Prediction Size*** ∈ {3, 12}, which regulates the maximum distance between the current word and the predicted one, and ***Minimum Frequency Count*** ∈ {2, 11}, which is related to the minimum number of occurrences of the calls in the document. The *TaggedDocuments* generated in the tokenization phase were vectorized in the different configurations.

Analyzed approach

The feature vectors obtained, whose dimension was regulated by Vector Size, were subjected to two classifiers, with the aim of obtaining a preliminary evaluation of their performance. The selection of classifiers was based on ease of use, given by a low number of hyperparameters to configure, and low computational and temporal complexity. These were a ***KNeighborsClassifier*** and a ***LogisticRegression*** model. In Table 2, the 10 configurations that achieved the highest performance in terms of accuracy out of the 60 tested are highlighted.

Min Count	Vector Size	Window Size	KNC Acc.	LR Acc.
8	400	3	0.9415	0.644
8	400	3	0.937	0.646
10	500	3	0.936	0.653
6	300	3	0.935	0.639
4	500	3	0.934	0.633
2	300	3	0.932	0.624
10	300	3	0.932	0.654
8	500	4	0.931	0.645
6	500	3	0.930	0.641
2	400	3	0.927	0.625

Table 2 - Parameters and results on validation Set Dataset 1 of Doc2Vec

The vectors generated by the top 5 Doc2vec models were further analyzed using different models. In particular, it was chosen to test them with a ***Random Forest*** and a ***one-dimensional or 1D Covolution Neural Network***.

Random Forest

The training of the Random Forest was carried out through a Random Search, using as search space the following parameters: ***Criterion*** ∈ ['gini', 'entropy'], ***Number of estimators*** ∈ {10, 1000} ***and Maximum Depth*** ∈ {2, 32}. At the end of the search, the parameters associated with the best accuracy value were the following: [insert parameters].

Parameter	**Value**
Number of Estimators	*365*
Max Depth	*21*
Criterion	*Entropy*

Table 3 - Hyperparameters Selected from Random Search

These parameters were used for the different tests conducted on the generated vectors, producing the following results:

***Doc2Vec Model** [VecS, W, MinC]*	***Accuracy***
D2V [300,3,6]	*92.569*
D2V [400,3,8]	*92.711*
D2V [300,3,8]	*92.522*
D2V [500,3,10]	*93.090*
D2V [500,3,4]	*92.806*

Table 4 - Accuracy on Validation Set of Dataset 1 for Random Forest Classifier

CNN 1D

The second approach tested is a *Convolutional Neural Network* characterized by *Monodimensional Convolutional Layers*, since the output of the vectorization, through *Document Vectorization*, is a vector and not a multidimensional matrix.

A 1-dimensional convolution operation works on the same principles the 2-D version, with the only major difference that are the input and the filter dimensions; it is hence a vector scanning the program top to bottom, looking for features in this sequence. The structure of the simple CNN used is the following:

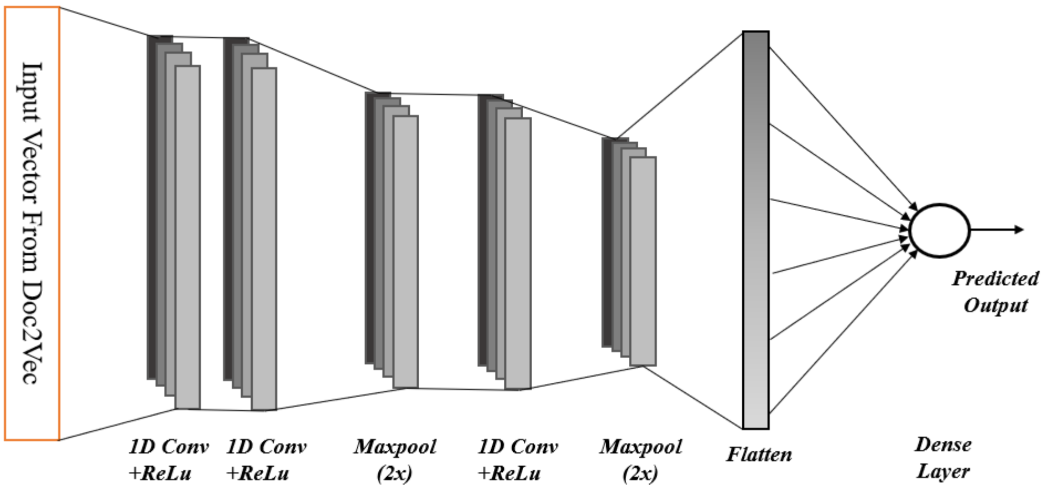


Figure 14 - CNN used for training with 1D Conv Layers

The training was conducted monitoring the loss on the Validation Set for Dataset 1 for at least 50 epochs saving each time the model with the best results. After training the Network on all the Vector generated from the Doc2Vec model the results were the following:

Doc2Vec Models	**Accuracy**
D2V [300,3,6]	*92.143*
D2V [400,3,8]	92.380
D2V [300,3,8]	*93.043*
D2V [500,3,10]	*92.995*
D2V [500,3,4]	*92.333*

Table 5 - Accuracy on Validation Set of Dataset 1 for CNN 1D.

Conclusion

Despite non-negligible performance on the Test set, this type of approach to Malware Detection was discarded. This due to the lack of common guidelines on the handling of API calls in terms of stemming and normalization of the calls and also because of the better performance on validation shown by the other models considered in the following chapters.

Ensemble Classifiers

Among the various approaches attempted to solve the problem in question, it has also been tried the use of simple classifiers by exploiting the feature extracted through Ember, so avoiding the use of more complex neural networks. We were, however, aware of the difficulties associated to malware detection problem, and so animated by the desire to create a solid system even in presence of obfuscation, we designed classifiers ensemble architectures, deeming a single classifier not sufficient. The idea behind the choice of using an ensemble and not a single classifier, is that of giving credit to the goodness of ensemble learning, so that it was possible to obtain better predictions by aggregating the outcome of different predictors, obtaining a strong learner by combining weak ones. So, among all the possible ensembles we tried Random Forests, architectures based on Gradient Boosting approach and an ensemble made from scratch.

Feature Extraction: Ember Features

Given the nature of the datasets provided, composed of binary files, a data pre-processing phase was necessary. The methodology chosen for the conversion of raw data into a clean data collection to feed to the algorithm was to leverage the ***EMBER framework***.

Internally EMBER use the LIEF project to extract features from PE files. Raw features are extracted to JSON files and vectorized features can be produced from these raw features and saved in binary format from which they can be converted to CSV, dataframe, or any other format. To train the networks obviously the second option was chosen, given a PE binary file to EMBER it outputs a vector of features. This feature are the followings:

Byte Histogram A simple counting of how many times each byte occurs.

Byte Entropy Histogram (byteentropy) Sliding window entropy calculation, as stated in *Deep Neural Network Based Malware Detection Using Two-Dimensional Binary Program Features* [1] [1], this roughly approximates the joint probability of byte value and local entropy.

Section Information (Section) Entry section and a list of all sections with name, size, entropy, and other information given.

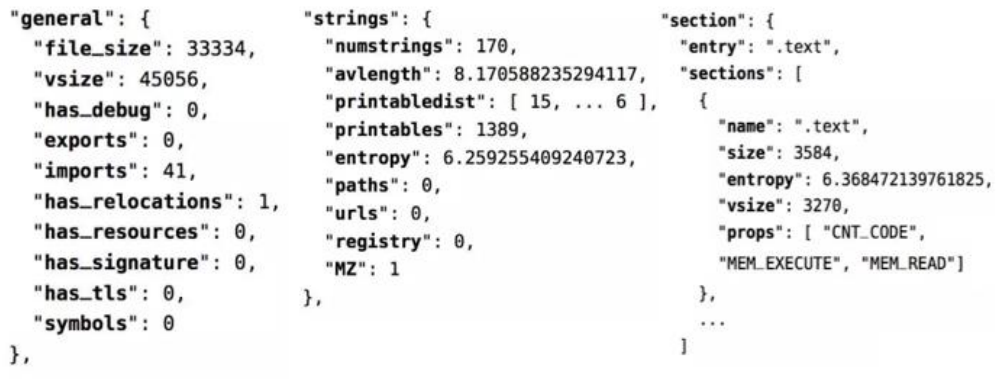
Import Information (imports) Each library imported from along with imported function names.

Export Information (exports) Exported function names.

String Information (strings) Number of strings, average length, character histogram, number of strings that match various patterns like URLs, MZ header, or registry keys.

General Information (general) Number of imports, exports, symbols and whether the file has relocations, resources, or a signature.

Header Information (header) Details about the machine the file was compiled on. Versions of linkers, images, and operating system. Etc.



```
"general": {
  "file_size": 33334,
  "vsize": 45056,
  "has_debug": 0,
  "exports": 0,
  "imports": 41,
  "has_relocations": 1,
  "has_resources": 0,
  "has_signature": 0,
  "has_tls": 0,
  "symbols": 0
},
```

```
"strings": {
  "numstrings": 170,
  "avlength": 8.170588235294117,
  "printabledist": [ 15, ... 6 ],
  "printables": 1389,
  "entropy": 6.259255409240723,
  "paths": 0,
  "urls": 0,
  "registry": 0,
  "MZ": 1
},
```

```
"section": {
 "entry": ".text",
 "sections": [
  {
    "name": ".text",
    "size": 3584,
    "entropy": 6.368472139761825,
    "vsize": 3270,
    "props": [ "CNT_CODE",
    "MEM_EXECUTE", "MEM_READ"]
  },
  ...
 ]
```

Figure 15 - Examples of JSON file portion used by EMBER to extract features.

Given as input a PE file, EMBER then analyzes its structure to produce a vectorization of the information contained in it. It is possible to see from the list of features that more or less all the information that makes up a PE file is exploited for producing an embedding, such as libraries, different sections, file length and entropy. From each PE file EMBER can produce a vector composed of ***2381 features***.

The extraction of the features proceeded without errors for each of the datasets provided, except for 5 files of *Test Dataset,* where the first 512 features had *NaN* value. By analyzing the structure of the EMBER feature extractor, it was possible to identify the type of unextracted feature. This were linked to the ByteHistogram and the ByteEntropyHistogram, due to an error in reading the *bincount* method of python. It was decided to keep the samples by setting these values to 0, which was identified as a default value of the extractor.

The vectors generated by the extractor are ***already normalized*** and therefore ready to be used with the different classifiers.

Analyzed approaches

Random Forest

A Random Forest is a more robust classifier obtained combining many independent decision trees, which hopefully are uncorrelated on the mistakes made. To achieve that goal all individual trees classifiers are trained on a random subset of total features, ensuring independence between classifiers, and at the same time also reducing the required computational cost. Independence can be achieved also by training each tree classifier using only a randomly sampled subset of the training set, also getting the benefit that all classifiers see densely populated regions of the input space, but outliers samples are less likely to be seen by the majority of the classifiers.

So, random forests classifiers have many parameters to be chosen, like the number of base estimators, the maximum depth of three or the function to measure the quality of a split and many others, all that affect performance.

So, because of this reason, we decided to help us, in our not trivial research of the best hyperparameters, using ***Optuna***. It is an automatic hyperparameter optimization framework that with a very high modularity allows to construct the search spaces for the hyperparameters, alternating phases *study* and *trial*. The *study* phase is devoted to the optimization based on an objective function, while the *trial* phase is a single execution of the objective function. So, the goal of a *study* is to find out the optimal set of hyperparameter values through multiple *trials*, and for this purpose Optuna is fundamental because is a framework designed for the automation and the acceleration of the optimization *studies*.

First, we try to conduct a study phase, divided into 100 trials, using **cross validation** approach with a number of fold equal to 5, and during which are searched the best values for the *number of trees* to be inserted in the random forest, for the *max depth* of each tree and also for the *criterion* to measure the quality of a split. Here with best parameters, we refer to that that maximize the accuracy. The choice of use a cross validation approach is that respect to a simple validation-set approach, it allows us to obtain more stable estimation, making the training independent of the specific split of training and validation set, and so helping to find the best hyperparameters' value with a more stable estimation of model's generalization error. In fact, during training procedure, it is important to ensure that the classifier was not overfitting and that maintained good generalization capability. In addition to this reason, it was also taken into account that the dataset was not very large, especially compared with features dimensionality, and so that a simple division in training and validation set to evaluate the performances, probably makes the evaluation inaccurate, risking that the error on the validation set not really represents the generalization error. Because of all these reasons we consider a cross validation approach more adequate. So, the whole training set has been divided into 5 independent subsets and so 5 repetitions have been done in which each time one subset is used as the validation set and the others to train the algorithm. In this way the average of the experiments has been used as a measure of the performance. Also observing the accuracy values obtained on each experiment and comparing this value with the average accuracy it was possible to identify overfitting problems if there was an excessive disparity between the values, but this is not the case.

Below are reported the range values in which the research has been conducted and the obtained results.

Parameter	***Values***
n_estimators *The number of trees in the forest*	*[10-1000]*
criterion *The function to measure the quality of a split*	*['gini', 'entropy']*
max_depth *The maximum depth of the tree*	*[2-32]*

Table 6 - Parameters utilized for Random Forest

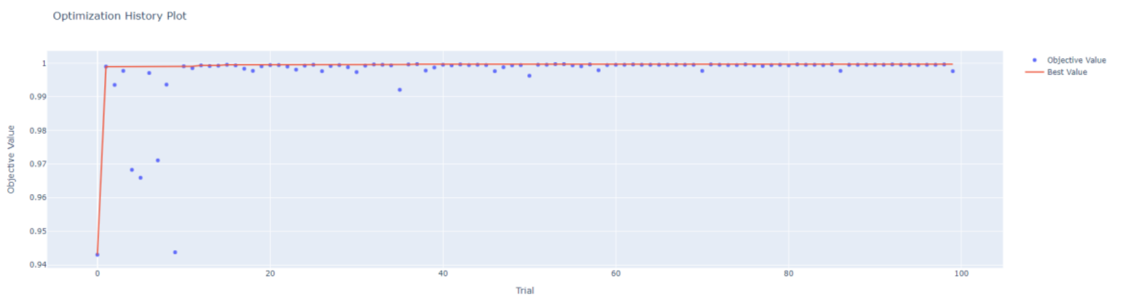
All the other parameters that are not shown in this table are left to their default value, like for example the *max_features* parameter was set to *'sqrt'*.

At the end of this study the best values for the hyperparameters were found in trial 37. Below are reported the obtained value and the accuracy of this random forest on the validation set.

n_estimators	***criterion***	***max_depth***	***accuracy*** *on validation set*
517	*'gini'*	*19*	*99.855*

Table 7 – The parameters and accuracy of best methods for cross-validation.

As it is possible to see in the optimization history plot, from a certain trial onwards there were no more improvements, and this is further confirmed by the obtained results.



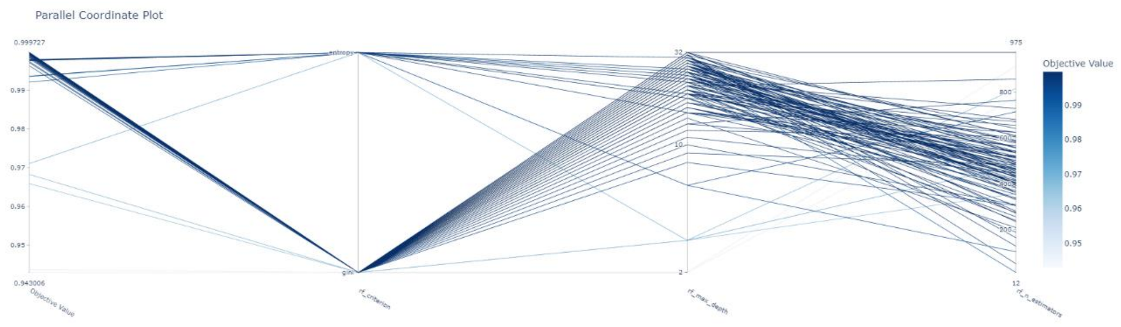


Figure 16 – On the top the Optimization History Plot, on the bottom the Parallel Coordinate Plot.

It is possible to see the evolution of the parameters and the corresponding obtained values of accuracy. It is clear that the highest values of accuracy most of the time correspond to ‘gini’ criterion and to high values of tree’s depth. This is reasonable because higher trees mean deeper classifier that being more complex structures necessarily manage to fit better on a specific problem, reducing the bias of the model. About the number of trees in the ensemble, it can be seen that as this parameter increases, fewer and fewer configurations are found. Also, this is reasonable, because especially with deep classifier, increasing the number of classifiers means having an overspecialized ensemble and so the risk of overfitting becomes increasingly threatening.

To test purposes, and in particular to understand how a cross validation approach can influence the parameters found and the performances obtained, we use **Optuna** with a simple **validation-set approach** but exploiting also the **Pruning** feature in order to save time. This feature automatically stops unpromising trials at the early stages of the training (a.k.a., automated early stopping). Obviously, in order to be able to make a comparison, the study was carried out on the same three parameters and in the same range as which used for cross validation, for which the best results obtained are different, and are presented at the trial 24.

n_estimators	***criterion***	***max_depth***	***accuracy*** *on validation set*
481	*‘gini’*	*27*	*99.891*

Table 8 - The parameters and accuracy of best methods for validation-set.

The pruning feature allowed us to not finish 21 trials that were not promising. Comparing these results with those obtained using a cross validation approach, it is possible to see that there are no notable differences in terms of accuracy performance. Instead, it is clear that the simple validation-set approach has identified as best parameters two higher values for the number of trees to be involved in the ensemble and also deeper. Following are reported both the optimization history plot and also the parallel coordinate plot of this experiment, but only for documentation purposes, because, according to the results observed, the approach with cross validation is to be preferred. This is because in general it is a more stable approach, that consent to better estimate the future test error and also because in this case led us to a less complex model, avoiding introducing excessive variance due to extreme flexible model, and so avoiding overfitting.

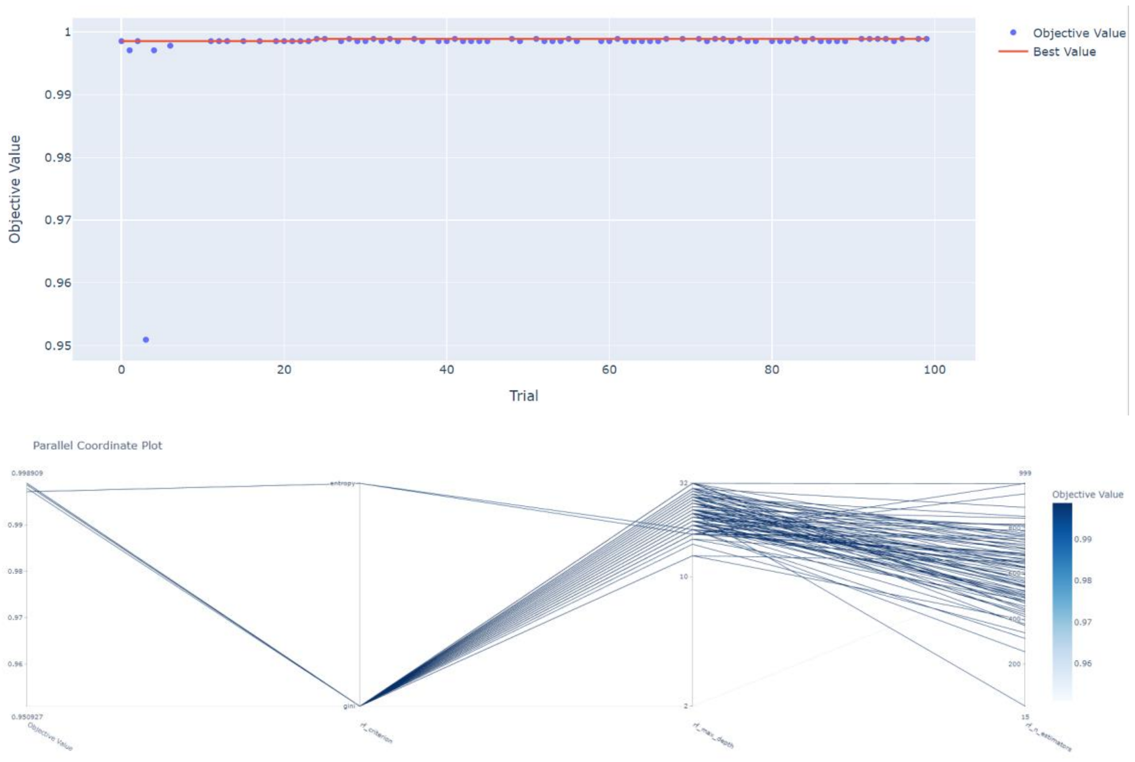


Figure 17 - On the top the Optimization History Plot, on the bottom the Parallel Coordinate Plot.

The last test we have done is that of try to apply a **dimensionality reduction** on the extracted feature because we notice that they are very much for each sample, respect to the total number of samples, and we thought this could make the learning more difficult. So, we applied PCA on the extracted features. In order to make comparison, also in this case we used Optuna with a simple validation-set approach, and the parameters involved in the studies are the same and in the same range of that of the two previous experiments. The only difference is that in this case another parameter on which to perform the study has been added, that is the number of components to consider on the new feature space. The analyzed values for this parameter are that from 600 to 300, every 100, so that the best number of components to better fit the problem were automatically computed. Also in this case, in order to save time, the feature of pruning was used, avoiding completing 51 useless trials. As we expected the obtained results are very different from those of the previous experiments, and the best set of parameters have been obtained in trial 45.

n_estimators	***criterion***	***max_depth***	***pca_components***	***accuracy*** *on validation set*
205	*'gini'*	*18*	*300*	*96.626*

Table 9 - The parameters and accuracy of best methods for dimensionality reduction.

Contrary to what we expected, dimensionality reduction does not improve performance. Probably it is because in this way we are forcing the classifiers to operate in a different space, in which the features are a combination of the first one, preventing him from selecting the features that he deems actually significant and important for the purposes of classification. Specifically, we transformed the optimization problem as a constrained one, adding an additional hyperparameter that limits its performance. Following are reported both the optimization history plot and the parallel coordinate plot of this experiment, but only for documentation purposes, because, according to the results observed, this cannot be considered a valid approach.

In the image above it is possible to see that there are not many configurations that involve a number of PCA components greater than 300.

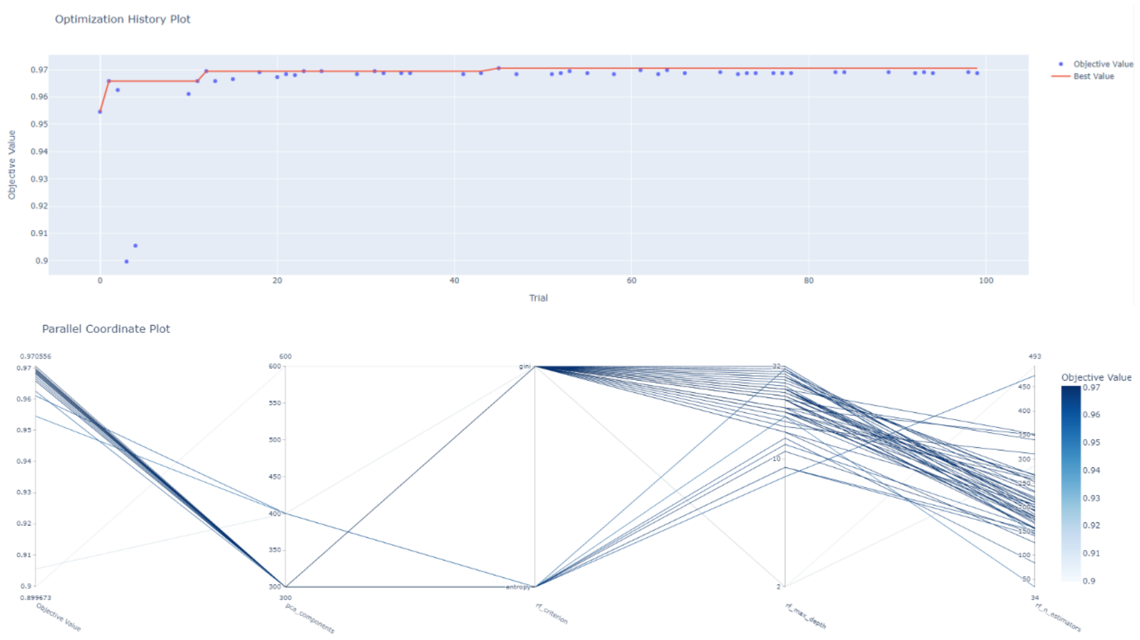


Figure 18 - On the top the Optimization History Plot, on the bottom the Parallel Coordinate Plot.

So, among all the possible tests done using Optuna and Random forest we can conclude that without doubts, the approach with cross validation is the best in terms of accuracy, stability of results and guarantees regarding the risk of overfitting, because did not design an excessive flexible model.

Gradient Boosting

Boosting algorithms combine different predictors, but unlike forests in which each tree is independent and so fits to a parallel construction, boosted tree ensemble is inherently sequential because each subsequent model attempts to fix the errors of its predecessor. Firstly, we make a comparison between different types of boosting algorithms: AdaBoost, XGBoost and LightGBM. We prefer one of these last two that are gradient boosting algorithms that, despite of AdaBoost do not penalize missed-classified cases but use loss function instead, and also use the gradient descent method to continuously minimize loss function to find the optimal point. Because of these reasons, even if are more prone to overfitting problem, gradient boosting methods theoretically perform better than AdaBoost.

Between different types of gradient boosting algorithm, we consider that LightGBM was more appropriate for our problem, and related with our limited computational resources. In fact, LightGBM optimizes runtime speed and accuracy, because it adopts the histogram-based algorithm, splitting the continuous variables into different buckets, and also uses leaf-wise tree growth method instead of the level-wise tree growth method (used by XGBoost). The leaf-wise tree growth method allows the leaf part with higher loss to continue to grow and thus minimizes the overall loss function. So, in general the training time for XGBoost kept on increasing with an increase in sample size almost linearly, while Light GBM requires only a small fraction of this time even if the performances of the two models go hand-in-hand.

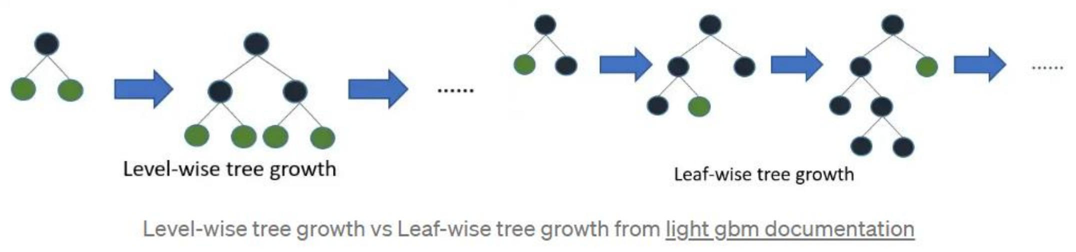

Level-wise tree growth
Leaf-wise tree growth
Level-wise tree growth vs Leaf-wise tree growth from light gbm documentation

Figure 19 - Level-wise tree ecc.ecc.

So, for the above explained reason, among all the possible bosting classifiers we decided to use LightGBM. There are many parameters associated to this boosting ensemble so, also in this case, as was done for the random forests, we used the **Optuna** framework as a support in this non-trivial search of hyperparameters. So, because of this reason we followed the same path made for random forests, first trying a cross validation approach, then a validation-set approach with Pruning feature and finally a dimensionality reduction approach.

Regarding to the **cross-validation** approach it was conducted a *study* for 100 trials on the following parameters:

Parameter	***Values***
max_depth *The maximum depth of the tree*	*[3-10]*
learning_rate *boosting learning rate*	*10**[-4, -1]*
boosting_type *Different types of gradient boosting methods*	*['gbdt', 'dart']*
min_child_samples *Minimum number of data needed in a leaf*	*[20,400]*

subsample *Subsample ratio of the training instance*	*[0.1,0.9] step 0.2*
colsample_bytree *subsample ratio of columns for each tree*	*[0.1,0.9] step 0.2*
min_data_in_leaf *minimal amount of data in one leaf*	*trainset_size/[50,350]*

Table 10 - Parameters utilized for LightGBM with cross-validation

Following are reported the obtained results:

max_depth	***Exp_lr***	***subsample***	***colsample_bytree***	***m_data_in_leaf***	***boosting type***	***accuracy*** *on validation set*
4	*0.0001*	*0.5*	*0.5*	*83*	*'dart'*	*100*

Table 11 - The parameters and accuracy of best methods for cross-validation

Also are reported the optimization History plot from which it is possible to see that the best trial obtained was the second and the Parallel coordinate plot from which it is possible to see that there are no values of the hyperparameters that are never selected, and so that each one of this could potentially be a good value respect to the environment, and so respect to the other parameters.

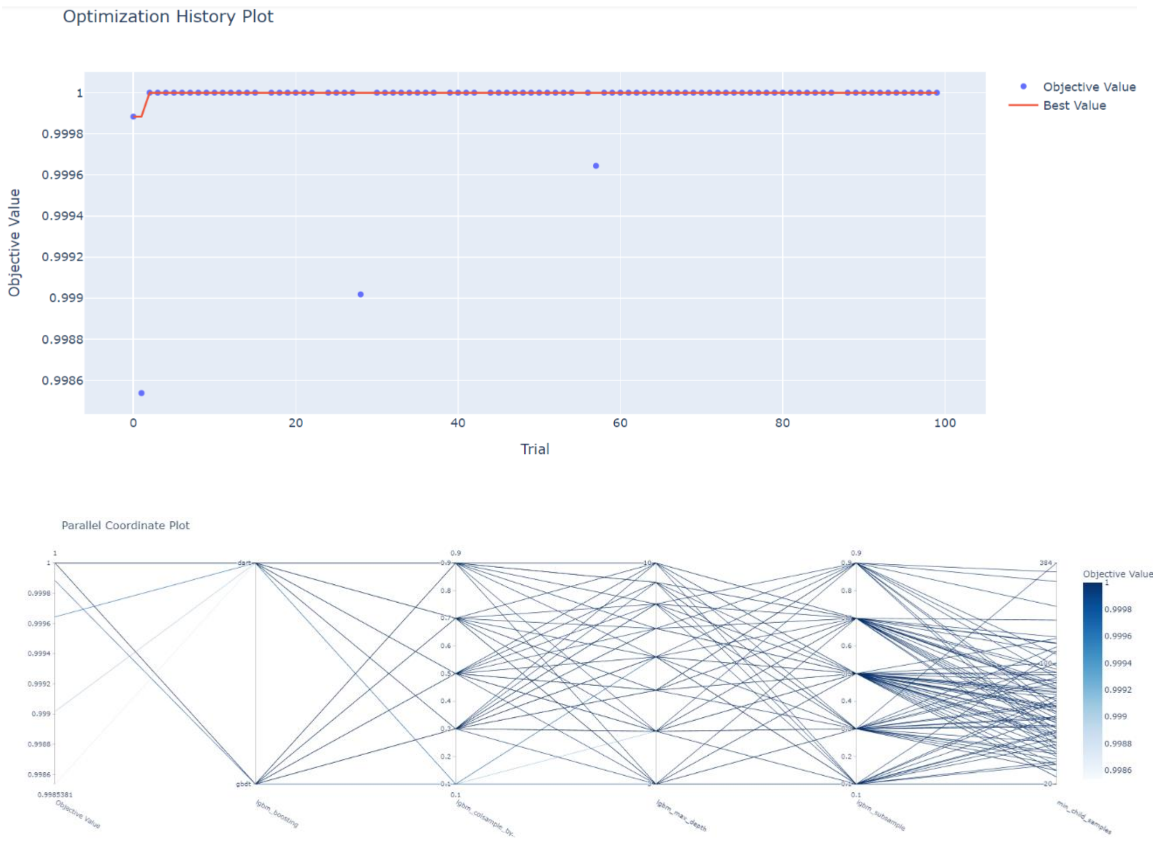


Figure 20 - On the top the Optimization History Plot, on the bottom the Parallel Coordinate Plot.

Only for a completeness in the description, also the value obtained for the other two experiments are reported, even if, as discussed above for the random forests, these experiments have been conducted only for testing purposes.

Regarding to the **validation-set** approach with **Pruning** feature and the **dimensionality reduction** approach it was conducted a *study* for 100 trials on the following parameters, that are the same for both experiments:

Parameter	***Values***
max_depth *The maximum depth of the tree*	*[2-32]*
learning_rate *boosting learning rate*	*10**[-3, -1]*
subsample *Subsample ratio of the training instance*	*[0.1,0.9] step 0.2*
max_features *Number of features to consider for the split*	*[0.1,0.9] step 0.2*
features_fraction *The fraction to select column sampling*	*[0.1,0.9] step 0.2*
max_samples *the size of the samples subset*	*[0.3,0.7] step 0.2*

Table 12 - Parameters utilized with validation-set and dimensionality reduction

Following are reported the obtained results of **validation-set** approach:

max_depth	***Exp_lr***	***subsample***	***max_features***	***feature_fraction***	***max_samples***	***accuracy*** *on validation set*
2	*0.1*	*0.9*	*0.9*	*0.9*	*0.5*	*100*

Table 13 - The parameters and accuracy of validation set approach.

Following are reported the obtained results of **dimensionality reduction** approach:

max_depth	***Exp_lr***	***subsample***	***max_features***	***feature_fraction***	***max_samples***	***accuracy*** *on validation set*
11	*0.1*	*0.7*	*0.9*	*0.9*	*0.7*	*97.985*

Table 14 - The parameters and accuracy of dimensionality reduction

So also in this case the cross-validation approach proved to be the best, both in terms of accuracy and in terms of the values chosen for each parameter, also as above explained it gives use much more guarantees respect to the stability of results.

Ensemble from scratch

The last ensemble classifiers architecture we tried was one made from scratch, combining some of the best weak classifiers we have evaluated. In particular we use this approach because, compared to the use of already designed ensembles, such as Random Forests or LightGBM it left us with greater freedom of action with respect to the type of classifiers to be involved and also the way in which they are combined. To realize this system, we use three different classifier, two of them representing the two most promising classifier among those tested, and so a random forest and a LightGBM whose parameters have been obtained through the Optuna optimization study above descripted. The last classifier, instead, is a simple K-NN. The hyperparameters of each weak learner are reported below.

LightGBMClassifier:

max_depth	***num_leaves***	***lr***	***subsample***	***colsample_bytree***	***m_data_in_leaf***	***boosting_type***
4	*128*	*0.0001*	*0.5*	*0.5*	*83*	*‘dart’*

Table 15 – Parameters of LightGBM classifier.

RandomForest:

max_depth	***num_estimators***	***max_features***	***sample_split***
3	*28*	*‘sqrt’*	*6*

Table 16 - Parameters of Random Forest

K-NN:

K
3

Table 17 - Parameters of K-NN

Then the output of these two classifiers has been combined using a ***hard majority voting***, in this way the response is that which obtained the maximum agreement between the different classifiers. We choose this approach respect to the soft voting, because in this second one approach the combination rule is made using the probabilities output from the Softmax, and we consider riskier rely on this aspect because the three classifiers are different from each other, and we would have been forced to make an assumption about their confidence level during the prediction phase. With this approach the obtained accuracy on the validation set was **99.7%**.

Chosen approach

Among this three classifiers ensemble we consider that the best one is the third and so this made from scratch. This because, even if the obtained accuracy on the training set was slightly lower, it gave us more guarantees in terms of robustness. In fact, it is an ensemble with a limited number of weak learners who are above very different from each other, ensuring that they are able to obtain the property of orthogonality with greater probability, and making the most of the power of an ensemble.

Accuracy on the test set

Because of the structure of the test set, we expected that the performance obtained will be very good; in fact the distribution and the type of samples in the test set are very similar to those of the training set, the families of malware are in fact the same.

Also among all the possible approaches, we choose that with the more general architecture, and also all our validation methods are oriented to obtain stability and reproducibility like cross-validation approaches. Accordingly to what we expected, the obtained performance on the test set is very similar to that of the validation set and in fact are equal to **99,1%**.

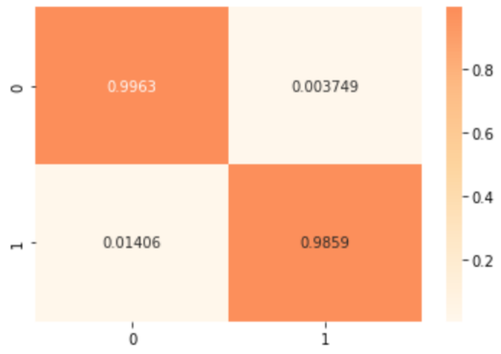


Figure 21 - Evaluation of accuracy on Test Set

Generalization Capability

An important aspect for the final evaluation of the chosen approach was to evaluate the capability of generalization, also to predict performance under real word usage. Obviously, as it was explained above, the type of malware contained in this test set to evaluate the generalization capability belong to families different from that on which our system has been trained, while the benign are of the same type. Despite these challenges the obtained accuracy is equal to **66,6%**, so not so bad. In particular, only for test purposes, after choosing what was the best model, based on the performance on the validation set, we tried the performance of the rejected methods on this generalization set, and we discovered that they were much worse.

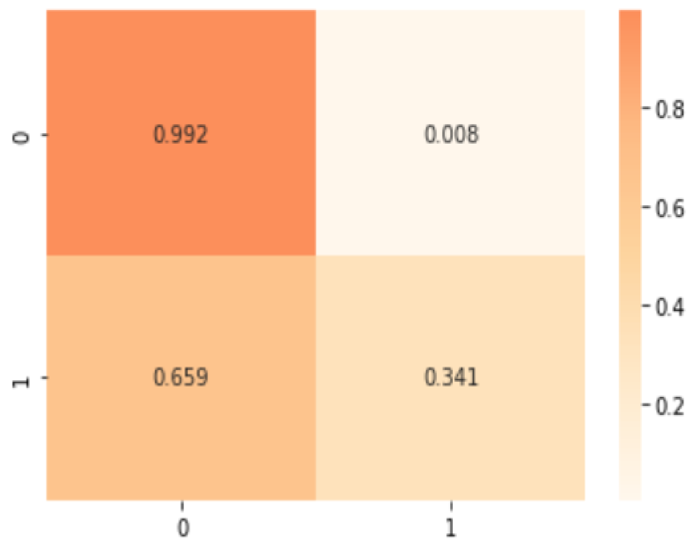


Figure 22 - Evaluation of accuracy on Generalization Set

As it is possible to see from the confusion matrix, there is a very marked difference in the obtained performance on benign and malware, it is reasonable because benign samples are very similar to those contained in the training set, unlike what happens for malware ones.

To further validate this hypothesis, we tried to modify this generalization dataset, deleting from it all the malware that are not recognized as such from at least 60% of antiviruses on VirusShare, and also deleting all malwares belonging to families different from them on which our model have been trained. The performance obtained on this modified generalization test is equal to **60,339%**, therefore the justifications proposed regarding the negative performances on this generalization test were correct.

Image Based

The second technique explored in this report is ***Image Based***. This approach leverages the generalization power of ***CNNs***, *Convolutional Neural Network*, that has been proved to be applicable to task other than Computer Vision itself with great results. In the State-of-the-Art, different works illustrate the capabilities and efficiency of this kind of models in the task of Malware Detection [2] [3].

As a baseline the approach followed consists in extracting images frow raw bytes that make up the files and fed them to the CNN that will search for distinctive patterns that can classify it as benign or malware. In the chapter follows an analysis on the techniques used to obtain the images starting from the raw data, and the different models tested with the obtained results.

Feature Extraction: Binary Images

As stated before to train a CNN, a dataset of images was needed. At a Machine Learning standpoint, an image is seen as a matrix of integer values between 0 to 255.

In literature there are different kinds of approaches to convert malware files into images, some complex one with RGB images format, as in [14], where the different channels of the images were generated starting from the different sections of PE file (.text .data, .rdata, etc.). The approach chosen was less complex and faster. The code was written reproducing the paper: the width of the image is based on the size of the file (to be specific, based on the kb of the image. For example, if a file size is between 500 and 1000kb, the width chosen is 512). Given the size of the file and the previously calculated width, we calculate the height. After that, the given binary file can be segmented to an 8-bit vector, each 8-bits vector can be represented as one grayscale pixel, that is an unsigned integer in the range 0, 255.

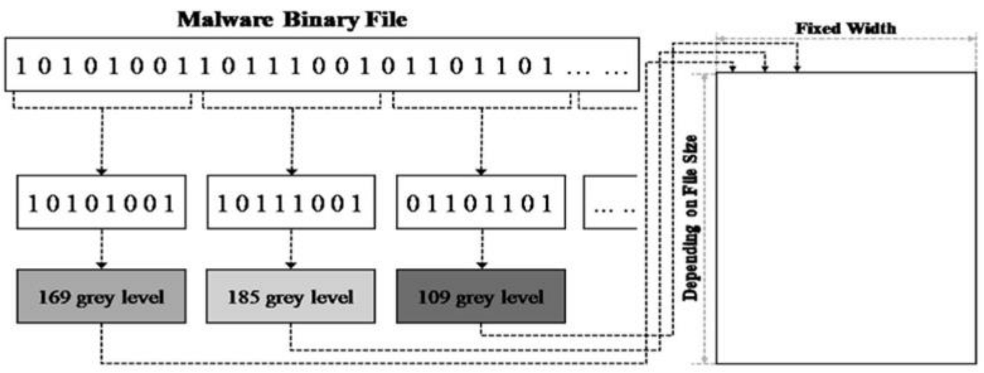


Figure 23 – Given a Binary File the first 8 bit are converted to a pixel with a specific grayscale level.

With this process the image obtained were obviously of different sizes, usually CNN operates with dataset composed of homogeneous images so these were reshaped based on the specific input of the network used. In Fig.18 there is an example of image before and after the resize.

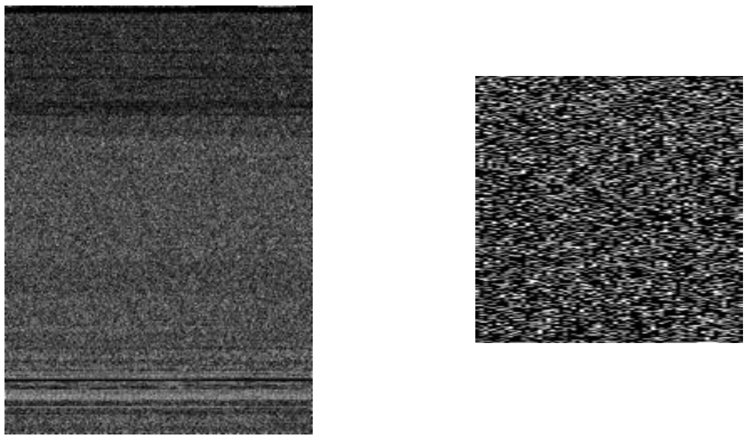


Figure 24 - The original image was resized in this case with a fixed 224x224 shape.

If the images were used with networks leveraging the weights of the ImageNet, they were appropriately normalized considering the mean and the std of the ImageNet dataset.

Analyzed Approaches

The experiments were conducted focusing on three main models widely used in the State-of-Art works. These were ***ResNet50, Inception*** and ***EfficientNet***. Before going into depth in the discussion of the results of the model and the specific approach used, there is a brief discussion of the common parameters used in the trials.

A first exploratory search was done for all the models, this was aimed at finding the best combination of *Batch size, optimization algorithms* and if train the model from scratch or use *transfer learning* with ImageNets weights.

Referring to the state of the art, several researchers have found excellent results using pre-trained network on the ImageNet, for this problem it wasn't clear if this approach would improve or not the performance. At first glance the *ImageNet*[8] challenge was based on images from the real world with little or no correlation with the task of Malware Detection from images generated through bytes analysis. Nevertheless, retaining only, the lower level of the networks could give an impressive advantage in *patter recognition* using the previous learned weights.

Regarding the optimization algorithm used, the choice fell on ***AdamW***, this is an optimization algorithm integrated in PyTorch. Often researchers prefer *SGD* over *Adam* because of better generalization capacity. In the [4] [4], the AdamW algorithm was

[8] https://www.image-net.org/challenges/LSVRC/

presented showing generalization capacity on par with SGD and a much faster training time. In the trial the *learning rate* started with a value of *0.005* and a multiplicative step size of 0.5 every 5 epochs.

As for *Batch Size* and *number of epochs* of training it was observed that the combination of *16* for the first and *20* for the latter was enough to stabilize the results and to get fast training time.

As loss function it was chosen the CrossEntropyLoss (also known as Log loss, aka Logistic loss) [2] provided by the framework, that also provided an interface to provide weight for the classes, this was experimented to reduce the number of false positive but gave in general worsened obtained accuracy.

After splitting the data in the *Training Dataset* in a Training and Validation set the trial on the different networks started and are reported in the following paragraphs:

ResNet50

ResNet50 is a popular deep neural network architecture that was introduced in 2015 by Microsoft Research. It is a 50-layer convolutional neural network designed to perform image classification tasks, and it has proven to be highly effective in this area, achieving state-of-the-art results on several benchmark datasets in the fields of object detection, semantic segmentation, and image captioning. The *ResNet50* architecture is known for its residual connections, which allow the network to learn and propagate information more effectively and efficiently, reducing the risk of vanishing gradients. The convolutional blocks are integrated with of ReLu activation function and a Max Pooling layer.

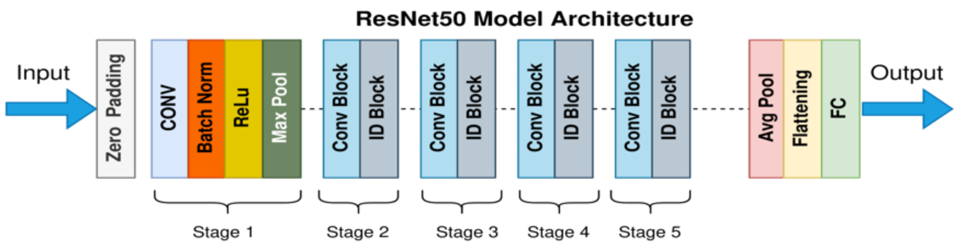


Figure 25 - Architecture of the Resnet50

PyTorch offered an implementation of the networks with the ImageNet dataset weights available, so both a ***training from scratch*** and a ***transfer learning*** approaches were tested.

For the transfer learning tests, it was decided to opt for the ***fine-tuning*** technique where, starting from the bottom, an increasing number of layers were frozen. The trials started from blocking the first 10 layer until the 45-th convolutional block. In tab.6 at size varying in range from 10 to 90 layers is showed the best results.

For this training the input images were resized to 224x224.

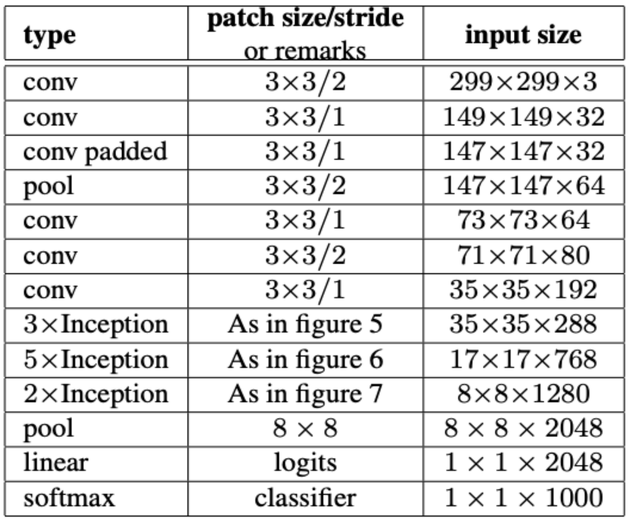
Layer Frozen	**Accuracy**
0	*0.8836*
10	*0.9146*
20	*0.8451*
30	*0.8651*
40	*0.8414*
50	*0.9005*
60	*0.9000*
70	*0.9055*
80	*0.9338*
90	*0.9337*

Table 18 - Results on the ResNet50 with different layers frozen.

Inception V3

Inception Version 3 is a deep convolutional neural network that was introduced in 2015 by Google. It is part of the Inception family of networks, which have become popular due to their strong performance on a wide range of computer vision tasks.

One of the key features of the Inception architecture is the use of "*Inception modules*" which are modules that perform multiple parallel convolutions of different kernel sizes on the input, allowing the network to capture information at multiple scales.



type	**patch size/stride or remarks**	**input size**
conv	3×3/2	299×299×3
conv	3×3/1	149×149×32
conv padded	3×3/1	147×147×32
pool	3×3/2	147×147×64
conv	3×3/1	73×73×64
conv	3×3/2	71×71×80
conv	3×3/1	35×35×192
3×Inception	As in figure 5	35×35×288
5×Inception	As in figure 6	17×17×768
2×Inception	As in figure 7	8×8×1280
pool	8 × 8	8 × 8 × 2048
linear	logits	1 × 1 × 2048
softmax	classifier	1 × 1 × 1000

Figure 26 - Architecture of Inception V3

The Inception v3 network also incorporates auxiliary classifiers, which are small classifiers added to intermediate layers of the network, to improve the overall training process. This combination of Inception modules and auxiliary classifiers allows the network to effectively learn and classify complex and diverse objects in images.

The training was carried out as the previous model, the only exception is that the transfer learning phase was handled freezing the *inception blocks* of the network. Results are available in Tab.7. The model was trained with images resized to 299x299.

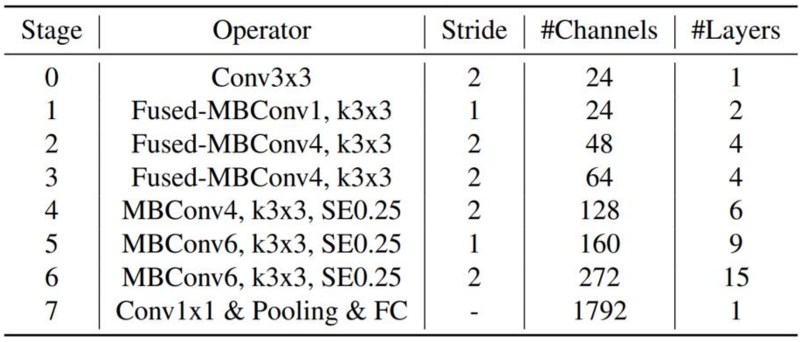
Block Frozen	**Accuracy**
0	*0.7605*
1	*0.8359*
2	*0.9142*
3	*0.9133*

Table 19 - Results on the IceptionV3 with different inception Blocks frozen.

EfficientNetV2

EfficientNet V2 is a deep learning model that has been developed to improve upon the state-of-the-art performance of previous models in image classification tasks. The model was introduced in 2021 and has quickly become a go-to choice for researchers and practitioners who require high accuracy and efficiency in their deep learning models.

The EfficientNet V2 model is designed to address some of the limitations of previous models, such as the use of hand-designed architectures, the limitations of transfer learning, and the need for large amounts of data to train. The model is optimized using a scaling method, which allows the model to be scaled to different sizes depending on the desired accuracy and efficiency. This scalable architecture makes EfficientNet V2 a flexible model.



Stage	Operator	Stride	#Channels	#Layers
0	Conv3x3	2	24	1
1	Fused-MBConv1, k3x3	1	24	2
2	Fused-MBConv4, k3x3	2	48	4
3	Fused-MBConv4, k3x3	2	64	4
4	MBConv4, k3x3, SE0.25	2	128	6
5	MBConv6, k3x3, SE0.25	1	160	9
6	MBConv6, k3x3, SE0.25	2	272	15
7	Conv1x1 & Pooling & FC	-	1792	1

Figure 27 - Structure of the EfficientNet

EfficientNet V2 is a refined version of the EfficientNet deep learning model, aimed at improving its training speed while preserving its parameter efficiency [5]. In addition to its high accuracy and efficiency, EfficientNet V2 is also designed to be easy to use and implement. The model is available in popular deep learning frameworks such as PyTorch, and can be fine-tuned on various datasets with ease. The validation of the model was conducted with the same fashion as the two previously described with results available in Tab.8.

Layer Frozen	**Accuracy**
No Transfer Learning	0.8750
600	*0.9328*
650	*0.9300*
690	*0.9332*
695	*0.9314*
700	*0.9312*
705	*0.9328*
710	*0.9341*
730	*0.9048*
750	*0.9250*

Table 20 - Results on the EfficientNet with different layers frozen.

The test was carried out to compare the performance of the fine-tuned network with that of a network trained from scratch, and to understand how much the pre-training process contributed to the final accuracy. The results of this test showed that the network trained from scratch achieved a lower accuracy compared to the fine-tuned network, indicating that pre-training was helpful in improving the performance and also the time of training requiring less parameter optimization.

Chosen approach

Therefore, after a very extensive series of consideration and training with different number of layers and combination of hyper-parameters, the best result turned out to be that of EfficientNet V2 Large. In general, all the models performed better retaining some of the layers already trained with the ImageNet weights, this probably due the partially already developed capability of pattern recognition.

The best performances were obtained freezing **710 layers** of the EfficientNet V2 L with an **accuracy of *0.9341*** on the *Validation Set*. In the figure below it is possible to see the training values for accuracy and loss. The models aren’t prone to overfitting being the Training accuracy not much higher than the validation accuracy. Furthermore, the choice of 20 epochs was correct given the reaching of a plateau in performance increase.

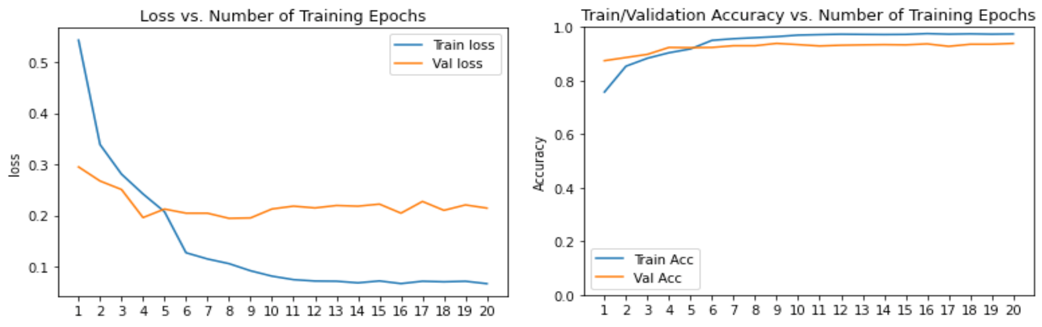


Figure 28 - Accuracy and Loss graphs for the training phase of EfficientNet V2 L

Accuracy on the test set

After this validation phase the model was tested on the ***Test Set*** to provide a thorough assessment of the model's ability to accurately classify malware samples, and to compare the performance with the other models. Results of this testing phase are shown in the following Confusion Matrix where the overall accuracy was ***93.02%***. The high performances were likely due to similarity in samples between the ***Training set*** and ***Test set*** that composed Dataset 1.

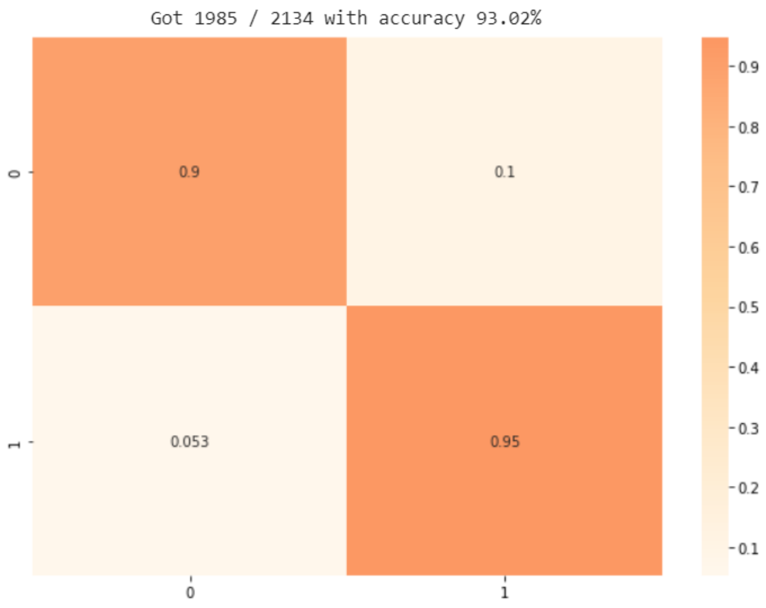


Figure 29 - Evaluation of accuracy on Test Set

Generalization Capability

After this, performances on ***Generalization set*** were tested. This revealed to be significantly worse than expected. This discrepancy can be attributed to the fact that the malware in this dataset turned out to be very different from the ones in the original dataset provided as stated in the analysis carried out in the first paragraph.

The differences in families, classes and nature of malwares heavily contributed on the degradation of performances, this is glaring in the results showed below in the confusion matrix with an accuracy of ***58.3%***.

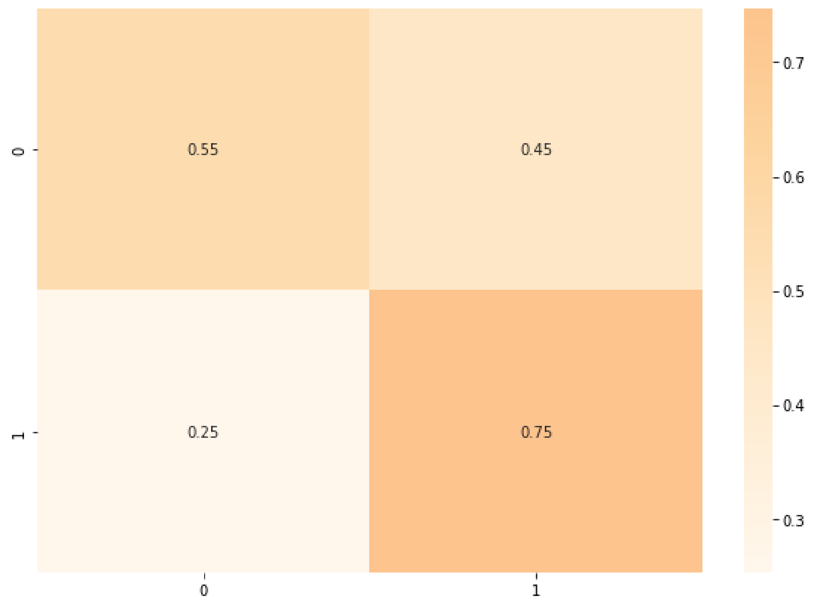


Figure 30 - Evaluation of accuracy on Generalization Set

Looking at the confusion matrix, we see a marked difference in performance between benign and malware samples, it appears that model has difficulty to recognizing malware samples. This implies that further analyzes are probably needed to improve the model on these samples.

One of the key factors that can affect the accuracy of the results is the quality and diversity of the data used for training the model. If the training data is diverse and representative of the types of viruses that are likely to be encountered in real-world scenarios, the model is more likely to be able to accurately classify new viruses. However, in the case of Generalization Set, the model may have struggled to accurately classify the viruses, with an accuracy of only 58 percent and, as further studies have shown, this could be due to the fact that the viruses in of Generalization Set are very different from those in Test Set, or because they are even more recent and not yet well-represented in the training data.

The performances were also tested on the ***Most Viral*** DataSet built as presented in the DataSet chapter. The performance obtained on Most Viral Dataset drastically decreases, this model achieved an accuracy of **16.61%.**

Following this result, some analyzes were made and it was realized that the images that were extracted from these viruses were very different from those obtained from the dataset provided to us. This shows that the viruses contained in this dataset are very different from those used for the training phase.

It is likely that the difference between the images in the two datasets is due to the fact that the viruses in the second dataset are *more recent* and have a different and an updated structure compared to those in the first dataset. This could be the result of the constant evolution and adaptation of malware to evade detection and exploit vulnerabilities in systems. As the image shows, there are significant **differences** between the malware images from the two datasets. In the second dataset, the image appears to lack distinct patterns, which may prevent the neural network from correctly classifying the image.

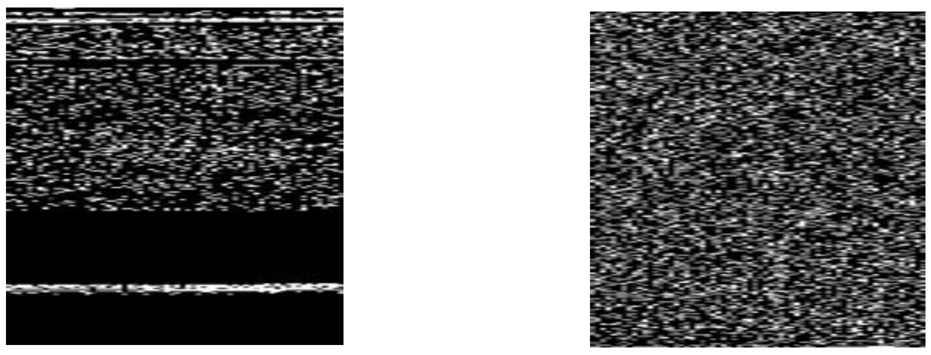


Figure 31 - On the left side a malware from DataSet 1 on the Right from Dataset2

This can be attributed to the fact that the network is trained on a specific set of patterns and structures that are not present in the second dataset. As a result, the network may struggle to recognize and classify the malware images correctly. To address this issue, it may be necessary to update the neural network with new patterns and features that are characteristic of the malware in the second dataset. This could involve collecting additional training data that is representative of the newer types of malwares, or developing new feature extraction techniques that can identify the unique characteristics of these malware types. By improving the network's ability to recognize and classify these newer types of malwares, we can enhance its overall performance and accuracy in real-world applications.

Malconv Based Networks

The major issue with the classical machine learning based malware detection system is that they rely on feature engineering, feature learning and feature representation techniques that require an extensive domain level knowledge. Moreover, once an attacker comes to know the features, the malware detector can be evaded easily. Furthermore, for CNN learning, the methodology to treat raw binaries as images loses the structure of the code, and more worryingly conflates vertical and horizontal proximity in the 2-d space (one meaningful, the other meaningless): for these reasons it can reveal a poor methodological choice.

A more sophisticated approach is to use the notion of ‘embedding space’, a concept borrowed from the NLP community where the distributional hypothesis (“words that occur in similar contexts tend to have similar meanings”) is used to justify the approach. The idea of performing malware classification using raw bytes has long existed with the purpose of alleviating expensive and constant feature engineering. ***MalConv*** was the leading network in this “revolution”, paving the way for more sophisticated approaches. Though an embedding layer of $\mathbb{R}^8$ used over an alphabet of 257 tokens (256 bytes + an End of File marker) the network analyse the files through two convolutional blocks with a gating mechanism.

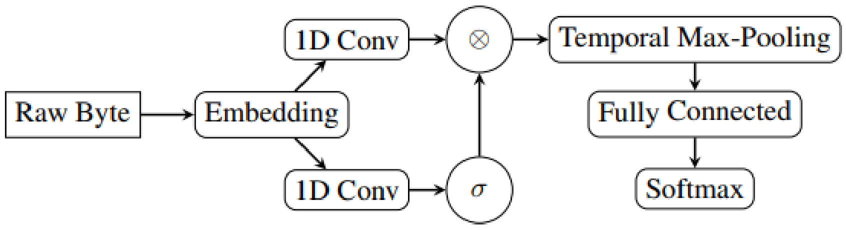


Figure 32 - MalConv Structure. Starting from the embedding layer, two Convolution block are gated.

Despite its simplicity, this first implementation of the MalConv presented huge drawbacks. The network could process only the first 2 MB of the input because it required 128 GB of GPU memory to train on a batch of 256 files up to the 2MB limit. In the following chapter more sophisticated and efficient network based on the initial MalConv ideas will be presented, after an initial discussion on the DataSet building.

To be successful, this network based on raw bytes as input, requires data with a really big variety of patterns of malware. The publicly available benchmark data for malware analysis research is very low due to security and privacy concerns. Though few datasets exist, and most of them are outdated. Furthermore, many of the published results of machine learning based malware analysis have used their own datasets, these issues are the main drawbacks behind developing this type of networks.

Dataset Generation

This kind of models didn't require any feature extraction being their main advantage the usage of the full or partial (limited by an amount specified at model creation) binary file. So, the focus was shifted on the Dataset creation and the PyTorch loader itself.

To do so the classes from the GitHub repository linked to the paper [8] [6] were used. In this paper the authors build a faster and less memory hungry version of the MalConv Network called ***MalConv with GCG*** and provide the code to truly leverage its potential.

The repository in fact contains a class called *BinaryLoader*, able to generate DataLoader compatible with the PyTorch models. Batches will have similar sizes, and will be padded accordingly to the largest file size, to minimize the excess in padding used during training. These expedients really speed up the training times.

Analyzed Approaches

The models chosen for the analysis were taken from the official GitHub repository[9] attached to the paper previously mentioned. These were the *AvastConv*, a variation of the MalConv developed by Avast[10] reasearchers, and the *MalConvGCG*, an interesting evolution of the base network that will be discussed in the proper paragraph. Along with the model's code there were also available some weights, resulting from a training of over 20 epochs and other provided by the lecturers. The training approaches for each model are reported after a brief presentation of the common parameters used for training both the models selected.

These models overcome one of the huge limitations of the original model from which they are derived, that is the *Malconv*, making possible the training and evaluation with different sizes of Malware. The choice of the file sizes was driven mainly by two factors:

Analysis of the datasets, as stated in the chapter regarding the Dataset, the average sizes of the file oscillated between 1MB and 1,3MB. This was enough to consider most of the file in the Dataset in their entirety or at least considering the main portion of the file itself where the malicious logic could hide.

The time of training, reducing the file input size had a great impact on the training time so a tradeoff between speed and malware consideration was taken into account.

At the end the choice fell on two sizes ***1MB*** and ***1,5MB*** as max inputs for the nets. In addition, as Optimization algorithms *AdamW* and *Adam* were tested with a learning rate starting from 0.005 and a scheduling that multiplied it by 0.01 every 5 epochs of training. The batch size in the range [8,16,32] because higher or lower parameter increased too much the computational time of the trainings.

[9] https://github.com/NeuromorphicComputationResearchProgram/MalConv2
[10] https://www.avast.com/it-it/about#pc

AvastConv

This network was presented by the authors with the statement: *“Our approach is the end-to-end counterpart of so-called static malware analysis: the network is given mere sequence of bytes that the executable consists of.”*.

This net is inspired by classical architecture in machine learning, it brought the peculiarity of using a deep learning model in an environment (malware detection) that used to heavily rely on hand-crafted features. The structure is outlined in Fig.22.

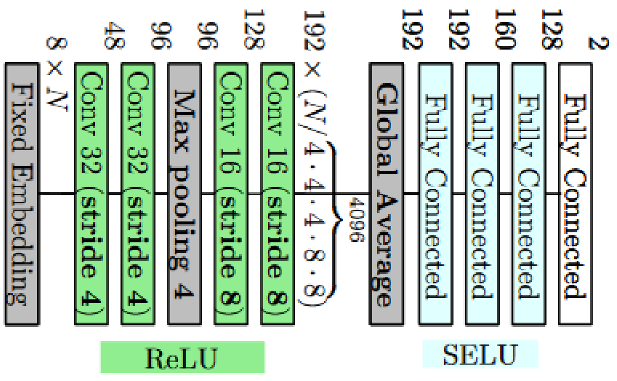


Figure 33 - AvastConv Strucutre

For the training phase both a transfer learning approach, using the weight provided, and a training from scratch were tried. The results are presented in the two Tables below:

	Parameters			**Accuracy**
Optimizer	*Transfer Learning*	*Batch Size*	*FileSizes (MB)*	
	False	*8*	*1*	*89.00*
	False	*16*	*1*	*92.37*
	False	*32*	*1*	*92.05*
AdamW	*True*	*8*	*1*	*94.68*
	True	*16*	*1*	*95.46*
	True	*32*	*1*	*95.41*

Table 21 - Results of AdamW optimizer on Validation Set

	Parameters			**Accuracy**
Optimizer	*Transfer Learning*	*Batch Size*	*FileSizes (MB)*	
	False	*8*	*1*	*93.68*
	False	*16*	*1*	*94.28*
	False	*32*	*1*	*91.50*
Adam	*True*	*8*	*1*	*94.78*
	True	*16*	*1*	*95.27*
	True	*32*	*1*	*94.46*

Table 22 - Results with Adam Optimizaer on Validation Set

From this tables it is possible to see that the training done without transfer learning obtained lower results compared with the transfer learning approach. Another important consideration is that the training done on only 1 MB obtain results very similar with the ones trained on 1,5MB files. In a real application was deemed a better approach to use the bigger size of the two, both for the performance showed by the models and also to analyze a bigger part of the file (more robust).

	Parameters			**Accuracy**
Optimizer	*Transfer Learning*	*Batch Size*	*FileSizes (MB)*	
	False	*8*	*1,5*	*91.64*
	False	*16*	*1,5*	*92.05*
AdamW	*False*	*32*	*1,5*	*92.17*
	True	*8*	*1,5*	*94.91*
	True	*16*	*1,5*	*95.46*
	True	*32*	*1,5*	*94.96*

Table 24 - Results with AdamW Optimizaer on Validation Set

		Parameters		**Accuracy**
Optimizer	*Transfer Learning*	*Batch Size*	*FileSizes (MB)*	
	False	*8*	*1,5*	*92.69*
	False	*16*	*1,5*	*92.82*
	False	*32*	*1,5*	*91.28*
Adam	*True*	*8*	*1,5*	*94.96*
	True	*16*	*1,5*	*94.37*
	True	*32*	*1,5*	*94.64*

Table 25 - Results with AdamW Optimizaer on Validation Set

The model used in the testing phase was the one with ***1,5MB*** of input size, ***Transfer Learning***, *batch size* of **16** and ***AdamW*** as Optimizer.

MalConv with GCG

The MalConvGCG was introduced as a solution to the high memory cost to train MalConv, making the memory use invariant to the length of the input — allowing us to train on over 200MB using a single GPU. This reduces the memory used to train MalConv by a factor of 116× while simultaneously providing an up to 25.8× speedup, reducing the compute requirements from a DGX-1 down to a free Google Colab instance, as stated by the authors[8].

In doing so this architecture avoids the problems of the original MalConv implementation, obtaining a model that uses the information in the context of each byte. This is done through a merging with a block called Global Channel Gating (GCG):

$$GCG_W(\boldsymbol{x}_t, \bar{\boldsymbol{g}}) = \boldsymbol{x}_t \cdot \sigma\left(\boldsymbol{x}_t^\intercal \tanh\left(W^\intercal \bar{\boldsymbol{g}}\right)\right)$$

Figure 34 - Formulation of the gating contribute.

This is the Structure of the network:

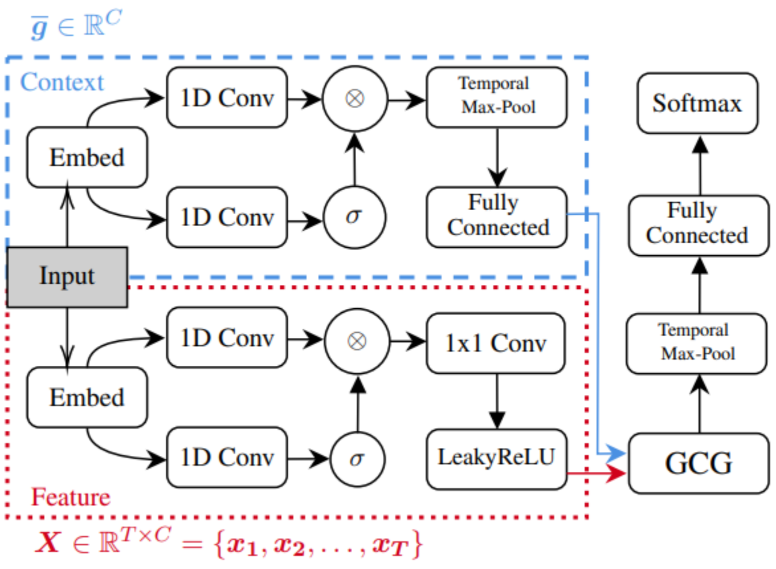


Figure 35 – Structure of MalConv GCG

For the training phase both a transfer learning approach, using the weight provided and a training from scratch were tried. The results are presented in the two Tables below:

	Parameters			**Accuracy**
Optimizer	*Transfer Learning*	*Batch Size*	*FileSizes (MB)*	
	False	*8*	*1*	*93.41*
AdamW	*False*	*16*	*1*	*93.91*
	True	*8*	*1*	*98.05*
	True	*16*	*1*	*98.00*

Table 26 – AdamW Results

	Parameters			**Accuracy**
Optimizer	*Transfer Learning*	*Batch Size*	*FileSizes (MB)*	
	False	*8*	*1,5*	*94.32*
AdamW	*False*	*16*	*1,5*	*94.91*
	True	*8*	*1,5*	*98.27*
	True	*16*	*1,5*	*98.59*

Table 27 - Adam Results

From this graph it is possible to see as the training done without transfer learning obtained lower results, so we decide to scrap them. Another consideration that is important is that the training done on only 1 MB obtain good results but, in a real application, obviously considering a bigger part of the file is better.

The model used in the testing phase was the one with ***1,5MB*** of input size, ***Transfer Learning***, *batch size* of **16** and ***AdamW*** as Optimizer.

Chosen Model

As the results on the validation set were analyzed the GCG model showed better accuracy on the Validation Set. Further results on the Testing Set and Generalization set are reported below.

Accuracy on the Test set

On the Test Set the performance were the one reported in the table below:

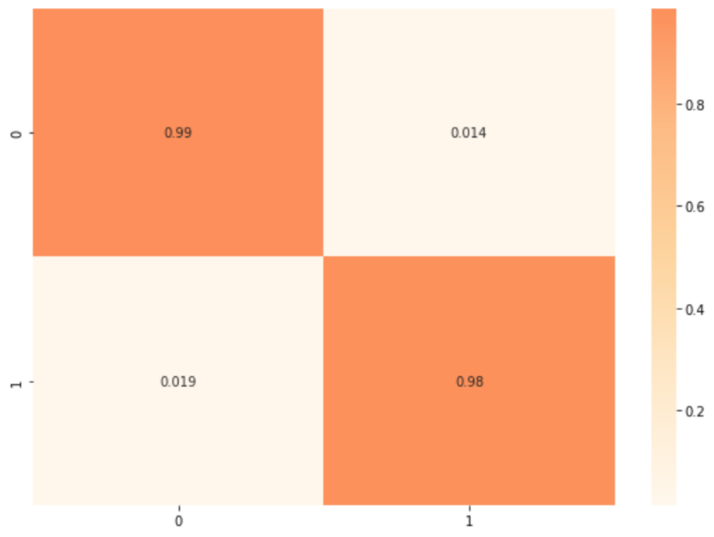


Figure 36 - Results on the Test Set

This is probably due as stated for the other models tested a real high similarity between this dataset and the Training one.

Generalization Capability

The result obtained on the *Generalization set* were quite poor in terms of malware detection. As stated in the Dataset 2 analysis paragraph a lot of factor plays in the bad performances. Compared with other models the performances are quite similar.

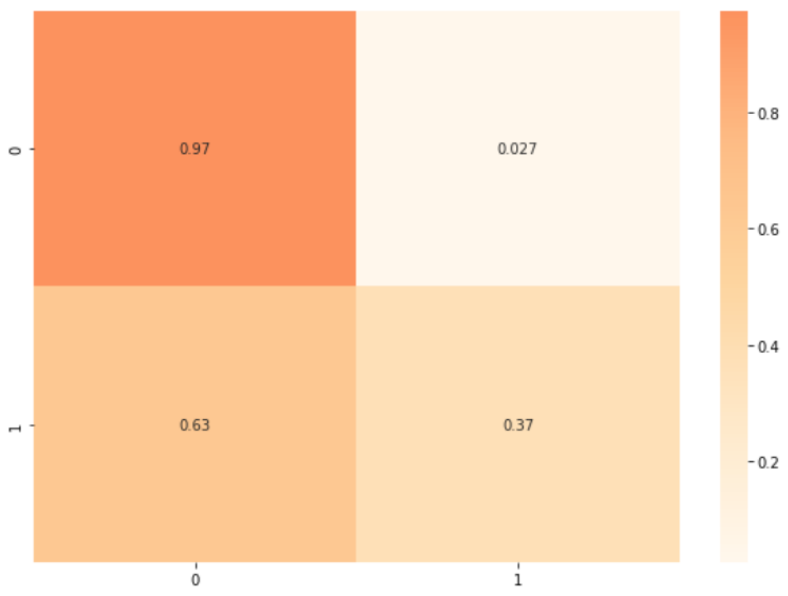


Figure 37 - Confusion Matrix on the Generalization Set

To further validate this hypothesis, the dataset was tested on the *Most Viral* Dataset The performance obtained on this modified generalization test is equal to **60,339%**, therefore the justifications proposed regarding the negative performances on this generalization test were correct.

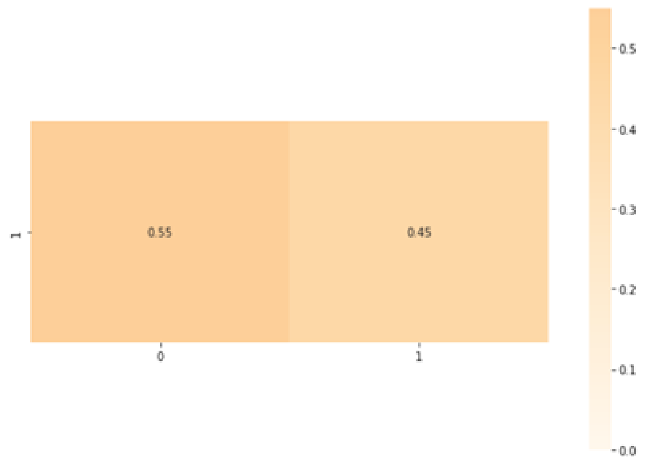


Figure 38 - Confusion Matrix on the Most Viral Set

Baseline MalConv

As a comparison for the performances obtained by the MalconvGCG model, trained with the pipeline illustrated above performances on the Test Set and Generalization Set were evaluated also on the MalConv available in the library *secml_malware*[11]. This model was trained by EndGame, a company known for providing web-based malware detection services with a dataset built upon 900.000 training samples with over 300.000 malware samples.

[11] https://github.com/pralab/secml_malware

The results are shown in the following confusion matrix, the network shows high recognition capabilities about malware file. On the contrary malware recognition appears to be worse than the GCG model trained in the report.

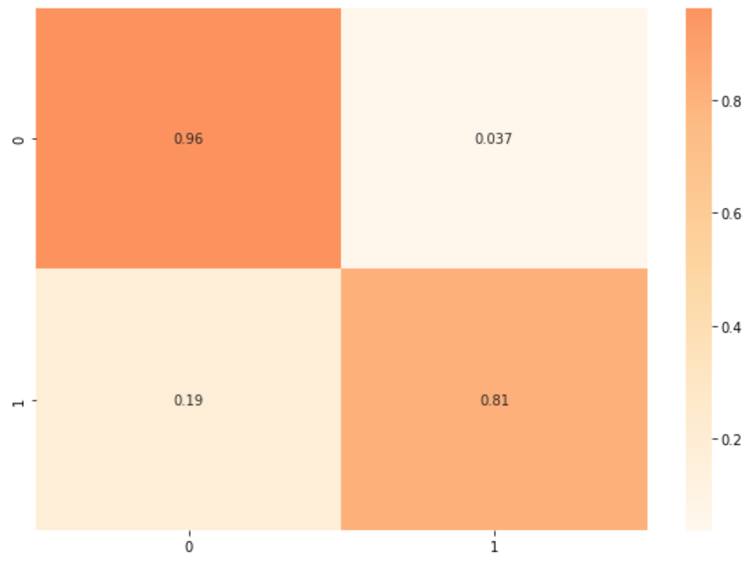


Figure 39 - Results on the Test Set

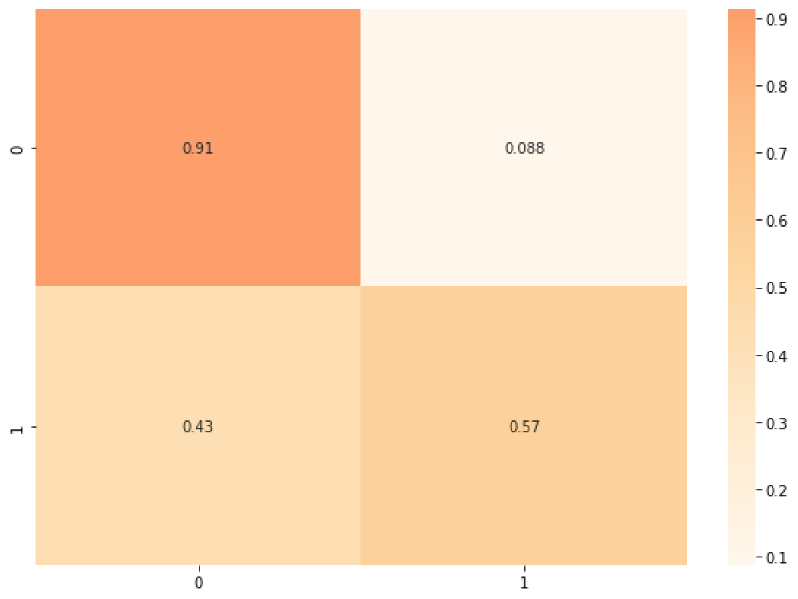


Figure 4040 - Results on Genralization Set

These results are easy to explain, the network from the *secml_malware* package is trained on a huge and diversified Training set, with different classes and families of malware. Probably it had also the opportunity to leverage huge training resources that enabled the researchers to experiment with different and more demanding configurations.

To further confirm this, it was also evaluated on the *Most Viral DataSet* which is composed of 295 Malware that comes from Generalization set but with 60% of agreement on Virus Total, results are available in the table below.

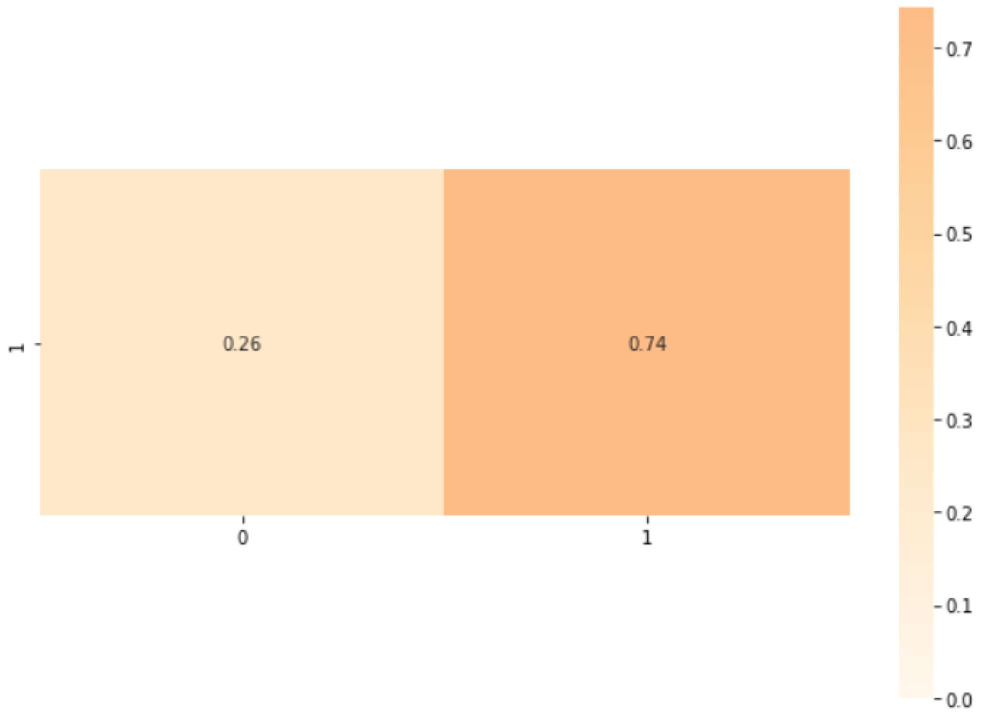


Figure 41 - Confusion Matrix on Most Viral Dataset

The strength of this network on this malware, support this hypothesis who it achieve quite good performances as generalization capability. It is also a good information due to the fact that “*secml-Malconv*” was used in the following chapter to carry on the Gamma attack.

Robustness of chosen approaches

At the beginning of this report, we underlined how nowadays the consequences caused by malware are always increasing in quantity and quality terms, because even if the detection models are refined day by day, also the techniques for masking malware, by making it go unnoticed, are becoming more and more efficient.

Therefore, for any malware detection system it is essential to ensures its resistance to attacks that may attempt to obfuscate malicious files. In this context is inserted the following experimentation in which we tested the robustness of the chosen models. Obviously, the different malware families are already very diverse, also the number and the possible obfuscations that could be applied are very high, so a *Malware Detection* system should take many different aspects into considerations, and because of this reason it is not possible to have, regarding this problem, a "*brute force*" approach. So, it is necessary to insert a general approach with good performance in all situations. But at this point a question arises: "*How can computers learn to solve problems without being explicitly programmed?*" [7].

To tackle that central question different approaches were proposed and among them the use of the biological theory of evolution arised. Hence, Holland demonstrated how evolution in nature can be applied to solve problems using a technique called ***Genetic Algorithms (GA)***. GA convert a population of individuals with its respective fitness through operations including crossover and mutation following the Darwinian principle. Combining that approach with computer programs brings us to ***Genetic Programs***, which is an approach to automated machine learning that attempts to evolve programs to solve a variety of problems that has been extensively studied for decades.

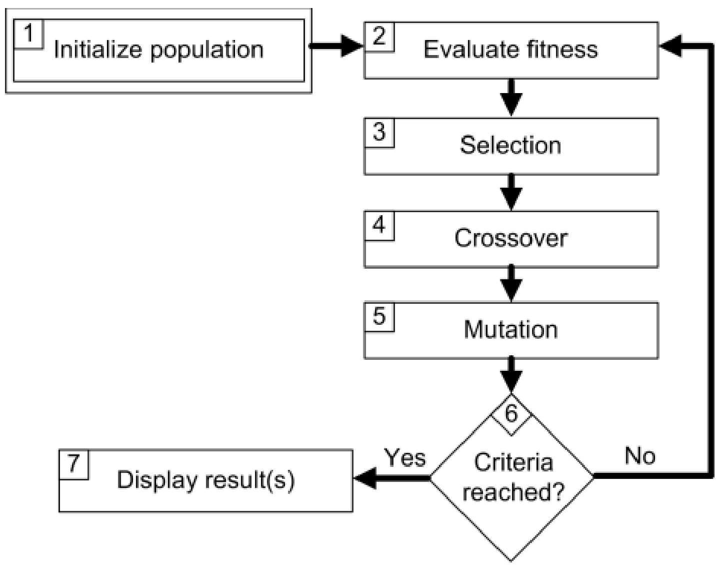


Figure 42 - GP mechanism

Among these genetic algorithms GAMMA was chosen in order to generate the obfuscated samples. The parameters required for the attack were:

- ***x*** the malware that must be obfuscate.
- ***s*** a manipulation to be applied on it.
- ***f(x)*** the output of the classification.

The aim of the algorithm is to minimize a function *F(s)* that consists of the sum of two conflicting terms:

f (x ⊕ s): the classification output on the manipulated program (where x ⊕ s is a new malware where the s manipulation is applied),

λ * C(s): a weighted penalty function that evaluates the number of injected bytes into the input malware.

λ is > 0 and tunes the trade-off between these two terms, in fact it promotes solutions with smaller number of injected bytes C(s) at the expense of reducing the probability that the sample is misclassified as benign. This was the main parameter used to generate the *Security Evaluation Curves* showed below.

Attack generation

We decided to implement a gamma attack by generating adversarial samples against the MalConv provided by *secml_malware* a repository providing different kind of attacks on ML networks. The mutations we chose to introduce fall into the category of ***section injection***. The possible sections to be injected were specified a priori along with parameters such as:

how_many: number of sections to be extracted from all the files.

bening_folder: source folder to extract the sections.

iterations: number of iterations of the algorithm.

population_size: population size at each round of the algorithm.

sections_to_extract: section to be extracted from the benign files.

penalty_regularizer: the value of lambda.

We tried ***three different configurations*** of GAMMA attacks that differ from one to each other in the number and type of sections and in the number of iterations. The obfuscated samples with each of these configurations are saved into three different folders that are ***ADV***, ***ADV2***, ***ADV4***.

The *first attack* we tried is ADV in which we do not add many sections, but we try an high value of iterations. In order to make the strength of our attack higher we try to add other sections, choosing from the ones contained into benign files, leaving the number of iterations unchanged, and so ADV2 was created. After this there are no other notable PE file sections available to increase the strength of the attack and so we tried to decrease the number of iterations to 30, in order to evaluate the impact that the iteration have on the attack generation, being sure that in this way the strength become smaller, this was ADV4.

For each configuration of these two parameters, we tried the different values of *penalty_regularizer*, in order to analyze how the attack strength varies.

Folder	***sections_to_extract***	***iterations***	***penalty_regularizer***
ADV	*[‘.rdata’, ‘.data’, ‘.text’, ‘.reloc’]*	*60*	*[0.0005, 0.0001, 5e-05, 1e-05, 5e-06, 1e-06, 1e07]*
ADV2	*[‘.rdata’, ‘.data’, ‘.text’, ‘.rsrc’, ‘.idata’]*	*60*	*[0.0005, 0.0001, 5e-05, 1e-05, 5e-06, 1e-0606, 1e07]*
ADV4	*[‘.rdata’, ‘.data’, ‘.text’, ‘.rsrc’, ‘.idata’]*	*30*	*[0.0005, 0.0001, 5e-05, 1e-05, 5e-06, 1e-0606, 1e07]*

In order to obtain the obfuscated samples, we apply the GAMMA attack with the parameters described above. Because generating this samples required a lot of computational time, they were made on a subset of the Generalization Dataset.

In particular starting this malware samples we selected only those that are recognized as malicious from at least the 80% of VirusTotal antiviruses, so we obtain a set of 95 samples, from which only those recognized as malware from the Malconv network are used to generate obfuscated samples, for a total of 92 samples.

In generating these experiments, we have paid attention in generating obfuscated samples that are not too big, otherwise the obfuscation code could be inserted in a section of the binary that was not evaluated by the MalConv network whose input size is limited, vanishing the effect of the obfuscation.

All these configurations have been first tried on the model for which the attacks have been generated and in particular for the MalConv network. The results are presented below:

Performance Obtained on *secml-malware Malconv*			
Lambda	**ADV**	**ADV2**	**ADV4**
0.0005	91.3%	79.3%	77.2%
0.0001	91.3%	76.1%	58.7%
5e-05	85.9%	63.0%	32.6%
1e-05	48.9%	22.8%	2.2%
5e-06	32.6%	10.9%	1.1%
1e-06	14.1%	2.2%	1.1%
1e-07	1.1%	1.1%	1.1%

Robustness evaluation

Starting from the obfuscated samples we generated through GAMMA, we subjected these to the models described in the previous chapters to assess the transferability property.

Some general considerations can be made for all the networks in particular the ones that are different in their architecture with respect to the MalConv. The obfuscated samples were generated considering the prediction of the secml_malware model, so these attacks weren’t tailored on different models, and so it was only possible to evaluate the transferability properties of this attack and not its strength. Furthermore, it is important to consider that probably when the attack strength increase on the MalConv, the samples are more fitted to damage it, but they lose the transferability capability causing a sort of ***inverse proportionality relationship***.

Ensemble Classifiers

In submitting the obfuscated samples to this kind of networks obviously feature extraction was necessary. This was carried out with the same methodologies as the one applied in the correspondent chapter.

The main precaution was to be aware of the impact given by the pre-processing in evaluating the reaction of this model on obfuscated samples. In fact, the trained completely rely on the representation given by the feature engineering process. The features extracted with EMBER represents an embedding containing contents about libraries, different sections, file length and entropy. This means that when the GAMMA attack inserts benign sections in a malware file, it is important to detect if these changes are reflected by the extracted features. As stated in the feature extraction paragraph of the Ensemble chapter, the different parts of the vector generated take into account the augmentation in size and entropy, but they lack a direct mapping with the particular section injected because of this reason we do not expect a good robustness of this approach in analysing obfuscated sample, and in fact the result obtained proved this.

To better understand the results reported below and the consequences of this attack it is important to report that the value of performance done on the samples used for GAMMA before obfuscating them are equal to **86.316**%.

Results on ADV:

With the obfuscated samples obtained with the *weaker* GAMMA attack, it is possible to see that there is a performance degradation increasing the strength of the attack by lambda.

For the Malconv network increasing lambda meant decreasing the strength of the attack. Observing the results obtained for the ensemble classifiers is clear that it is not possible to identify a trend of increase or the decrease as lambda varies, but instead there are swings. So, it is evident that, for this first type of experiment the transferability is quite high, because the performances are still acceptable. Considering the above explanation about the feature it appears clear that selecting a small fraction of the section for the generation of the attack wasn't enough to fully break the model.

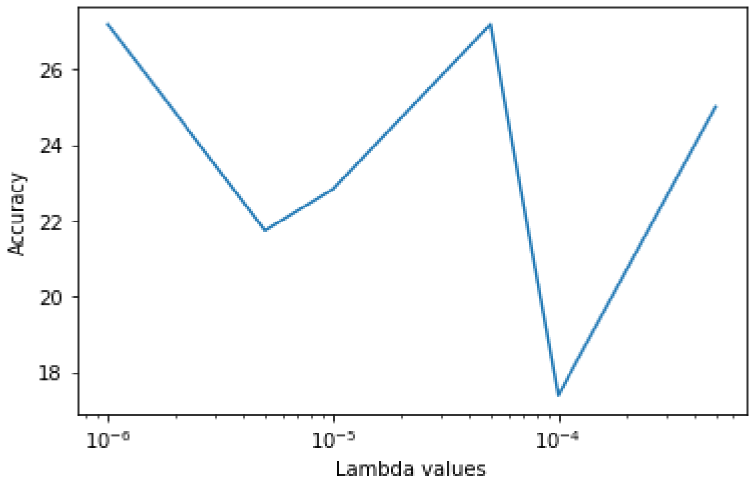


Figure 43 - Security Evaluation Curve on ADV obfuscated samples.

Results on ADV2:

The same considerations can also be done in the case of ADV2, in which the deterioration in performance is more evident. The obfuscated samples contained in ADV2 should be stronger that those contained in ADV so this attack puts the ensemble of classifier in trouble as it is shown in the *Security Evaluation Curve* above. This is because what differs from the obfuscated samples in ADV2 and the obfuscated samples in ADV is the fact that in the first type of samples more sections have been used. This means that creating the embedding of this samples, before of feeding them to the classifier, there are many differences in different points of the created features vector.

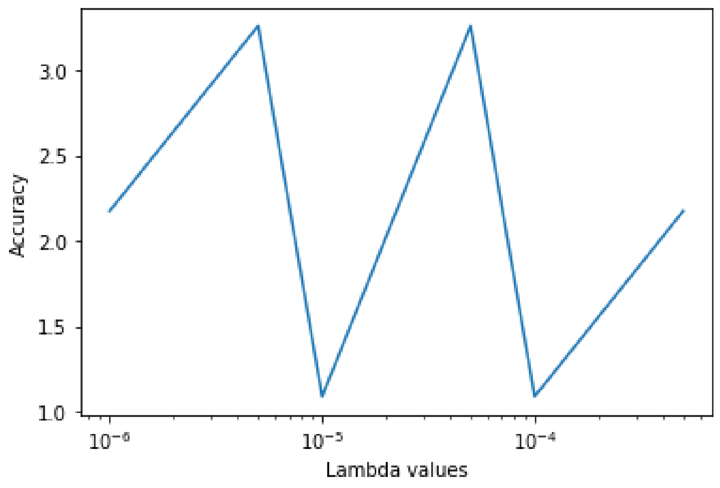


Figure 44 - Security Evaluation Curve on ADV2 obfuscated samples

Results on ADV4:

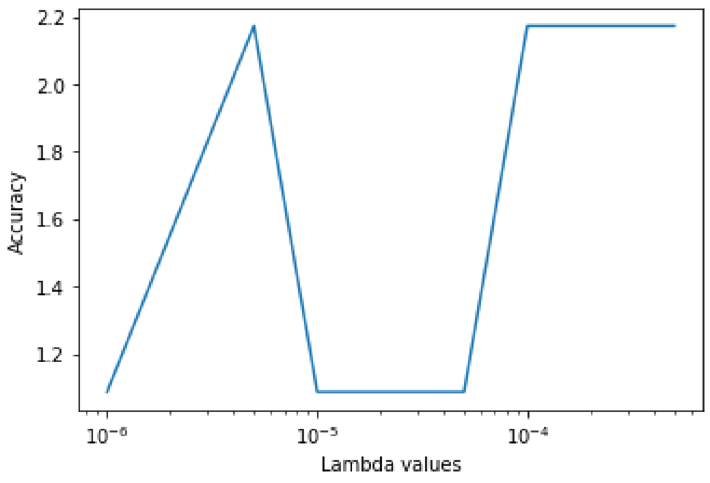


Figure 45 - Security Evaluation Curve on ADV4 obfuscated samples

Accordingly, to which happens for the MalConv these samples cause the greater performance decay. In this case probably the attack became so strong that even with different values of lambda there are no evident differences as the swings in the curve are really contained.

Image Based

As for the Ensemble Classifier also CNN networks rely on a feature extraction process aimed at generating images from binary file. So it was crucial to take into account this transformation in the process of attack generation. A section injection attack, increasing the file size has a clear feedback on image variation. This leads to pattern alteration that can make it more difficult to identify malware files.

Results on ADV:

As far as ADV is concerned, the performances in general compared to the basic samples are much worse, consequently going to obfuscate the sample has concretely degraded the performance, however they are not so drastic compared to what we expected. From the image shown, however, we can see that the pattern is very different

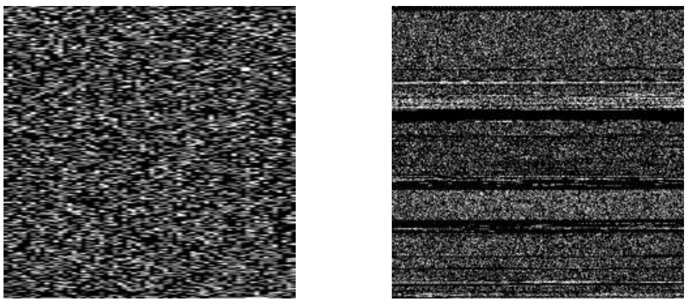


Figure 47 - On the Left the Original Image on the Right the one obtained from ADV

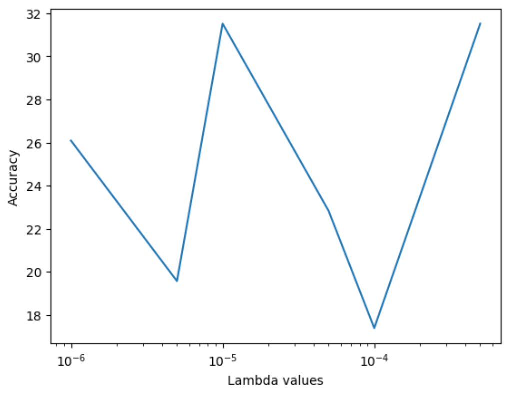


Figure 47 - ADV - Accuracy vs Lamba Values

from the previous one, thus indicating the effective functioning of the obfuscation.

Results on ADV2:

The performances between ADV and ADV2 are not too different, in general neither of the two attacks manages to completely destroy the models in terms of accuracy. Furthermore, the accuracies obtained do not differ excessively between the two attacks even if the analysis of the images makes them quite different.

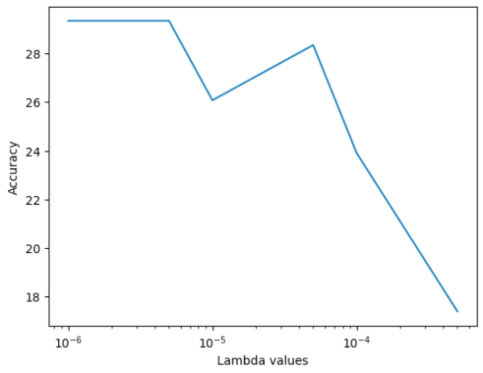


Fig 28 - ADV2 - Accuracy vs Lambda Values

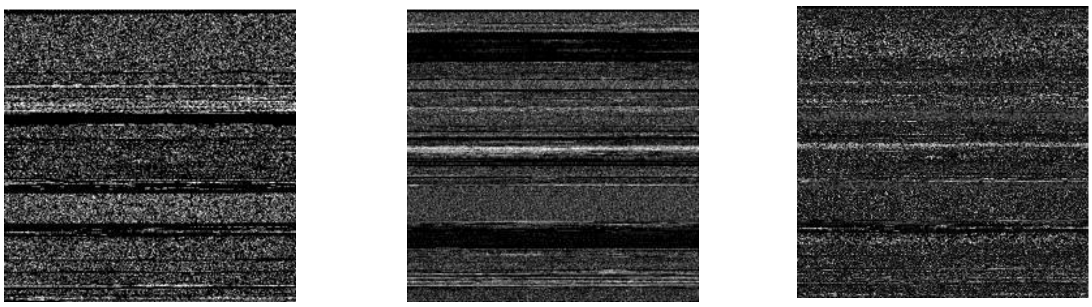


Figure 48 - On the Left the image from ADV, in the center the image from ADV2 and on the right the difference between images.

Results on ADV4:

Between ADV2 and ADV4, however, we note how with the increase in the strength of the attack, the accuracy of the network decreases. Above all, the differences are also noticeable in the generated samples, the images shown below are very different from each other. Specifically, we have completely lost the patterns we had in the previous analysis, thus making recognition much more complex. The graph below demonstrates the significant impact of the penalty regularizer on the success of the attack. As previously mentioned in earlier sections, we anticipated a decline in performance as the penalty parameter decreased. While the reduction in accuracy was not linear, the effects on performance were unmistakable.

In conclusion of this specific analysis we have demonstrated how the transferability of this attack has succeeded in an effective way above all by greatly increasing the strength of the attack itself. However, unlike what has been reported for Malconv, this attack is not immediately destructive.

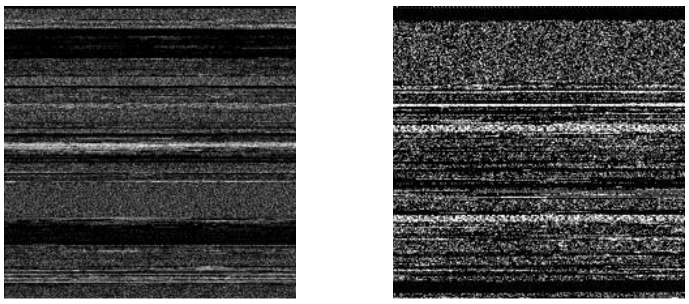


Figure 50 - On the Left the image from ADV2 and on the right the image from ADV4

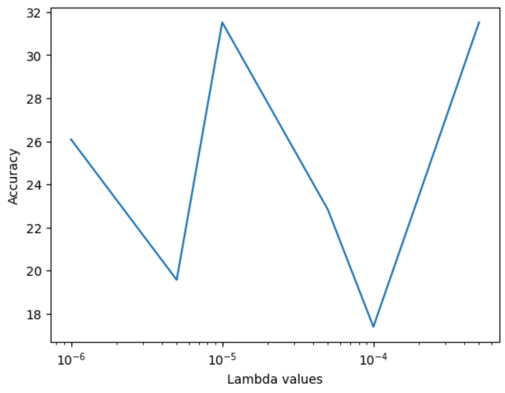


Figure 50 - ADV4 - Lambda vs Lambda Values

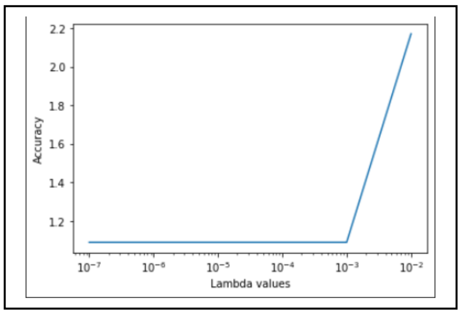
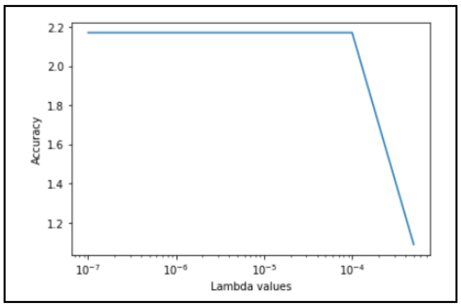
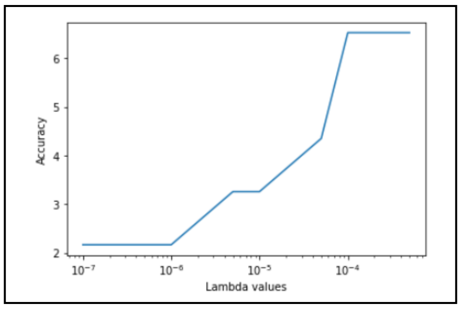
Malconv Based Networks

Deriving from the base MalConv, we expect that a black box attack on the MalconvGCG to show similar results. Is important to say that we made sure that the adversarial samples obtained through GAMMA had mean sizes below 1,5 MB that are the max input size of this network.

We deemed important to compare the performance of both this models on the not obfuscated samples as reported in the table below:

MalconGCG	**MalConv secml_malware**
55.79%	*96.84 %*

it shows that the starting performances are different between the base and the MalConv with GCG who probably look at really different pattern in this Malware and so is able to detect the sample in this dataset. Regarding the GCG model at this time we expect bad performances in terms of accuracy on GAMMA attack given the already great indecision about these samples.

Performance Obtained By Best Malconv with GCG		
ADV	**ADV2**	**ADV4**
		

As we expected regardless of the hyper-parameters chosen, this network possesses a particularly negative response, paradoxically the attack that was found to be the greatest strength on the Malconv by secml (ADV4) turns out to be less incisive on the network, fixed the value of lambda. We are analyzing, supporting the starting hypothesis that the two networks then have very different discriminants in figuring out given an input sample whether this is a malware or not.

References

[1] K. B. Joshua Saxe, «Deep Neural Network Based Malware Detection Using Two Dimensional,» 2015.

[2] X. Y. Y. W. Wai Weng Lo, «An Xception Convolutional Neural Network for malware classification with transfer learning,» IEE.

[3] T. J. M. M. O. P. P. D. P. M. P. Jin Ho Go, «Visualization Approach for Malware Classification with ResNeXt,» IEEE.

[4] F. H. Ilya Loshchilov, «Decoupled Weight Decay Regularization,» University of Freiburg in Germany, 2017.

[5] M. T. V. Le, «EfficientNetV2: Smaller Models and Faster Training,» 2021.

[6] E. Raff, W. Fleshman, R. Zak, H. S. Anderson, B. Filar e M. McLean, «Classifying Sequences of Extreme Length with Constant Memory Applied to Malware Detection,» 2020.

[7] R. L. Castro, C. Schmitt e G. Dreo, «Evolving Malware with Genetic Programming to Evade Detection,» IEEE, 2019.

[8] J. S. L. a. S. G. Hu, «queeze-and-excitation net- works.,» CVPR, 2018.