Artificial Intelligence for Cybersecurity

Create robust malware detection system using machine learning and deep learning approaches

Aliberti Luca, Amato Mario, Del Regno Michele, and Della Monica Pierpaolo

[1] Università degli studi di Salerno, Fisciano SA 84084, IT - Group 11
[2] {l.aliberti19,m.amato72,m.delregno24,p.dellamonica9}@studenti.unisa.it

Abstract. In this project, we aimed to investigate the performance of classical machine learning and deep learning methods for malware detection. To this end, we used two provided datasets: one for training and validation and one for testing. The models were trained to classify software samples as either benign or malicious. The results of the analysis revealed that both the machine learning and deep learning [5] models performed quite well in detecting malware. Furthermore, we evaluated the models' robustness by generating adversarial samples using genetic algorithms such as GAMMA [2] and found that some models were able to generalize, although with slightly lower performance, even in the presence of these samples. The results of this study demonstrate the potential of using machine learning and deep learning techniques for malware detection. Additionally, the use of adversarial samples generated with the genetic algorithms can be an effective method to evaluate the robustness of the models and to verify the transferability of malware obfuscation across heterogeneous models in terms of architecture.

Keywords: Artificial Intelligence · Cybersecurity · Malware Analysis

1 Introduction

Machine learning is becoming increasingly widespread in the field of cybersecurity. All IT fields are investing resources to apply these techniques to solve the difficult task of malware detection. This is reinforced by the fact that a significant number of malicious programs are uploaded to existing malware repositories every day, such as VirusShare.

Modern and classical machine learning techniques are used to detect such threats on a large scale, leveraging many different learning algorithms and feature sets. Those fields have lent themselves incredibly well to solving this task for two main reasons. The first is due to the ongoing evolution of attacker strategies, which give rise to the need to find a model that can generalize the discrimination of existing and evolving samples. The second is due to the fact that, at the state of the art, there are various techniques that allow for the manipulation of malware in order to evade detection, such as GAMMA [2].

Therefore, the purpose of this project is to explore three different methodologies in order to identify the most appropriate method for malware detection task. The first two are related to classical machine learning methods [4], while the last methodology uses a deep neural network approach [5] to implement malware classification. These methods will then be evaluated in terms of performance, specifically accuracy, precision, recall, F1-score, classification matrix, and ROC curve on a real malware data set.

However, the main challenge of this project is developing an effective malware detection system, due to the evolving nature of malware. Machine learning algorithms need to be trained on a diverse range of malware samples and updated regularly to keep pace with new threats.

2 Data modeling

In the following section, we will discuss aspects related to the data used both in the learning phase and in the testing phase of the proposed models. Specifically, there will be an initial data analysis to understand their size distribution, followed by a description of the workflow used to perform preprocessing and feature extraction necessary for the various learning processes.

2.1 Data description

The datasets provided for the project are mainly two. The first dataset is a subset of the dataset built by the SOREL-20M project [3], which contains approximately 8TB of files, used for models training. The second dataset, built with samples obtained from VirusShare, is used to evaluate the final performance of the model.

Initially, the two datasets were strongly imbalanced, particularly the data split is shown in Fig. 2.1. Therefore, we perform a data balancing operation adding bening files from an external provided dataset.

	Benign	**Malware**
Dataset 1 (Training)	759	6567
Dataset 2 (Test)	0	1000

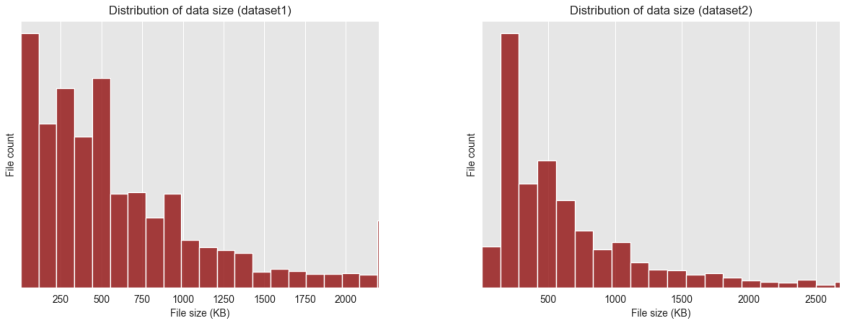
Fig. 2.1: Data division before the balancing operation

As seen from the table, the balancing was made by adding samples belonging to the benign class within each dataset. Specifically, 5808 samples were added in the first dataset, while 1000 samples were added in the second dataset.

In terms of data size distribution, both datasets have a fairly similar distribution to each other. As can be seen from the Figures 2.2a and 2.2b, both have an exponentially decreasing trend, with some outliers.

However, after a thorough analysis of the data sources, we have noticed that the data present in dataset 1 are qualitatively different from those in dataset 2. In fact, the dataset provided by the SOREL-20M project consists of disarmed malware, while the VirusShare repository includes active viruses. Therefore, a section will be dedicated to discussing the procedure for disarming the viruses contained in the VirusShare dataset, in order to assess the impact of this process on the models' training procedure and identify any potential bias issues arising from the disarming process, in order to uniform the training and testing procedures to the same type of data.

This procedure will be carried out after the model evaluation phase; since, for the purposes of the project, we mainly wanted to test the models' ability to generalize on the provided datasets, without any modifications to them.



(a) Dataset 1 (Sorel-20M) (b) Dataset 2 (VirusShare)

Fig. 2.2: Distribution of data size

2.2 Data extraction

The next section will showcase the feature extraction process, which was a crucial step in preparing the data for the two machine learning models. In the preliminary phase, we filtered the data in order to eliminate any data that wouldn't lead to correct feature extraction or was not in a suitable format for an executable.

The process involved using three separate tools to validate the data, followed by a feature extraction for each file only if it passed the validation phase.

Preprocessing. The process involved the use of three key tools. Initially, we used two tools to check the nature of the various files, verifying if they actually belonged to their respective classes. Subsequently, we used an additional methodology to ensure that relevant information could be extracted from the files, meaning that they were not obscured or partially corrupted.

VirusTotal. An online service that provides antivirus scanning on suspicious files and checks if they have been identified as malware by multiple antivirus sources. In order to provide a more reliable service, it collaborates with about 70 antivirus and security technologies to provide various perspectives on the security of the file. To this purpose, we created a Python module which interacts through the APIs provided by the service itself. After the service's analysis of the specific file, it was then evaluated using a threshold to determine whether the sample belonged to its class. Specifically, given a benign file, the number of antivirus that considered it malware was evaluated and if they were greater than a certain threshold compared to the total number, the sample was not considered benign. This procedure, of course, with an inverted logic, was also carried out for the malware.

MalwareBytes. A software application that allows to analyze all the files within a directory and then produce a report listing all files that have been identified as malware. Once the report was obtained, similar to the previously explained method, we removed all files that did not meet their classification criteria.

Finally, we extracted features related to API calls in the various files, in order to eliminate any files where it was not possible to perform this operation due to an error in the file structure.

	Benign	**Malware**
Dataset 1 (Training)	5870	6102
Dataset 2 (Test)	883	868

Fig. 2.3: Data division after the filtering operation

Extraction. We utilized two primary approaches to extract features from executable files. The first approach involved the EMBER library, which leverages LIEF to facilitate the extraction of features from an executable file, resulting in a feature vector that describes it. The second method relies on the Leaf library to extract API calls for a thorough analysis of the executable file, including the extraction of imported external libraries.

All the extracted information was then added to a csv file for easy use during the interaction with various models.

3 Methods and classifiers

In this section, we will discuss the architectures used for the proposed models and also provide the various hyperparameters to achieve the reproducibility of the conducted experiments.

3.1 Classical machine learning methods

Regarding the approach using traditional machine learning methods, after a careful research stage, evaluation of multiple methods and techniques for binary classification, we selected the binary classifier provided by LightGBM.

LightGBM. A gradient boosting framework that utilizes tree-based learning algorithms, which are widely used in both industries and academic research. Its design allows for efficiency and scalability, making it capable of handling large datasets and high-dimensional data. The LightGBMClassifier is a type of LightGBM [4] algorithm that is specifically designed for binary classification problems, where the target variable can have two possible outcomes (e.g. 0 or 1). This approach was selected due to its fast training times, high accuracy, and ability to handle large datasets.

Boosting with byte features. The baseline models used are those provided by SOREL-20M, which have been trained on a large-scale dataset consisting of nearly 20 million files with pre-extracted features and metadata, high-quality labels derived from multiple sources, information about vendor detections of the malware at the time of collection, and additional "tags" related to each malware sample to serve as additional targets. [3]

In order to use these baseline models with our datasets, it was first necessary to extract the binary features of the data, using the methods provided by the EMBER library [1], a feature set used for malware detection in the field of computer security. The features, in the EMBER feature set, are engineered to be representative of the behavior of malware, and are designed to be robust and effective for detecting a wide range of malware types. These consists of a combination of static and dynamic features that describe the behavior of a given executable file. In our approach, we used the static features, that include information such as file size, entropy, number of sections, and the presence of certain API calls.

The choice to use these models was based on several factors. Not only have they shown exceptional results in terms of performance and efficiency when compared to other classifiers, such as XGBoost, but we were also able to leverage pre-trained model checkpoints for a robust starting point for our training and testing phases.

Boosting with API features. API calls are the functions that are executed by a program to interact with the operating system or other software components. In the context of malware detection, we used API calls as a feature to describe the behavior of an executable file and determine whether it is malicious or benign. In order to do that, we leveraged the LIEF library for parsing, manipulating, and analyzing API calls from executable files. By extracting API calls from a file, it becomes possible to use these calls as features in a machine learning model to detect malware.

To accomplish this, after extracting the API calls from the executables, we employed a natural language processing (NLP) based approach to provide the features as input to the model. Initially, we implemented a methodology to obtain n-grams of specific length from the API calls. Subsequently, the n-grams were supplied as input to a pipeline consisting of two tasks. The first task performs an n-gram count by inserting the terms into a matrix, where each cell represents the Tf-idf value of the n-gram. The second task involves training the previously described LightGBM model.

In this case, it was not possible to start from any baseline, because the models provided by SOREL-20M were trained with the binary features and not with the API calls. The choice was still to use the LightGBM classifier, but obviously in this case starting from scratch with training.

Selected models. To reproduce the discussed experiments, we present the significant parameters selected for both proposed models in the Fig. 3.1.

	LightGBM (binary features)	**LightGBM (API calls)**
Number of leaves	64	31
Learning rate	0.1	0.1
Number of estimators	500	100
Max depth	-1	100
N_gram lenght	/	2

Fig. 3.1: Significant parameters for the proposed models

3.2 Deep learning methods

To utilize a different and modern approach to the malware detection problem, we focused on two popular, state-of-the-art deep learning-based detectors.

In particular, one of the two models has been coded, trained, and publicly released by Neuromorphic Computation Research Program [5].

MalConv GCG. A deep learning-based malware detection model that utilizes a convolutional neural network (ConvNet) architecture with gated convolutional groups (GCG). The top half of the network serves to learn a global context, which is used as input to the GCG. The bottom half of the architecture shows the feature sub-network.

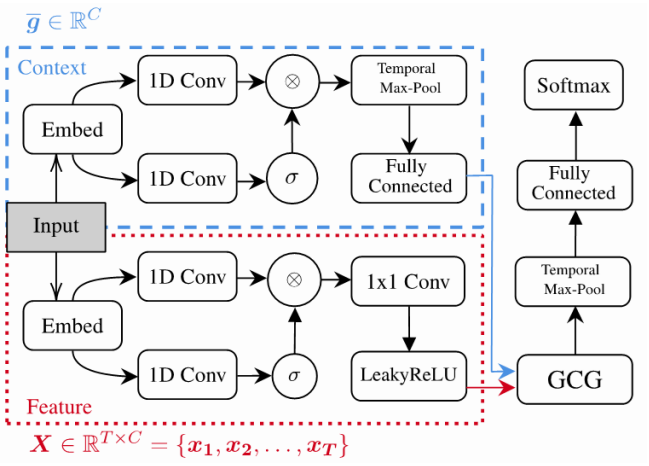


Fig. 3.2: MalConvGCG Architecture

The choice of this model is tied to several factors. The main reason was to choose a different approach from classical machine learning methods. Additionally, we selected a pre-trained model that has been trained on 20 epochs on which we performed fine-tuning, allowing us to save time resources during training. Another reason is that it is an improvement over its predecessor MalConv, since it extracts more information from the context in which the bytes are located, allowing for better generalization on malware that evolves or that are made for different platforms.

AvastConv. A deep learning-based malware detection model that utilizes a convolutional neural network (ConvNet) architecture. It is trained on EMBER dataset, which includes both malicious and benign software samples. The ConvNet part of the architecture enables AvastConv to learn important features and relationships within the raw binary code of software, allowing it to accurately identify malware. The output of the model indicates the likelihood that the input software is malicious.

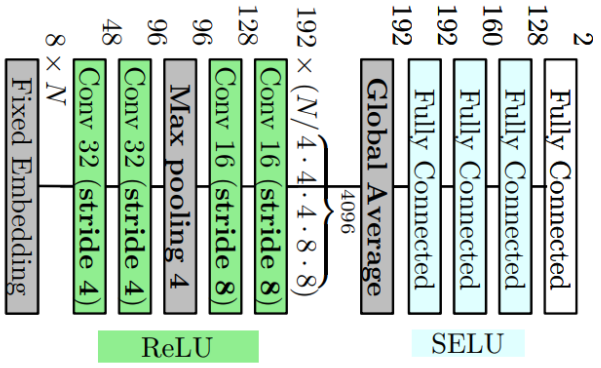


Fig. 3.3: AvastConv architecture

We chose this network mainly for two reasons. Firstly, similarly to the previously discussed model, we fine tuned a pre-trained model that was trained on 2 epochs and a dataset much larger than the one we had at our disposal. The second reason is related to the efficiency of the model, because it could process smaller inputs of maximum 1 MB in size, unlike MalConvGCG which operates on 2 MB files. Therefore, the proposed implementation reduces both RAM usage and per-epoch time, significantly reducing the training and inference times. Thus, this model seemed to be more suitable for the computational resources available to us.

4 Results

In this section, we evaluated the performance of the proposed models on the two datasets described in section 2 to understand the quality of the classification. The performance was evaluated based on some metrics of interest such as accuracy, precision, recall, F1-score, the classification matrix, and the ROC curve.

4.1 Boosting with byte features

In this paragraph we will show the performance measures of the LightGBM models trained on a binary feature set.

Confusion matrix and ROC curve on dataset 1

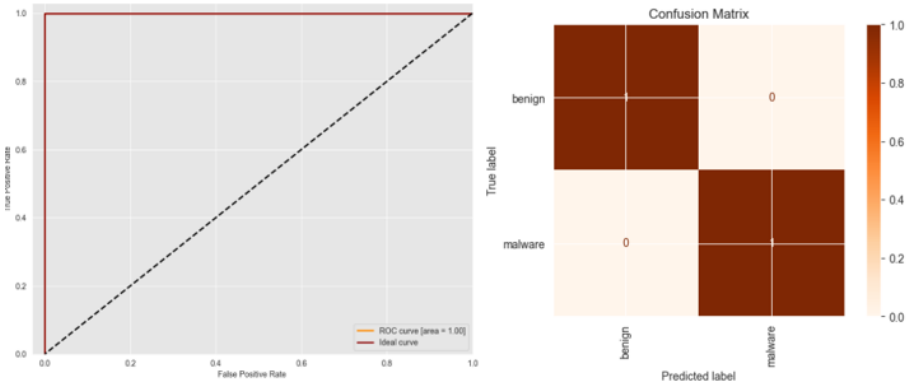


Fig. 4.1: Confusion Matrix and ROC Curve of LightGBM (seed 0, 1, 2, 3 and 4)

Accuracy, precision, recall and F1-score on dataset 1

	Accuracy (%)	Precision	Recall	F1-Score
LightGBM (seed 0,1,2,3,4)	100	1	1	1

Fig. 4.2: Accuracy, precision, recall and F1-score of LightGBM

Confusion matrix and ROC curve on dataset 2

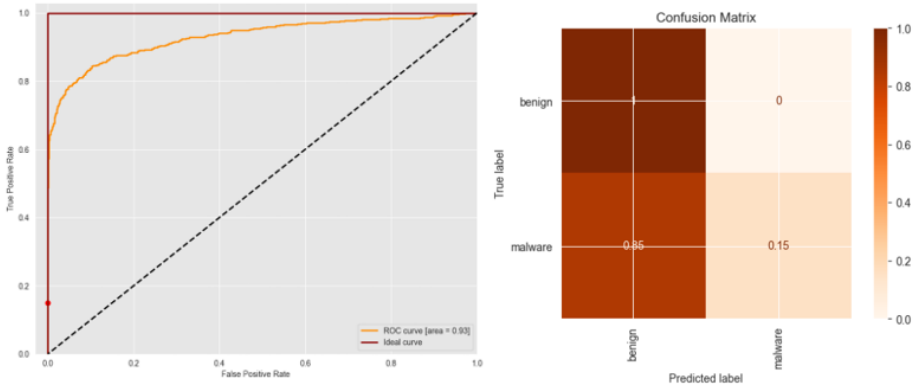


Fig. 4.3: Confusion Matrix and ROC Curve of LightGBM (seed 0 and 1)

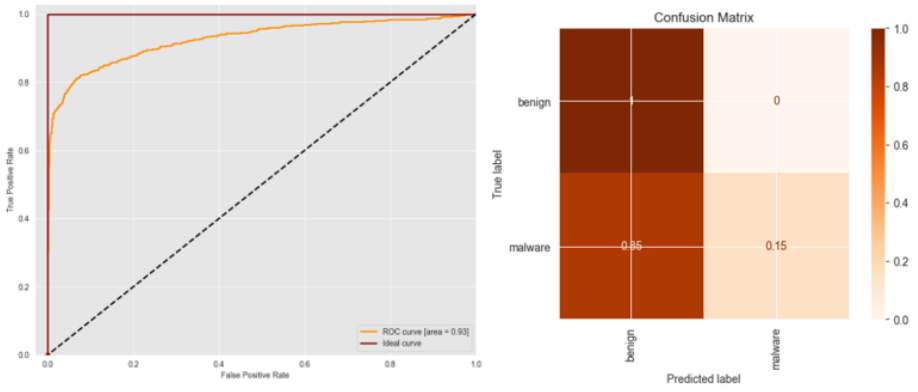


Fig. 4.4: Confusion Matrix and ROC Curve of LightGBM (seed 2)

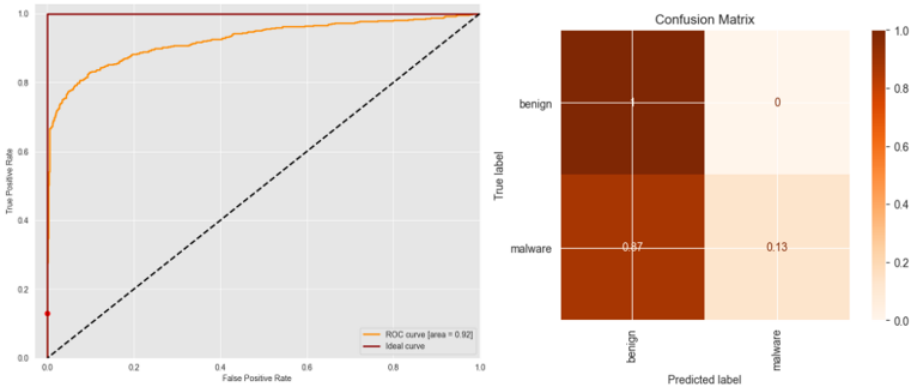


Fig. 4.5: Confusion Matrix and ROC Curve of LightGBM (seed 3)

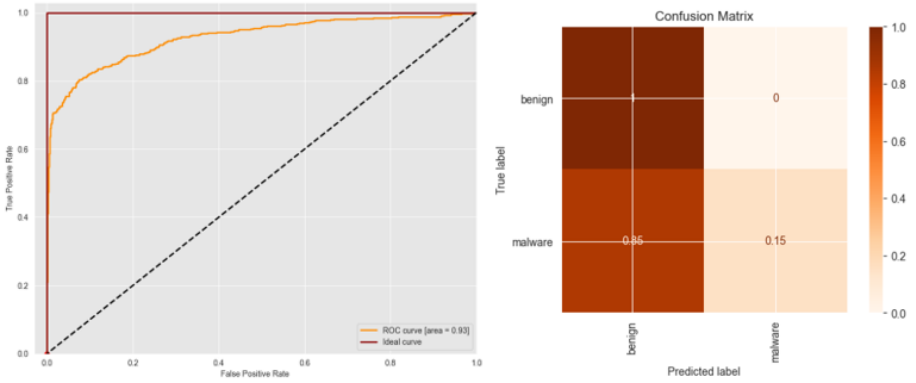


Fig. 4.6: Confusion Matrix and ROC Curve of LightGBM (seed 4)

Accuracy, precision, recall and F1-score on dataset 2

	Accuracy (%)	Precision	Recall	F1-Score
LightGBM (seed 0)	57.78	0.77	0.57	0.48
LightGBM (seed 1)	57.78	0.77	0.57	0.48
LightGBM (seed 2)	**58.08**	**0.77**	**0.58**	**0.49**
LightGBM (seed 3)	56.88	0.77	0.57	0.47
LightGBM (seed 4)	57.68	0.77	0.57	0.48

Fig. 4.7: Accuracy, precision, recall and F1-score of LightGBM

Considerations As we can observe from the graphs and performance metrics, using pre-trained baseline models on a 9 million sample malware dataset [3] has made the models significantly robust in terms of malware detection. The performance on the test set of dataset 1 is much better compared to the test set of dataset 2. A possible explanation for this phenomenon is discussed in section 5. The performances has been tested on 5 models provided by SOREL-20M, trained with 5 slightly different seeds. In any case, the performance differences among the five models do not seem to be significant.

4.2 Boosting with API features

In this paragraph we will show the performance measures of the LightGBM models trained on an API calls feature set.

Confusion matrix and ROC curve on dataset 1

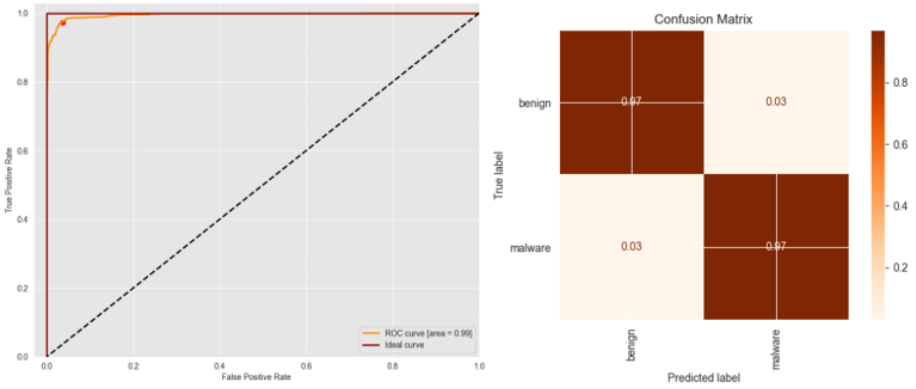


Fig. 4.8: Confusion Matrix and ROC Curve of LightGBM (ngrams = 2)

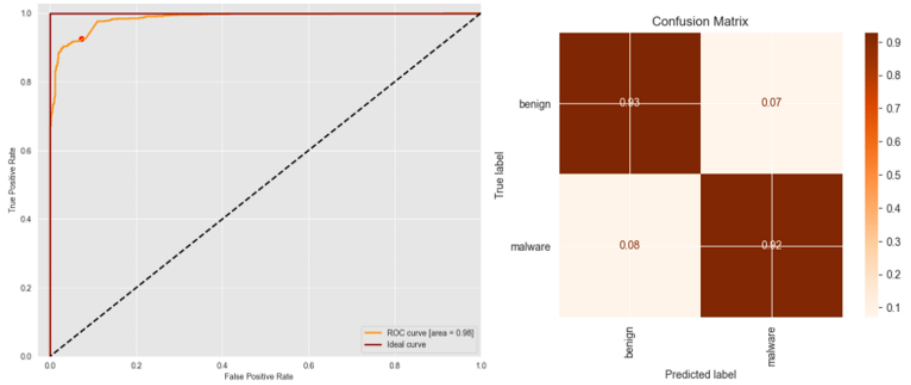


Fig. 4.9: Confusion Matrix and ROC Curve of LightGBM (ngrams = 3)

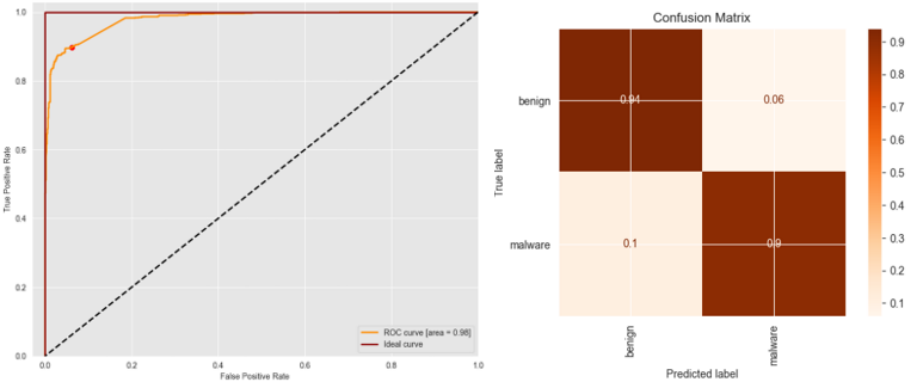


Fig. 4.10: Confusion Matrix and ROC Curve of LightGBM (ngrams = 4)

Accuracy, precision, recall and F1-score on dataset 1

	Accuracy (%)	Precision	Recall	F1-Score
LightGBM (ngram = 2)	**96.87**	**0.97**	**0.97**	**0.97**
LightGBM (ngram = 3)	92.47	0.92	0.92	0.92
LightGBM (ngram = 4)	91.77	0.92	0.92	0.92

Fig. 4.11: Accuracy, precision, recall and F1-score of LightGBM

Confusion matrix and ROC curve on dataset 2

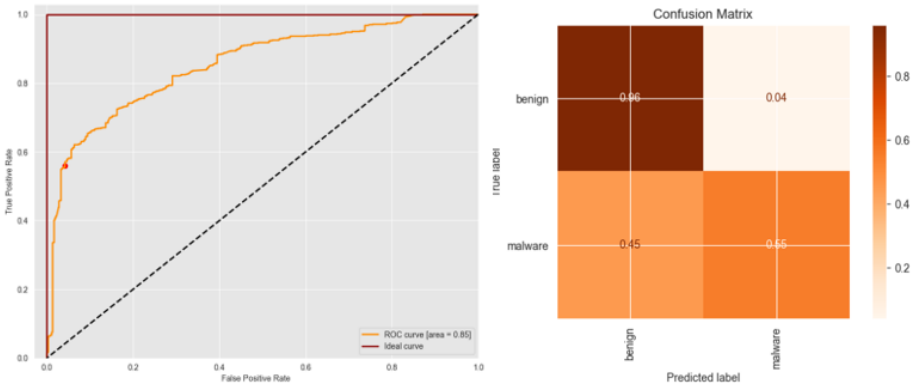


Fig. 4.12: Confusion Matrix and ROC Curve of LightGBM (ngrams = 2)

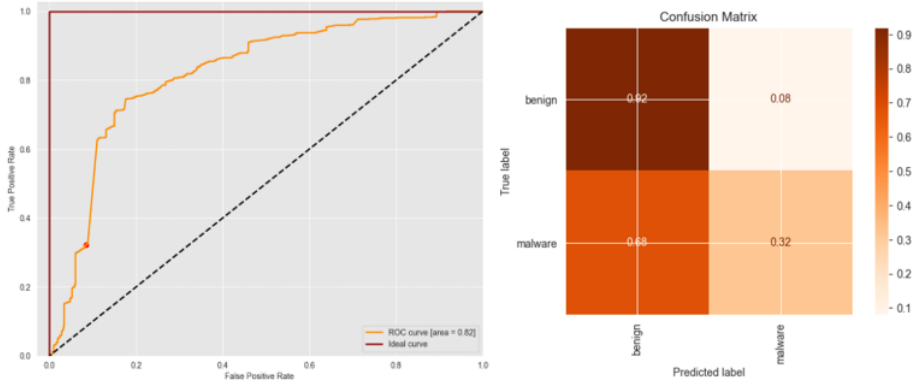


Fig. 4.13: Confusion Matrix and ROC Curve of LightGBM (ngrams = 3)

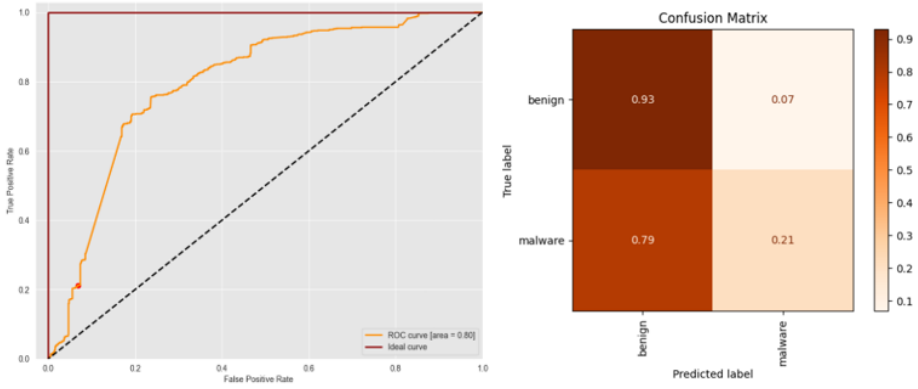


Fig. 4.14: Confusion Matrix and ROC Curve of LightGBM (ngrams = 4)

Accuracy, precision, recall and F1-score on dataset 2

	Accuracy (%)	Precision	Recall	F1-Score
LightGBM (ngram = 2)	**75.84**	**0.81**	**0.76**	**0.75**
LightGBM (ngram = 3)	62.05	0.69	0.62	0.58
LightGBM (ngram = 4)	57.35	0.65	0.57	0.51

Fig. 4.15: Accuracy, precision, recall and F1-score of LightGBM

Considerations Regarding the model trained on API calls, one of the first considerations we can make is that compared to the performances obtained by models trained on binary features, we see a clear decrease in terms of performances due to the fact that, in this case, it was not possible to reuse the previous models as a baseline, since they were trained on a completely different set of features.

The attempt, in this case, was to maintain the same line using LightGBM, but using as input for the classifier an alternative representation based only on the use of API calls through n-grams. As can be seen from the performance metrics, we have obtained better results by setting the number of n-grams to 2. This is likely due to the fact that larger window sizes carry a risk of introducing too many distinct elements into the data, resulting in overlapping information. On the other hand, using smaller window sizes can provide greater control over the data, enabling more accurate results.

However, even in this case, there is a strong degradation in performance on the dataset 2, likely due to the reasons already mentioned in the considerations of method 1 (w.r.t. section 4.1).

4.3 Deep learning methods

In this paragraph we will show the performance measures of the AvastConv and MalConvGCG models.

Confusion matrix and ROC curve on dataset 1

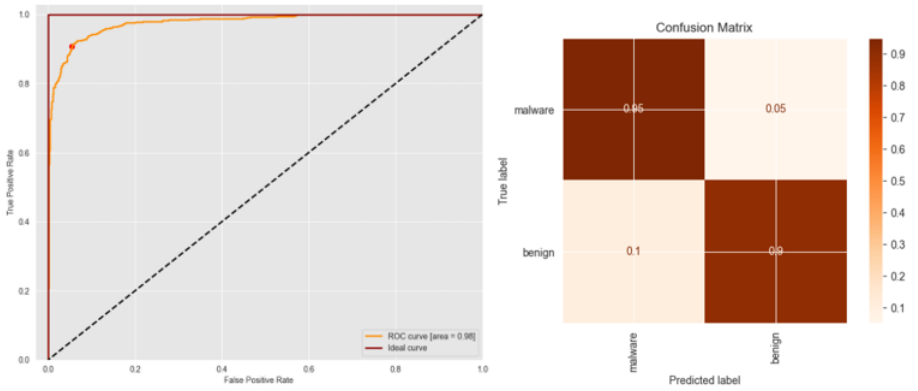


Fig. 4.16: Confusion Matrix and ROC Curve of AvastConv

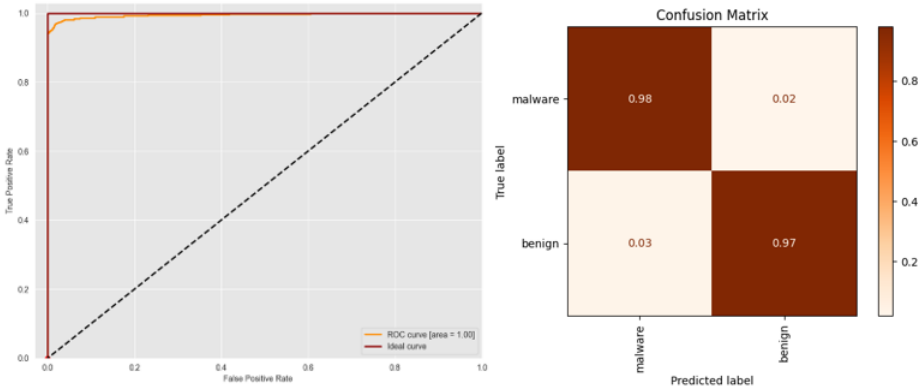


Fig. 4.17: Confusion Matrix and ROC Curve of MalConvGCG

Accuracy, precision, recall and F1-score on dataset 1

	Accuracy (%)	Precision	Recall	F1-Score
AvastConv	92.84	0.93	0.92	0.93
MalConv GCG	**97.4**	**0.97**	**0.97**	**0.97**

Fig. 4.18: Accuracy, precision, recall and F1-score of AvastConv and MalConvGCG

Confusion matrix and ROC curve on dataset 2

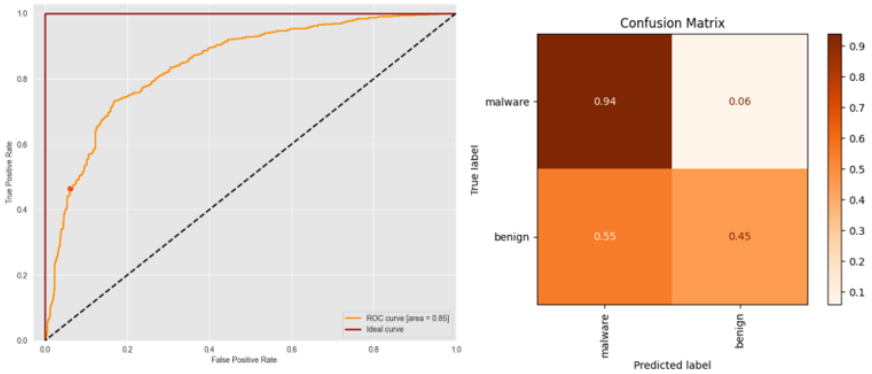


Fig. 4.19: Confusion Matrix and ROC Curve of AvastConv

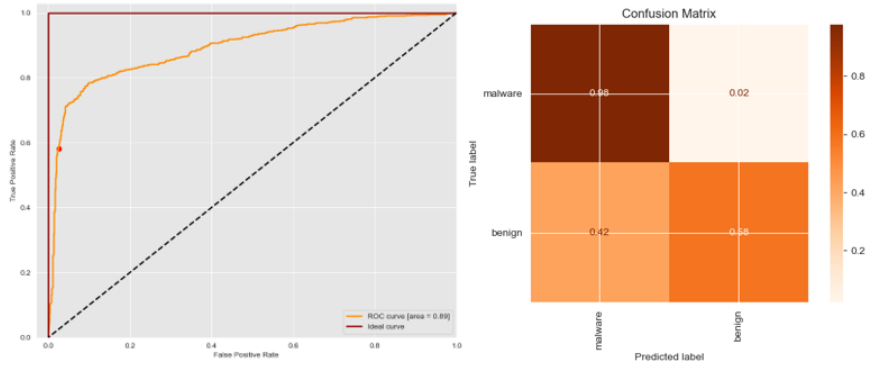


Fig. 4.20: Confusion Matrix and ROC Curve of MalconvGCG

Accuracy, precision, recall and F1-score on dataset 2

	Accuracy (%)	Precision	Recall	F1-Score
AvastConv	69.69	0.76	0.69	0.68
MalConv GCG	**78.01**	**0.83**	**0.78**	**0.77**

Fig. 4.21: Accuracy, precision, recall and F1-score of AvastConv and MalConvGCG

Considerations The differences in terms of performance between the two models is not noticeable on dataset 1, but it is clear on dataset 2. The reasons can be multiple. First of all, the pre-trained models were trained differently, in fact the base model for AvastConv was trained for only 2 epochs, while MalConvGCG was trained for about 20 epochs. Furthermore, MalConvGCG, with the advantage of having an attention module, can extract more information about the context of the bytes within the files. This advantage is also seen in terms of generalization on dataset 2, since MalConvCGC has far superior performance to AvastConv.

However, there is still a trade-off between the training and inference times and the excellent performances.

5 Possible improvements

As noted in section 4, the proposed models perform well on dataset 1, but do not perform the same on dataset 2.

Tunable system. A possible first improvement, as can be seen by looking at the ROC curves, could be to make the system tunable. In particular, allowing the adjustment of the classification threshold to tune metrics like TPR and FPR, in order to lower the false negative rate. This is useful in detection problems, since classifying a malware as benign is much more dangerous than classifying a benign as malware.

Disarm procedure. We conducted a further analysis to justify this behavior, analyzing in detail the nature of the errors. What emerges from the analysis is a difficulty in identifying the malware present in dataset 2, in fact, we can see how the performances are excellent only for the benign samples.

For this reason, it was hypothesized that there was a bias in the type of files being tested, compared to those used for training. This hypothesis was also supported by the origin of the malware itself, as the malware present in dataset 1 were taken from the SOREL-20M dataset, while those present in dataset 2 were provided by VirusShare. The difference between these two sources of malware is that SOREL-20M has applied a disarm procedure to the malware to prevent accidental executions, while VirusShare has not carried out any such procedure.

To confirm this hypothesis and try to understand if the performance degradation is related to the disarm procedure, we decided to replicate it on dataset 2, disarming all the malware.

In particular, the malware in dataset 1 have been disarmed by setting both the `optional_headers.subsystem` and `file_header.machine` flags to 0 in order to prevent execution, and we did the same for the malware in dataset 2.

The code for the disarm procedure is the following:

```
def disarm_procedure(path_armed, path_disarmed):
    try:
        files_path = [str(x) for x in Path(path_armed).rglob('*')
                      if x.is_file()]
        # Iterate over each file
        for file_path in tqdm(files_path):
            # Parse the binary file using lief
            binary = lief.parse(file_path)
            # Set the values of the flags to 0
            binary.header.machine=lief.PE.MACHINE_TYPES.UNKNOWN
            binary.optional_header.subsystem=lief.PE.SUBSYSTEM.UNKNOWN
            # Build the binary
            builder = lief.PE.Builder(binary)
            builder.build()
            # Write the disarmed binary to the disarmed directory
            path_to_write = path_disarmed + os.sep +
                            file_path.split(os.sep)[-1]
            builder.write(path_to_write)
    except Exception as e:
        print(e)
```

Listing 5.1: Code for the disarm procedure

5.1 Results after disarm

After the disarming of the malware, we repeated the testing procedure for all the models on the dataset 2.

The results obtained from testing the previously exposed models on the new dataset - which was built from dataset 2 and subjected to the disarm procedure for the malware - reveal that the performance of the models based on API calls and deep learning methods remains unchanged due to the fact that the information altered during the procedure is not significant for the classification of the samples. However, we observed a significant improvement in the method based on the binary feature extraction through EMBER.

The result confirms the previously established hypothesis. The reason for the poor performance of the model tested on dataset 2 is most likely due to the presence of disarm flags within the header. Specifically, during training, the network most likely established a correlation between the disarm flag and the fact that the sample was a malware.

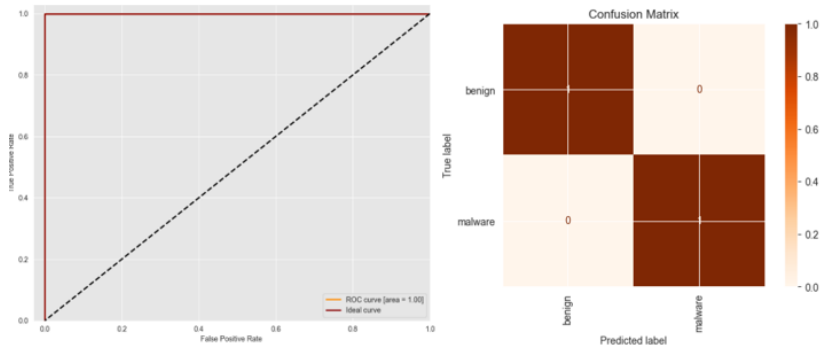


Fig. 5.1: Confusion Matrix and ROC Curve of LightGBM (seed 2) on dataset 2 after disarm

According to Fig. 5.1, we can immediately observe a significant improvement in performance. In particular, the table below shows the improvement by comparing in detail the performance metrics of the two models. The first is tested on the dataset with non-disarmed malware, and the second on the one with disarmed malware.

	Accuracy (%)	**Precision**	**Recall**	**F1-Score**
LightGBM seed 2	58.08	0.77	0.58	0.49
LightGBM seed 2 (disarmed)	99.943	1	1	1

Fig. 5.2: Comparison of performance on dataset 2 before and after disarm.

6 Attacks

In this section, we present a procedure to generate adversarial samples. Adversarial samples are malicious inputs designed to fool machine learning models, such as malware detectors.

The purpose is therefore to evaluate the effectiveness of the attacks in evading detection by the malware detectors, also paying attention to the cost to implement the attack.

6.1 GAMMA

GAMMA (Genetic Adversarial Machine learning Malware Attack), is a novel black-box attack framework, that can optimize adversarial malware samples while only requiring black-box access to the model, without accessing its internal structure and parameters. The attack is based on functionality preserving manipulations that exploit the PE format, used to store programs on disk, without altering its execution traces. The aim is also to formalize the attack as a constrained optimization problem which aims to maximize the probability of evading detection while also minimizing the size of the injected content via a specific penalty term and the number of queries to the models, making the attack stealthier.

In order to formalize the attack, it is necessary to introduce the fitness function $F(s)$, and explain why this attack is constrained to a minimization problem:

$$\begin{aligned} &\underset{s \in \mathcal{S}}{minimize}\ F(s) = f(x \oplus s) + \lambda \cdot \mathcal{C}(s) \\ &\text{subject to } q \leq T \end{aligned} \tag{1}$$

As can be observed, in Equation 1, the calculation of the fitness function $F(s)$ consists of two terms, which are in conflict.

In fact, the first term $f(x \oplus s)$, is the classification output of the attacked classifier, giving in input $x \oplus s$, where x is the original program (malware) and s is a set of possible manipulations that can be applied to x. The second term is a penalty function calculated through the multiplication $\lambda \cdot \mathcal{C}(s)$, where λ is a regularization factor, while $\mathcal{C}(s)$ is the penalty term.

The conflict lies in the fact that our goal is to minimize $F(s)$, but as the first term is minimized by injecting more benign sequences of bytes into the adversarial samples, the second term, specifically $\mathcal{C}(s)$, grows as the the number of injected bytes increases, thus going against the goal of minimizing the overall function.

Since optimizing the objective in a black-box manner requires querying the target model f repeatedly, we use the constraint $q \leq T$ to upper bound the maximum number of queries q that can be performed by the query budget T.

The presented algorithm uses a black-box genetic optimizer to solve the minimization problem:

Algorithm 1: Genetic optimization of adversarial malware with GAMMA.

Input : $\boldsymbol{x}$, the initial malware sample; λ, the regularization parameter; N, the population size; T, the query budget.

Output: $\boldsymbol{s}^\star$, the manipulations which minimize F.

$q \leftarrow 0$, $\mathbf{S} \leftarrow \emptyset$

$\mathbf{S}' \leftarrow (\boldsymbol{s}_1, ..., \boldsymbol{s}_N) \in \mathcal{S}^N$

while $q < T$ ***and*** *not converged* **do**

  $\mathbf{S} \leftarrow \text{selection}(\mathbf{S} \cup \mathbf{S}', F, \boldsymbol{x}, \lambda)$

  $\mathbf{S}' \leftarrow \text{crossover}(\mathbf{S})$

  $\mathbf{S}' \leftarrow \text{mutate}(\mathbf{S}')$

  $q \leftarrow q + N$

return $\boldsymbol{s}^\star$, best candidate from $\mathbf{S}$ with minimum F.

Fig. 6.1: Genetic Optimization of Adversarial Malware with GAMMA.

The algorithm utilizes a genetic optimization approach to solve a minimization problem by mimicking the process of biological evolution. The algorithm begins by randomly generating an initial population of N candidate manipulation vectors represented by a matrix S'. The genetic algorithm then progresses through three steps: *selection*, *crossover*, and *mutation*.

The selection step evaluates the candidates in S' using a fitness function and selects the best N candidates from the current and previous population. The crossover function creates a new set of N candidates by mixing the values of randomly chosen pairs of vector candidates. The mutation function then alters elements of the input vectors with a low probability.

This combination of crossover and mutation ensures that the new population is sufficiently different from the previous one, allowing for proper exploration of the feasible solution space. The algorithm performs N new queries to the target model to evaluate the fitness F on the new candidates and retains the best candidate population.

The algorithm returns the best manipulation vector s^* from the current population when either the maximum number of queries T or convergence is reached. The optimal adversarial malware x^* can be obtained by applying the optimal manipulation vector s^* to the input sample x through the manipulation operator, as $x^* = x \oplus s^*$ [2].

6.2 Preliminary operations

In this section, both the hardware components used for the attack and the various hyperparameters will be illustrated, in order to reproduce the attack. Subsequently, the main operations performed to perform the attack under the conditions we tested will be listed.

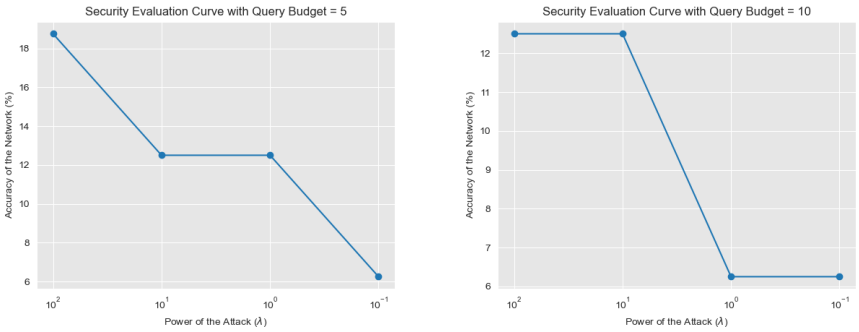
Attack setup. The evaluation of the attacks was made against the MalConv malware detector. In particular, we used a pre-trained version of this provided by the `secml-malware` library. The experiments were conducted on a Apple M1 chip with 8-core CPU and 8 GB of RAM. Due to computational limitations, we have utilized a set of 50 malware and 50 benign samples, and the attacks were tested using a population size of 6, a number of sections extracted from the benign programs of 40 (extracted from `.data` and `.rdata` sections), a penalty regularizer λ in the range from 0.1 to 100, and a number of iterations that varies in a range from 5 to 30.

Data setup. To perform the attack and accurately evaluate its performance, we identified 50 samples to submit to the genetic algorithm. The samples were chosen by ensuring that the black-box model and all our models classified them as malware, otherwise the attack would have been trivially successful. The same was done for the selected benign samples, in order to avoid injecting sections of malicious samples rather than benign ones.

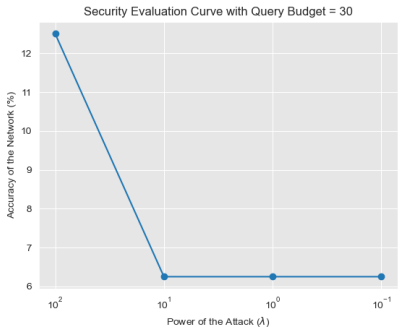
6.3 Attack results

In this section, we present the results of the attack both for the black-box model provided by the `secml-malware` library and for the models proposed in this project.

Black-box model Regarding the back-box model attack, as described in the section 6.1, by imposing increasingly stringent constraints on the penalty regularizer, the attack success rate becomes increasingly lower. This result is motivated by the fact that ever-increasing limits are imposed on the amount of bytes to be injected into the malware to be obfuscated.



(a) (b)



(c)

Fig. 6.2: Security Evaluation Curves for the attack on the black-box model. (a) SEC with query budget = 5; (b) SEC with query budget = 10; (c) SEC with query budget = 30

The results of the attack can be observed through the Security Evaluation Curves shown in Fig. 6.2. The x-axis displays the penalty regularizer which indicates the attack's penalization (λ), while the y-axis represents the accuracy of the black-box model on adversarial samples. In the shown graphs, for better visualization of the attack, the x-axis is inverted because as the penalty regularizer decreases, the attack power increases.

The graph shows that, in general, as the attack power increases, the performance of the black-box model drops dramatically. Additionally, observing the sequence of graphs, as the query budget increases, the drop in performance is reached faster. This is motivated by the fact that the genetic algorithm has the ability to generate a larger number of populations, increasing the likelihood of generating a set of transformations that minimize the fitness function.

Attack transferabilty on proposed models After generating the adversarial samples using the GAMMA algorithm and testing them on the black-box model provided by `secml-malware`, we evaluated the performance of the proposed models on these samples.

	Attack accuracy (%)	**Correctly classified**	**Missclassified**
LightGBM (binary)	0	50	0
LightGBM (API)	6,38	47	3
AvastConv	**24**	**38**	**12**
MalConvGCG	0	50	0

Fig. 6.3: Attack transferability to the proposed models

As demonstrated by the data presented in Fig. 6.3, the effectiveness of the attack varied based on the target model. The attack proves to be very effective against AvastConv, which was trained on a small dataset with only 2 epochs and has a similar architecture as the black-box model used to generate the adversarial samples, confirming the effective transferability of the attack. The attack was also partially effective against LightGBM trained on API calls. The results were quite predictable due to the fact that it is a model trained from scratch.

However, the attack proved to be ineffective against LightGBM trained on binary features and MalConvGCG due to their robustness. MalConvGCG was trained on a much larger dataset and for 20 epochs, making it significantly more robust than other models. In this case, there was no transferability due to the fact that the parameters used for the attack were not strong enough to evade classification. The LightGBM model trained on binary features was also trained on an extensive dataset (SOREL-20M), giving it an advantage in generalization ability compared to LightGBM trained on API calls.

The results of the attack also indicated that our available computational resources were not adequate to generate adversarial samples that could evade classification by the more robust models.

References

[1] H. S. Anderson and P. Roth. "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models". In: *ArXiv e-prints* (Apr. 2018). arXiv: 1804.04637 [cs.CR].

[2] Luca Demetrio et al. "Functionality-preserving Black-box Optimization of Adversarial Windows Malware". In: (2020). DOI: 10.48550/ARXIV.2003.13526. URL: https://arxiv.org/abs/2003.13526.

[3] Richard Harang and Ethan M. Rudd. *SOREL-20M: A Large Scale Benchmark Dataset for Malicious PE Detection*. 2020. DOI: 10.48550/ARXIV.2012.07634. URL: https://arxiv.org/abs/2012.07634.

[4] Guolin Ke et al. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[5] Edward Raff et al. *Classifying Sequences of Extreme Length with Constant Memory Applied to Malware Detection*. 2020. DOI: 10.48550/ARXIV.2012.09390. URL: https://arxiv.org/abs/2012.09390.