UNIVERSITY OF SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA

Artificial Intelligence for Cybersecurity

PROJECT WORK

Nome	Cognome	Matricola	E-mail
Alessandro	Accarino	0622702174	a.accarino6@studenti.unisa.it
Lorenzo	Borrelli	0622702431	l.borrelli11@studenti.unisa.it
Christian	Conato	0622702273	c.conato@studenti.unisa.it
Mariachiara	Garofalo	0622702173	m.garofalo38@studenti.unisa.it

ACADEMIC YEAR 2024/2025

Contents

1 Introduction....4
1.1 Executive Summary....4
1.2 Experimental Setup.... 5
2 Dataset Operations....6
2.1 Dataset1.... 6
2.2 Dataset 2.... 7
3 Machine Learning Model....8
3.1 Chosen Model....8
3.2 Features extraction....9
3.3 First experiment.... 10
3.4 Features Analysis.... 14
3.4.1 Features 632, 635, 655 (header - coff machine & optional subsystem).... 19
3.4.2 Features 2359, 2360 (data directories - exception table).... 22
3.4.3 Feature 97, 105 e 110 (histogram).... 25
3.4.4 Feature 623 (general - has signature).... 25
3.5 Removal of features and improvements.... 26
3.6.1 ROC Curve.... 29
3.7 Final evaluations....30
4 Deep Learning Model.... 34
4.1 Chosen model....34
4.2 AvastConv model training....40
4.3 Results of the First Experiment....42
4.4 Improvements.... 44
4.5 Final Experiment....46
5 GAMMA ATTACK.... 48
5.1 Reproducibility of the Experiments.... 49
5.2 Attack Limitations....49
5.3 First Experiment....51
5.3.1 Random Forest evaluation....51
5.3.2 AvastConv evaluation.... 52
5.4 Second Experiment.... 55
5.4.1 Random Forest evaluation....55
5.4.2 AvastConv evaluation.... 56
5.5 Third Experiment.... 58
5.5.1 Random Forest evaluation....58
5.5.2 AvastConv evaluation.... 60
5.6 Fourth Experiment.... 61
5.6.1 Random Forest evaluation....61
5.6.2 AvastConv evaluation.... 62
5.7 Fifth Experiment....64

5.7.1 Random Forest evaluation........64
5.7.2 AvastConv evaluation........65
5.8 Final Considerations........67
6 Defensive Methods........69
6.1 Adversarial training........69
6.2 GAMMA-attack detection........69
6.3 Final consideration On Certified Defenses........70

1 Introduction

1.1 Executive Summary

The aim of the project is to implement a malware detection system by exploiting Machine Learning and Deep Learning techniques studied during the Artificial Intelligence for Cybersecurity course. The following datasets have been provided for the realization of the system:

- **Dataset 1:** Comprising three subsets—training, validation, and test—extracted from PE files (malware and benign) of SoReL-20M.

- **Dataset 2:** Taken from VirusDataShare database, is intended to evaluate the generalization capability of the detection methods and to provide additional benign samples for the GAMMA attack.

Subsequently, the two classifiers were trained to solve the task, each exploiting a technique analyzed during the course:

- **Machine Learning Approach:** Employing a traditional classifier based on handcrafted features extracted from the executables.

- **Deep Learning Approach:** Implementing a detector that analyses the raw bytes (or a transformed representation).

The aim is to build models using these different approaches and then validate them, test them, and evaluate their ability to generalize to different data than those used in the training phases.

caThe transferability to commercial antivirus will also be analyzed, exploiting VirusTotal's functionalities.

Below will be described in more detail:

- The creation of the datasets used to train, validate and test the classifiers and an analysis of the composition of dataset 2 (used to test the ability to generalize), in Chapter 2.
- The operations performed for feature extraction and the training, validation and testing phases in Chapters 3 and 4;
- The configurations used to realize the GAMMA attacks and the results obtained, in Chapter 5;
- A final analysis of the defenses, in Chapter 6.

1.2 Experimental Setup

In this section, the experimental setup adopted for the project is described in detail to ensure reproducibility and justify the choices made, especially considering the resource limitations encountered.

To perform the operations outlined in the previous section, the project leverages the capabilities of Google Colab. This environment enabled the training of both the selected machine learning and deep learning classifiers by utilizing the free GPU resources provided, while managing the datasets stored on Google Drive and accessing them directly through Colab notebooks when necessary.

The decision to use Google Colab was driven by the inadequacy of local machines of all the members of the groups in terms of available RAM and GPU power. Despite some constraints on the free version of Colab, it offered a more efficient and accessible solution for conducting the experiments. The table below lists the main components, and their corresponding versions used throughout the project.

Framework	**Version**
Lief	0.16.4
Ember-mivia	0.0.6
Numpy	1.26.4
Deap	1.4.3
Pandas	2.2.2
Matplotlib	3.10.0
Tqdm	4.67.1
Python-magic	0.4.27
Ml-pentest	0.0.1
Pefile	2024.8.26

2 Dataset Operations

This chapter will detail the operations performed on the datasets provided in the specification to derive the final datasets used in the training, validation and testing operations of the classifiers.

2.1 Dataset1

The dataset made available and identified as Dataset 1 initially consisted of three folders, “train”, “validation” and “test”, both containing 2 subfolders:

- One associated with malware file executable (e.g., spyware, adware, droppers, etc…);
- One of benign executable files.

All the files in the folders associated with the particular types of malwares were placed in a single “malware” folder, while the benign files were placed in the “benign” folder.
Below there is a table showing the total number of files in these folders.

Train	#Files
Malware	4591
Benign	4376
Total	8967

Validation	#Files
Malware	1310
Benign	1240
Total	2550

Test	#Files
Malware	655
Benign	616
Total	1271

2.2 Dataset 2

Dataset 2 refers to the dataset provided to test the generalization capabilities of the models. This dataset is divided into two folders:

- One associated with malware file executable (e.g., spyware, adware, droppers, etc…);
- One of benign executable files.

All the files in the folders associated with the particular types of malwares were placed in a single "malware" folder, while the benign files were placed in the "benign" folder.
Below there is a table showing the total number of files in these folders.

Test	**#Files**
Malware	990
Benign	955
Total	1945

3 Machine Learning Model

3.1 Chosen Model

The choice of the machine learning model was based on the available datasets, described in the previous section, which were used for training and testing. In particular, given the large amount of data, an ensemble model was preferred.

Ensemble learning represents one of the most effective strategies in the field of machine learning to improve models' predictive capability. This approach is based on the idea of combining multiple weak classifiers, which individually offer limited performance, to obtain a model that is overall more robust and accurate.

The base models are aggregated according to specific strategies (i.e., "put into an ensemble") with the goal of reducing the overall error of the system.

Among the various ensemble learning approaches, one of the best known is **Random Forest**, which consists of an ensemble of independent decision trees trained in parallel on different portions of the dataset. The final predictions are combined through a majority vote mechanism in order to obtain a more stable and high-performing classifier.

Random Forest is based on the **bagging (bootstrap aggregating)** technique, which involves creating random subsets of the dataset (with replacement) for each tree.
Additionally, during the construction of each tree, **not all features are considered at each split**: for classification problems, the number of features randomly selected at each node is equal to **#used_features =** $\sqrt{\#features}$.

This strategy introduces additional diversity among the trees, helping to reduce overfitting and improve the model's generalization.

Bagging was also applied to the features (**bag of features**), meaning that each individual tree uses only a subset of features and data with the aim of increasing generalization; the subsets are randomly distributed, which is why the method is referred to as “Random Forests.”
Random Forest is therefore a multi-classifier trained using bagging techniques, with the individual classifiers being decision trees.

Random Forest was chosen as the binary classifier for several reasons:

- **Explainability:** A key advantage of using Random Forest classifier lies in its interpretability. Although the ensemble as a whole is complex, each individual decision tree is inherently transparent and traceable. This means that for any given prediction, it is possible to follow the decision path within each tree—identifying which features contributed and how. **This makes the model's output not only accurate but also understandable and justifiable to a human**.

- **Robustness and accuracy**: Random Forest is an ensemble of decision trees that reduces the risk of overfitting compared to individual decision trees. Thanks to its ability to "vote" across multiple trees, it can make more robust and accurate predictions.

- **Handling irrelevant features and missing data**: Random Forest is known for being resistant to the presence of irrelevant features, reducing their impact on model performance. Moreover, it can handle missing data more efficiently than other algorithms, for instance by ignoring or replacing them during tree construction.

- **Feature importance**: Random Forest provides a measure of the importance of each feature in the decision-making process, which can be useful for interpreting the model and analyzing the most relevant variables.

- **Generalization**: Since Random Forest uses bagging, which randomly samples data to build multiple trees, it is less prone to overfitting than a single decision tree. This makes it highly effective in generalizing to new data.

- **Parallel performance**: Each tree in a Random Forest can be built independently of the others, which allows parallel processing to speed up training.

3.2 Features extraction

Before proceeding with the training, validation, and testing phases of the classifier, the provided `.gz` archives were decompressed and organized into appropriate folders.
For each file in the training, validation, and test sets of Dataset 1, a feature vector was extracted using:

- the **Ember Feature Extractor**, for feature generation;

- the **LIEF library**, for parsing PE files.

For each subset, two feature vectors were saved: one for benign files and one for malware.
During the extraction process, some files could not be analyzed by LIEF due to corruption, obfuscation, or non-compliant formats. These cases were properly handled using a try-except block, which allowed exceptions raised during parsing to be caught, thereby avoiding process interruption and ensuring robustness and continuity in the extraction process.

After the extraction phase, the generated feature vectors were serialized in `.pkl` format and saved in the directory:
`/content/drive/Shared Drives/AI for Cybersecurity/Consegna/Dataset1`, within their respective training, validation, and test subfolders, in the files `benign_features.pkl` and `malware_features.pkl`, to be used in the classifier's training and testing process.

Discarded files were logged in a separate file (`bad_files.pkl`) for potential future analysis. However, due to time constraints, this analysis will not be part of the current project.
The same procedure was applied to Dataset 2, whose results were saved in the directory:
`/content/drive/Shared Drives/AIforCybersecurity/Consegna/Dataset2/general`.

The code for this phase is available in the notebook **Data_extractor.ipynb**.

3.3 First experiment

The following analysis concerns the notebook **RandomForestFinal.ipynb**.
To perform the necessary experiments, the working environment was prepared by loading from the drive all `.pkl` files containing the previously extracted features. For each file, the features were labeled with **1** for malware and **0** for benign files, using the **NumPy** library. The features and their corresponding labels were then aggregated, while still maintaining the distinction between the training, validation, and test sets.

To prevent potential errors during training, it was verified that there were no **NaN** (Not a Number) values, as these could compromise the correct functioning of the classifier. Any such values found were set to 0.
Below is the code related to the training set only:

```python
# Each malware file is labelled 1 and each benign file 0
training_malware_labels = list(np.ones(len(training_malware_features)))
training_benign_labels = list(np.zeros(len(training_benign_features)))

# Union of features and labels
x_train = np.array(training_malware_features + training_benign_features)
y_train = np.array(training_malware_labels + training_benign_labels)

# Replace NaN values at 0
x_train = np.nan_to_num(x_train, nan=0)
```

Afterward, the first experiment was carried out.

It was decided to train the machine learning model using the available training set and to perform the evaluation on the validation set.
To obtain the best combination of hyperparameters for the Random Forest, **Grid Search** was used, optimizing for **accuracy**.
Below is the dictionary with the chosen hyperparameters:

```python
# Hyperparameters grid
param_grid = {
       'n_estimators': [50, 100, 150],
       'max_depth': [10, 20, 50],
       'min_samples_leaf': [10, 20, 30],
       'max_features': ['sqrt']}
```

- **n_estimators**: Indicates how many trees to build in the forest. The more trees there are, the greater the robustness and stability of the model. A high number of trees increases training time but does not lead to overfitting.

- **max_depth**: Limits the maximum depth of each tree. The deeper a tree, the more complex it becomes, increasing the risk of overfitting. Reducing this parameter helps with generalization but may introduce bias by underestimating the system's complexity.

- **min_samples_leaf**: Ensures that each leaf (terminal node of the tree) contains at least *min_samples_leaf* samples. Increasing this value makes the tree less complex and more generalizable, while lowering it can lead to overfitting.

- **max_features**: The maximum number of features to consider when making a split at each node. `'sqrt'` means the square root of the total number of features will be used. This introduces diversity among the trees and helps reduce overfitting.

The combination of hyperparameters that produced the best result in terms of accuracy is as follows:

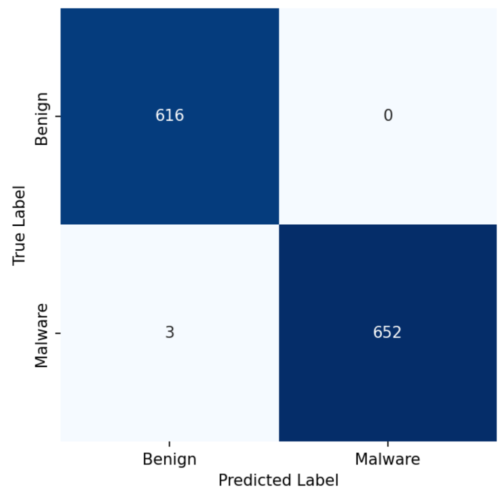
```python
Best parameters maximizing: Accuracy
{'max_depth': 50, 'max_features': 'sqrt', 'min_samples_leaf': 20,
'n_estimators': 50}
Accuracy: 1.000
```

It should be noted that the **accuracy is at its maximum**: the model correctly classified 100% of the validation set data. However, such accuracy is not always indicative of a high-quality model, as it could be the result of **overfitting**, especially if the validation set is small or not representative of the variability in the data.

For this reason, it is necessary to test the model on the **test set of Dataset 1**. However, even the results on the test set could appear excellent if it belongs to a distribution similar to that of the training set.

Therefore, the model will later be evaluated on **Dataset 2**, which is structurally separate from the first, in order to verify whether the model is truly capable of **generalizing effectively** to entirely new data.
To better understand the results on the test set, a **confusion matrix** was used—an extremely useful tool in classification problems that shows the rates of true positives, true negatives, false positives, and false negatives for each class under consideration.

```Python
Report on test set:
1268 exact previsions su 1271 samples
Accuracy: 0.997639653815893
Recall: 0.9969465648854962
F1 Score: 0.9977081741787625
Classification report
               precision    recall  f1-score   support
         0.0       1.00      1.00      1.00       616
         1.0       1.00      1.00      1.00       655
    accuracy                           1.00      1271
   macro avg       1.00      1.00      1.00      1271
weighted avg       1.00      1.00      1.00      1271
```



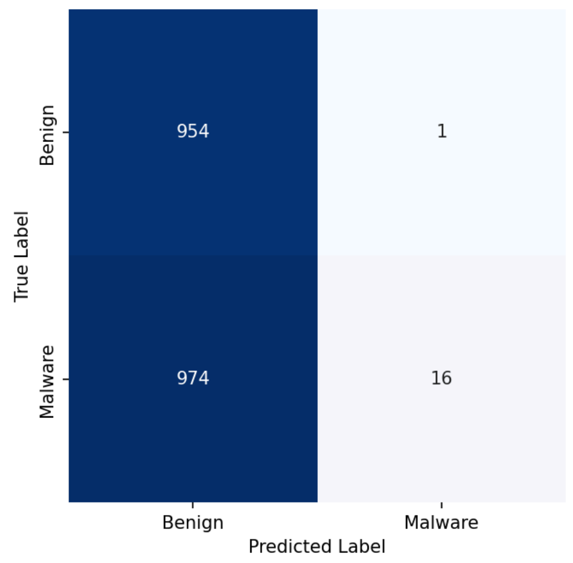
The result shows an **accuracy of approximately 100%**, which is the best that can be achieved.
 Specifically:

- **616** benign samples were correctly classified as benign (**true negatives**);
- **0** benign samples were incorrectly classified as malware (**false positives**);
- **3** malware samples were incorrectly classified as benign (**false negatives**);
- **652** malware samples were correctly classified as malware (**true positives**).

Di seguito invece, si osservi la matrice di confusione per il dataset 2:

```
Python
Report on dataset2:
970 exact previsions su 1945 samples
Accuracy of dataset2: 0.4987146529562982
Recall of dataset2: 0.01616161616161616
Classification report
               precision    recall  f1-score   support

         0.0       0.49      1.00      0.66       955
         1.0       0.94      0.02      0.03       990
    accuracy                           0.50      1945
   macro avg       0.72      0.51      0.35      1945
weighted avg       0.72      0.50      0.34      1945
```



Il risultato sul dataset 2 mostra un accuracy circa del 50% e in particolare:

- **954** campioni benigni sono stati classificati correttamente (**veri negativi**);
- **1** campione benigno è stato classificato erroneamente come malware (**falso positivo**);
- **974** campioni malware sono stati classificati erroneamente come benigni (**falsi negativi**);
- **16** campioni malware sono stati classificati correttamente (**veri positivi**).

Da come si evince, il modello fallisce completamente nel riconoscere i campioni malware sul dataset 2 riconoscendoli tutti come benigni: **scarsa capacità di generalizzazione**.
I risultati ottenuti confermano le aspettative, tuttavia è opportuno tornare a riflettere sull'elevato valore di **accuracy** osservato sul **validation set**. Un'accuratezza perfetta o quasi perfetta può apparire incoraggiante, ma solleva anche interrogativi sulla reale capacità di generalizzazione del modello.

Esistono due possibili spiegazioni per questo comportamento:

1. **Overfitting**: il modello potrebbe aver appreso troppo in dettaglio le caratteristiche del training set, perdendo la capacità di generalizzare su dati nuovi. Questo porta a prestazioni apparentemente ottime su dati simili a quelli di addestramento, ma

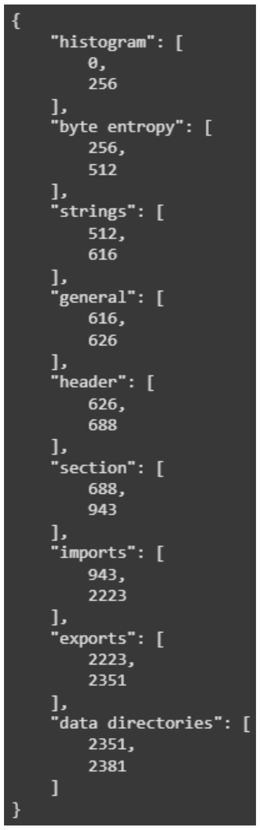
potenzialmente scarse su dati differenti.

2. **Bias nella classificazione causato dalle features**: alcune feature potrebbero seguire pattern presenti nel dataset ma non legati alla natura del problema reale o differenze marcate tra i valori delle feature nei file benigni e malware, che il modello sfrutta per separare le classi, ma che non sono causate da caratteristiche semantiche reali, bensì da artefatti nei dati.

3.4 Features Analysis

The decision was made to analyze the **features used to train the model**.
To better interpret the features, it is helpful to use the function `get_json_features(path)`, which returns the features in **JSON format** instead of a numerical list.

On the side, you can see the **Ember feature schema** for any PE file type, in particular:


```
{
    "histogram": [
        0,
        256
    ],
    "byte entropy": [
        256,
        512
    ],
    "strings": [
        512,
        616
    ],
    "general": [
        616,
        626
    ],
    "header": [
        626,
        688
    ],
    "section": [
        688,
        943
    ],
    "imports": [
        943,
        2223
    ],
    "exports": [
        2223,
        2351
    ],
    "data directories": [
        2351,
        2381
    ]
}
```

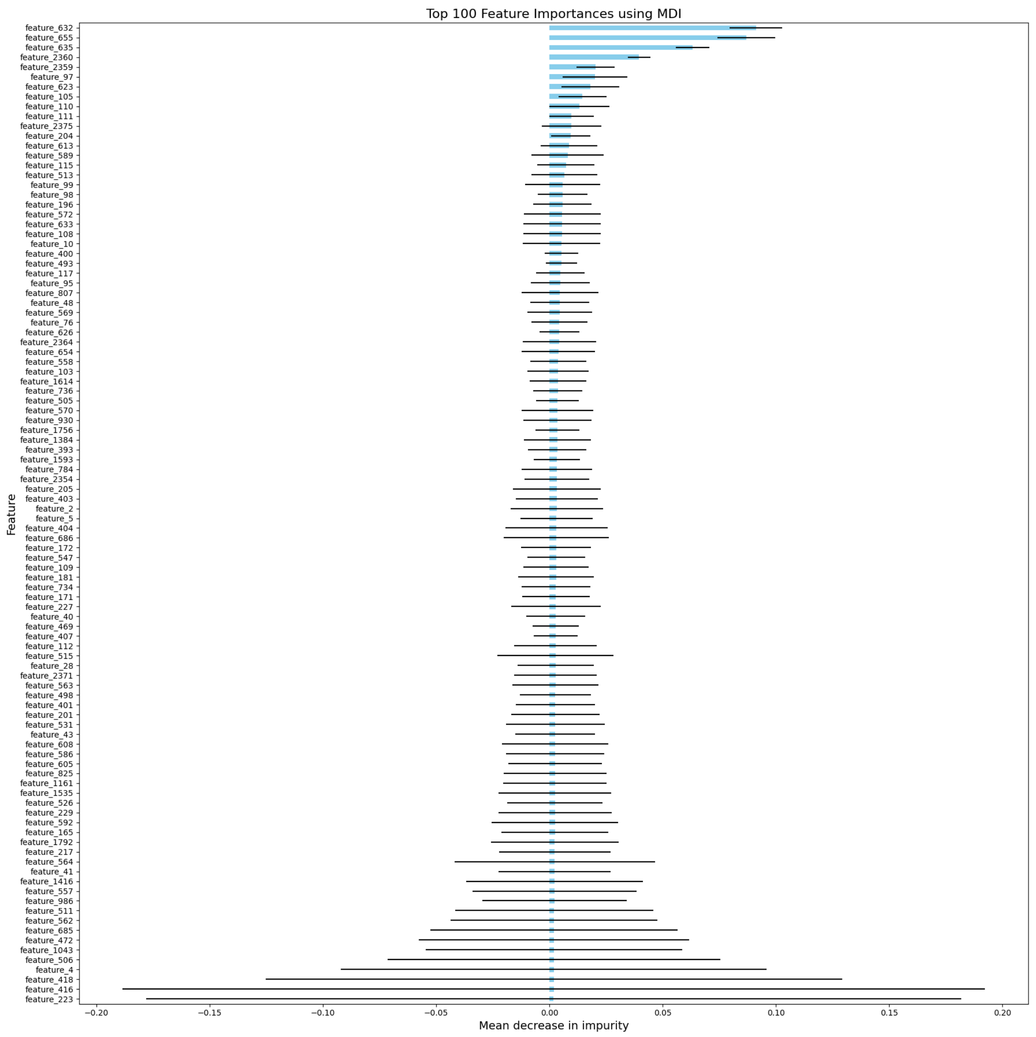
- **histogram**: represents the histogram of the file's bytes; it counts how many times each byte value (from 0 to 255) appears in the file;
- **byte entropy**: measures the entropy of byte blocks, i.e., an indication of data randomness. Encrypted or compressed code will have higher entropy, as malware often obfuscates or encrypts parts of its code;
- **strings**: counts and analyzes the ASCII strings contained in the file, as malware may include suspicious or obfuscated strings;
- **general**: general file features such as size, number of sections, base addresses, number of imported/exported functions, resources, etc. These characteristics often differ between benign and malicious files;
- **header**: includes information extracted from the PE file header, such as machine type, entry point, timestamp, and other metadata;
- **section**: statistics about the PE file's sections (e.g., `.text`, `.data`, `.rsrc`). Malware often modifies or hides data within these sections;
- **imports**: list of external library functions (Dynamic Link Libraries - DLLs) imported by the file;
- **exports**: list of functions the file exports—i.e., makes available to other programs (executables);
- **data directories**: specifies pointers to PE structures (e.g., import/export table, resources, relocations), indicating the internal structure and potentially revealing anomalies.

Random Forest assigns an **importance** score to each feature used during training, based on the feature's contribution to the decision-making process. This importance is represented by a numerical value ranging from 0 to 1—higher values indicate greater importance.

To better understand the model’s behavior, these values were sorted in descending order, and only the top 100 most important features were displayed in the output, calculated using the **Mean Decrease in Impurity (MDI)** technique.

Below is the image in which the following components can be observed:

- **Y-axis**: features (the dataset’s columns);
- **X-axis**: feature importance value, calculated as the *mean decrease in impurity*, which indicates how much that feature contributes, on average, to improving the purity of the nodes in the trees. In other words, it shows how discriminative the feature is for the model;
- **Horizontal blue bars**: represent the **importance** of each feature for the model—the longer the bar, the greater the importance;
- **Error bars (black lines)**: show the **variability** (standard deviation) of the importance values across all the trees in the forest.



From the image, it can be seen that the top features (**feature_632**, **feature_655**, and **feature_635**) contribute the most to the model's decisions, while the features at the bottom (**feature_416**, **feature_223**) have very low or zero importance and may be irrelevant—or even introduce noise into the model.

The analysis then continues with the most discriminative features identified in the previous graph: **632, 655, 635, 2360, 2359, 97, 623, 105, 110**, by examining the first 10 malware samples and the first 10 benign files from the training set of **Dataset 1**.

The first step was to verify the **category** to which each feature belongs:

```Python
Rank 1: Feature 632 - Importance: 0.0912 - Category: header
Rank 2: Feature 655 - Importance: 0.0870 - Category: header
Rank 3: Feature 635 - Importance: 0.0631 - Category: header
Rank 4: Feature 2360 - Importance: 0.0396 - Category: data directories
Rank 5: Feature 2359 - Importance: 0.0203 - Category: data directories
Rank 6: Feature 97 - Importance: 0.0201 - Category: histogram
Rank 7: Feature 623 - Importance: 0.0181 - Category: general
Rank 8: Feature 105 - Importance: 0.0146 - Category: histogram
Rank 9: Feature 110 - Importance: 0.0132 - Category: histogram
```

To verify whether the features deemed most important might introduce **bias** into the classification, an exploratory analysis was conducted on the first 10 malware files and the first 10 benign files from the dataset.
For each of these files, the **actual values** of the selected features were printed, with the goal of identifying any **marked differences** or **recurring patterns** that could unduly favor the separation between classes.

```Python
# Valori delle feature da analizzare
feature_indices = [632, 655, 635, 2360, 2359, 97, 623, 105, 110]
...
# Stampa dei valori per i primi 10 malware
print("MALWARE FEATURES:\n")
for idx in range(10):
    values = [training_malware_features[idx][i] for i in feature_indices]
    formatted_values = [format_value(v) for v in values]
    print(f"Malware {idx + 1}: [{', '.join(formatted_values)}]")

# Stampa dei valori per i primi 10 benigni
print("\nBENIGN FEATURES:\n")
for idx in range(10):
    values = [training_benign_features[idx][i] for i in feature_indices]
    formatted_values = [format_value(v) for v in values]
    print(f"Benigno {idx + 1}: [{', '.join(formatted_values)}]")
```

Running the code above gives the following output:

```Python
MALWARE FEATURES:
Malware 1: [0.0, 1.0, 1.0, 0.0, 0.0, 0.00273, 0.0, 0.00248, 0.00264]
Malware 2: [0.0, 1.0, 1.0, 0.0, 0.0, 0.00294, 0.0, 0.00252, 0.00298]
Malware 3: [0.0, 1.0, 1.0, 0.0, 0.0, 0.00559, 0.0, 0.00407, 0.00353]
Malware 4: [0.0, 1.0, 1.0, 0.0, 0.0, 0.00183, 0.0, 0.00170, 0.00165]
Malware 5: [0.0, 1.0, 1.0, 0.0, 0.0, 0.00394, 0.0, 0.00391, 0.00380]
Malware 6: [0.0, 1.0, 1.0, 0.0, 0.0, 0.00114, 0.0, 0.00117, 0.00093]
Malware 7: [0.0, 1.0, 1.0, 0.0, 0.0, 0.00294, 0.0, 0.00252, 0.00299]
Malware 8: [0.0, 1.0, 1.0, 0.0, 0.0, 0.00182, 0.0, 0.00160, 0.00182]
Malware 9: [0.0, 1.0, 1.0, 0.0, 0.0, 0.00114, 0.0, 0.00117, 0.00093]
Malware 10: [0.0, 1.0, 1.0, 0.0, 0.0, 0.00302, 0.0, 0.00237, 0.00439]

BENIGN FEATURES:
Benigno 1: [1.0, -1.0, 0.0, 140784.0, 6688.0, 0.01589, 1.0, 0.00936, 0.00897]
Benigno 2: [1.0, -1.0, 0.0, 0.0, 0.0, 0.00596, 0.0, 0.00462, 0.00514]
Benigno 3: [1.0, 0.0, 0.0, 169472.0, 6328.0, 0.00461, 1.0, 0.00516, 0.00480]
Benigno 4: [1.0, -1.0, 0.0, 0.0, 0.0, 0.00395, 0.0, 0.00408, 0.00397]
Benigno 5: [0.0, 0.0, 0.0, 0.0, 0.0, 0.00335, 0.0, 0.00287, 0.00264]
Benigno 6: [1.0, 0.0, 0.0, 72704.0, 5960.0, 0.01930, 1.0, 0.01672, 0.01805]
Benigno 7: [1.0, -1.0, 0.0, 0.0, 0.0, 0.00432, 0.0, 0.00423, 0.00434]
Benigno 8: [0.0, -1.0, 0.0, 0.0, 0.0, 0.00295, 0.0, 0.00332, 0.00365]
Benigno 9: [1.0, -1.0, 0.0, 0.0, 0.0, 0.00662, 0.0, 0.00606, 0.00670]
Benigno 10: [1.0, -1.0, 0.0, 3581192.0, 5744.0, 0.00394, 1.0, 0.00397, 0.00395]
```

From the results above, it can be observed that the analyzed features **allow us to determine whether a file is malicious or benign**. This holds true only for the **first five features** and the **seventh**, but not for the remaining ones, as they show **different values across all malware and benign files**, making it impossible to perform a clear classification based on them.
To support this hypothesis, the same analysis was carried out on **Dataset 2**, and the results are shown below:

```Python
MALWARE FEATURES:
Malware 1: [1.0, 0.0, 0.0, 1884160.0, 7432.0, 0.00845, 1.0, 0.00815, 0.00770]
Malware 2: [1.0, 0.0, 0.0, 24576.0, 9224.0, 0.00511, 1.0, 0.00544, 0.00503]
Malware 3: [1.0, -1.0, 0.0, 0.0, 0.0, 0.00351, 0.0, 0.00358, 0.00355]
Malware 4: [1.0, 0.0, 0.0, 0.0, 0.0, 0.00386, 0.0, 0.00410, 0.00370]
Malware 5: [1.0, -1.0, 0.0, 0.0, 0.0, 0.00288, 0.0, 0.00248, 0.00277]
Malware 6: [1.0, -1.0, 0.0, 0.0, 0.0, 0.00367, 0.0, 0.00368, 0.00365]
Malware 7: [1.0, 0.0, 0.0, 0.0, 0.0, 0.00412, 0.0, 0.00392, 0.00386]
Malware 8: [1.0, -1.0, 0.0, 0.0, 0.0, 0.00393, 0.0, 0.00390, 0.00392]
Malware 9: [1.0, -1.0, 0.0, 0.0, 0.0, 0.00616, 0.0, 0.00814, 0.00707]
Malware 10: [1.0, -1.0, 0.0, 56832.0, 14240.0, 0.00519, 1.0, 0.00712, 0.00626]

BENIGN FEATURES:
Benigno 1: [0.0, -1.0, 0.0, 135168.0, 9976.0, 0.00163, 1.0, 0.00216, 0.00162]
Benigno 2: [1.0, 0.0, 0.0, 0.0, 0.0, 0.02837, 0.0, 0.03301, 0.03747]
Benigno 3: [1.0, 0.0, 0.0, 0.0, 0.0, 0.01223, 0.0, 0.01003, 0.00966]
Benigno 4: [1.0, 0.0, 0.0, 0.0, 0.0, 0.00449, 0.0, 0.00414, 0.00429]
Benigno 5: [1.0, -1.0, 0.0, 0.0, 0.0, 0.00374, 0.0, 0.00397, 0.00400]
Benigno 6: [1.0, -1.0, 0.0, 0.0, 0.0, 0.00403, 0.0, 0.00342, 0.00324]
Benigno 7: [0.0, 0.0, 0.0, 0.0, 0.0, 0.01095, 0.0, 0.01125, 0.01129]
Benigno 8: [1.0, -1.0, 0.0, 0.0, 0.0, 0.00157, 0.0, 0.00247, 0.00139]
Benigno 9: [0.0, -1.0, 0.0, 55296.0, 12056.0, 0.00802, 1.0, 0.00974, 0.00944]
Benigno 10: [1.0, 0.0, 0.0, 0.0, 0.0, 0.01142, 0.0, 0.01125, 0.01225]
```

From the output analysis, it is observed that some of the features considered highly discriminative by the Random Forest model do **not actually behave as such across all distributions**. On the contrary, they **introduce bias into the model**, enabling an **artificial separation** between classes based on **non-generalizable characteristics**.

These features could potentially be **removed**, followed by **retraining the Random Forest model**, in order to assess whether this modification leads to an **improvement in the classifier's generalization ability**, particularly on the **test set** and on **Dataset 2**.

However, before proceeding with their removal, it is important to **analyze each suspicious feature in detail**, in order to understand which ones truly have a **distorting effect**, and which may still provide **valuable contributions** to the classification task.

3.4.1 Features 632, 635, 655 (header - coff machine & optional subsystem)

The first features analyzed belong to the **header**.
Although only **features 636, 635, and 655** appear to be truly discriminative, it is still useful to **print all the header features** starting from their respective initial index, since the other features may still have a **negative impact**, for example by introducing **noise**.

```
Python
#It obtains the initial index of features belonging to the “header” category
(among all features in the EMBER vector).
starting_value_header = extractor.get_feature_range()['header'][0]
#Extracts the index for the PE header-specific features contained in that file.
rv = extractor.get_header_features_index(bytes)
print("HEADER")
for key in rv:
  if type(rv[key]) == tuple: #Some features cover a range of indices (e.g.
coff-characteristics: (11, 21)), so they are tuples (start, end).
      print(f'{key}: {rv[key][0] + starting_value_header} - {rv[key][1] +
starting_value_header}')
  else:
    print(f'{key}: {rv[key] + starting_value_header}')
```

With the following output

```
Python
HEADER
coff-timestamp: 626
coff-machine: 627 - 637
coff-characteristics: 637 - 647
optional-subsystem: 647 - 657
optional-dll_characteristics: 657 - 667
optional-magic: 667 - 677
optional-major_image_version: 677
optional-minor_image_version: 678
optional-major_linker_version: 679
optional-minor_linker_version: 680
optional-major_operating_system_version: 681
optional-minor_operating_system_version: 682
optional-major_subsystem_version: 683
optional-minor_subsystem_version: 684
optional-sizeof_code: 685
optional-sizeof_headers: 686
optional-sizeof_heap_commit: 687
```

From the output above, we can infer that **features 632 and 635** are part of **coff-machine**, which specifies the type of processor architecture the file was compiled for, while **feature 655** belongs to **optional-subsystem**, which indicates the runtime environment required by the file.

For this reason, we will later consider the **range to which these header features belong** in order to proceed with their removal.

Removing this type of feature is beneficial for improving **generalization on Dataset 2**, since **coff-machine** and **optional-subsystem** do not describe the **behavior** of the file but rather **compilation metadata**, and are often **strongly correlated with the source dataset**.

This is also confirmed by the following analysis, conducted on just 5 malware and benign files (for benign files, replace `malware_dir` with `benign_dir`):

```python
for file_name in os.listdir(malware_dir):
  file_path = os.path.join(malware_dir, file_name)
  with open(file_path, 'rb') as file:
      raw, _ = extractor.get_raw_and_processed_features(file.read(), 'header')
  print(f"File ({category}): {file_name}")
  for key_path in features_to_extract:
      try:
          parts = key_path.split('.')
          value = raw
          for part in parts:
              value = value[part]
          print(f"\t{key_path}: {value}")
      except KeyError:
          print(f"\t{key_path}: [KEY NOT FOUND]")
  print()
```

Which produces the following outputs for malware and benign files:

```python
File (malware):33ca40e533b01144252c93798839bb5519f5030c99ad93d954ad9c8e256f09d9
       coff.machine: UNKNOWN
       optional.subsystem: UNKNOWN
File (malware):9ce8cddd53eb41425163a5daf21bbb62d3df83287ad99e6f33e1cdb0a883188e
       coff.machine: UNKNOWN
       optional.subsystem: UNKNOWN
File (malware):631bc90de79ac0a65812a6fa4889bf03185ee1f92370db86a22f63bbb875b0e4
       coff.machine: UNKNOWN
       optional.subsystem: UNKNOWN
```

```
File (malware):8da703a70be8e5ef4ab624d1dd6f0a69b16ffcbb853011fbdb3ebbf5c937efdc
        coff.machine: UNKNOWN
        optional.subsystem: UNKNOWN
File (malware):393304fde3e5f36cfd3c3c56fd9fb8515088de3b45e1d0d6133e8c4f21ea2419
        coff.machine: UNKNOWN
        optional.subsystem: UNKNOWN
```

```Python
File (benign): b039bc8a626729101420967f70f6ef8469b47b3a1357f14271b44a086338bf9e
        coff.machine: AMD64
        optional.subsystem: WINDOWS_CUI
File (benign): b01afda35fe7ae222b1a5cb58d7a061c566904e6d53858b7f9ae76e3cd73dc03
        coff.machine: AMD64
        optional.subsystem: WINDOWS_CUI
File (benign): f4bfec3ba96c1d1c8fc6d646f9a13872d4cb14b1fbd9fb3dbd37379160e83287
        coff.machine: AMD64
        optional.subsystem: WINDOWS_CUI
File (benign): 1a0b917d701bae3277ba654f602976dd7720d5f993a54f3c2efc073a6a44a92d
        coff.machine: I386
        optional.subsystem: WINDOWS_GUI
File (benign): 0f089f8a29745b24daa48139e713df7b0e2290a4ff03a598d639927166451327
        coff.machine: AMD64
        optional.subsystem: WINDOWS_CUI
```

From these two outputs, it can be observed that **benign files show varying values** for this feature, while **malware files consistently have the value "UNKNOWN"**, which is a typical behavior of malware to obfuscate the file.

From this, we can conclude that these features are **highly discriminative** but **not generalizable**, contributing to an **illusion of accuracy** during training and validation.
By removing them, the model will focus more on **behavioral features**, which could significantly improve its ability to **generalize** to unseen data.

3.4.2 Features 2359, 2360 (data directories - exception table)

Next, the same analysis was performed on **features 2360 and 2359**, which, as previously mentioned, belong to **data directories**.
The following code is used to print all features within that range:

```
Python
HEADER
rv={}
# Print feature range
print(extractor.get_feature_range())

# Gets the initial value for features in the ‘Data directories’ category
starting_value = extractor.get_feature_range()['data directories'][0]

# Extract the raw features and retrieve the keys
rv = extractor.raw_features(bytes)['datadirectories']
for r in rv:
    rvKeys = r.keys()
    # Iterate on features to calculate their range
     for key in rvKeys:
     if isinstance(r[key], list):
        print(f'{key}: {r[key]} {starting_value} - {len(rv[key]) +
starting_value}')
        starting_value += len(r[key])
     else:
        print(f'{key}: {r[key]} {starting_value}')
        starting_value += 1
```

The output is as follows

```
Python
name: TYPES.EXPORT_TABLE 2351
size: 0 2352
virtual_address: 0 2353
name: TYPES.IMPORT_TABLE 2354
size: 280 2355
virtual_address: 461652 2356
name: TYPES.RESOURCE_TABLE 2357
size: 36864 2358
virtual_address: 577536 2359
name: TYPES.EXCEPTION_TABLE 2360
size: 0 2361
```

```
virtual_address: 0 2362
name: TYPES.CERTIFICATE_TABLE 2363
size: 0 2364
...
```

The **data directories range** (from feature **2351 to 2398**) contains pointers and sizes of PE tables such as the **EXPORT_TABLE** (where exported functions are defined), **IMPORT_TABLE** (modules and functions imported), **RESOURCE_TABLE**, **EXCEPTION_TABLE**, etc.
Each table entry has two elements: the **Virtual Address** (where the structure is located in the file) and the **Size** (how much space it occupies). In the case of the features being analyzed, they represent:

- **2359**: Virtual address of the **RESOURCE_TABLE**
- **2360**: Size of the **EXCEPTION_TABLE**

An **EXCEPTION_TABLE** is a structure used in 64-bit executable files in the **.pdata** section that contains:

- Information on **runtime exception handling** (e.g., try/catch);
- **Function mapping and unwind routines**, related to the **stack unwinding** process—i.e., the process Windows uses to clean up the stack after an exception.

By itself, **feature 2359** is not highly discriminative, but it may have a **negative contribution** when used alongside its size counterpart—**feature 2360**.

Specifically, if **feature 2360 = 0**, it indicates that the **EXCEPTION_TABLE does not exist**.
Conversely, **feature 2360** is much more informative: the presence of a well-defined **Exception Table** can indicate a **benign file**.

Many **malware samples either omit the EXCEPTION_TABLE** or leave it empty.
Indeed, if this feature's value is **zero (size = 0)**, the file **does not use structured exception handling**, which may indicate that the file is **malicious**. On the other hand, if **size > 0**, the file includes exception-handling routines and may be **benign**.

As evidence of this, a count of malware samples was performed based on this feature's value, using the following code:

```python
i,j = 0
print("Malware:")
for vector in x_train[:4598]:
    if vector[2360] != 0:
        i += 1
    else:
        j += 1
print("sum(0):", j)
print("sum(!=0):", i)
i,j = 0
print("\nBenigni:")
for vector in x_train[4598:]:
    if vector[2360] != 0:
        i += 1
    else:
        j += 1
print("sum(0):", j)
print("sum(!=0):", i)
```

Producing the following output:

```python
Malware:
sum(0): 4383
sum(!=0): 215

Benigni:
sum(0): 2100
sum(!=0): 2269
```

From this result, we can infer that **about 95% of malware samples do not have a valid EXCEPTION_TABLE**, while benign files show a **more balanced distribution**, indicating that many of them use structured exception handling mechanisms.

This leads the model to **interpret the absence of a valid EXCEPTION_TABLE as a strong indicator of malware presence**. However, in reality, this correlation is **not always reliable**, as seen in the **distribution among benign files**, where some do not include an EXCEPTION_TABLE for various reasons.

As a result, the model risks **overfitting to this specific characteristic** of files in **Dataset 1**, making it an overly **discriminative feature** and, therefore, a **candidate for removal**.

3.4.3 Feature 97, 105 e 110 (histogram)

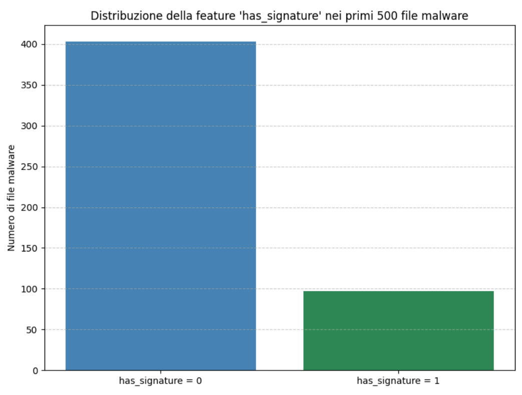
The **second-to-last feature (97)** and the **last two (105 and 110)**, as previously observed, belong to the **histogram** category.
These features will **not be removed**, as they are **not dataset-specific**. On the contrary, they reflect the **binary distribution of the file content**; they capture **intrinsic behavior** of the file that cannot be easily explained or interpreted by humans.
For this reason, the same type of analysis carried out on the other features **cannot be applied** to these.

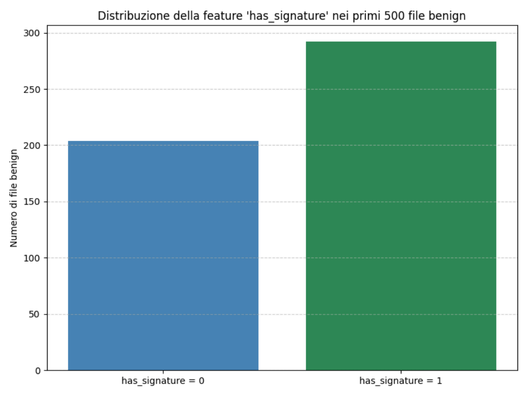
3.4.4 Feature 623 (general - has signature)

Feature 623 belongs to the **general** category: it contains general information about the file, such as size, number of sections, base addresses, number of imported and exported functions, resources, etc.In particular, from the output below, it can be seen that **feature 623 represents `has_signature`**, which indicates whether or not a **digital signature** is present within the file.

```python
size: 616
vsize: 617
has_debug: 618
exports: 619
imports: 620
has_relocations: 621
has_resources: 622
has_signature: 623
has_tls: 624
symbols: 625
```

This is mainly due to the fact that benign files, especially legitimate files distributed by trustworthy developers, tend more frequently to have a valid signature, whereas malware often lack one or report a missing or invalid value.





To evaluate the distribution of values for this feature, **not all malware and benign files were considered** due to time constraints. Instead, only the **first 500 malware samples** and the **first 500 benign samples** were used.
From the results obtained, as shown in the histogram on the left, **403 malware samples do not have a digital signature** (`has_signature = 0`), while **97 malware samples are signed** (`has_signature = 1`). This can be due to several reasons:

- **Avoiding traceability**: Digitally signing malware requires the use of a certificate that can be linked to a specific entity. By signing the malware, authors could leave behind traces that facilitate their identification.
- **Reducing cost and complexity**: Obtaining a valid signing certificate involves expenses and identity verification procedures.
- **Avoiding certificate revocation**: Even if malware is signed, once it is discovered, the certificate can be revoked by the relevant authorities, rendering the signature useless and potentially exposing the author to legal risks.
- **Exploiting the lack of strict controls**: Many operating systems and users do not enforce strict restrictions on the execution of unsigned files, allowing unsigned malware to run without significant obstacles.

For **benign files**, on the other hand, the situation is quite different: as shown in the histogram on the right, there is a **fairly balanced distribution** between the two classes (`has_signature = 0` and `has_signature = 1`).
The result obtained for the malware samples could be **misleading for the model**, as it may introduce a **bias related to the training set** of Dataset 1.
For this reason, it can be concluded that this is a **highly discriminative feature**, and it will therefore be **removed** from the dataset.

3.5 Removal of features and improvements

After a thorough analysis of the most discriminative features in the previous section, we now proceed to the **model optimization phase** for the **Random Forest** classifier. This phase involves **removing specific features** and implementing additional enhancements aimed at improving the model's **generalization capability**.
The **feature removal process** entails excluding from the dataset (which includes both malware and benign files) those features identified as being **overly dataset-specific**.
This step is critical for several reasons:

- It **reduces overfitting**, as overly influential features in the training set may lead the model to memorize the data. Removing them forces the model to learn more **robust and generalizable patterns**;
- It **enhances generalization**, enabling the model to perform better on real-world data, such as **Dataset 2**;
- It **improves overall performance**, both in terms of accuracy and stability.

The following code was used for the removal of features:

```python
# Indices to be removed (all inclusive)
ranges_to_remove = (
    list(range(626, 637)) +     # coff-machine (header)
    list(range(647, 657)) +     # optional-subsystem (header)
    list(range(623, 624)) +     # has signature
    list(range(2357, 2363))     # Data directories
)

def remove_features_from_dataset(features, feature_indices_to_remove):
    features_filtered = copy.deepcopy(features)  # Create a deep copy
    for feature_vector in features_filtered:
        for idx in feature_indices_to_remove:
            feature_vector[idx] = 0 # You set the features to 0
    return features_filtered

# Apply removal
x_train_filtered = remove_features_from_dataset(x_train, ranges_to_remove)
x_validation_filtered = remove_features_from_dataset(x_validation,
ranges_to_remove)
```

As can be observed, the variable `ranges_to_remove` contains the **indices of all the features selected for exclusion**. However, these features are **not actually removed** from the dataset—instead, their values are simply **set to zero**: they retain their position in the array, but their value is replaced with 0.
This approach avoids a critical issue: **index shifting**.
If the features were completely removed, the indices of the remaining features would change, leading to inconsistencies in any subsequent selection or removal operations.
By **zeroing them out**, the **structure of the dataset is preserved**, ensuring **consistency** throughout the various processing stages.
An additional improvement to the model involved **adjusting the parameter settings** for the Grid Search as follows:

```python
param_grid = {
    'n_estimators': [50, 100],
    'max_depth': [10, 20, 50],
    'min_samples_leaf': [20, 30],
    'max_features': ['sqrt']
```

- The number of estimators (**n_estimators**) was reduced because a high number of trees (such as 150) can significantly **increase training time**, especially when performing **multiple hyperparameter combinations**, as in this case.
- The **minimum number of samples per leaf** (**min_samples_leaf**) was increased (i.e., larger leaves), also to help **prevent overfitting**.

In addition, the **`class_weight = {0: w1, 1: w2}`** parameter was introduced to **balance class importance** during training.
What happens is that, during training, **each misclassification is weighted** according to the **weight of its class**—the higher the weight, the more costly it is for the model to make an error on that class.
The default value is `"balanced"`, but it can also be set **manually** as follows:

```python
'class_weight': ['balanced', {0:1, 1:5}, {0:1, 1:10}]
```

Each error made on a **class 0 sample** (benign) is given a **weight of 1**, while **malware samples (class 1)** are assigned a weight of **5 or 10**. This means that every error made on a class 1 sample is **5 or 10 times more severe** than an error on a benign sample. In other words, **misclassifying a malware sample is considered much more critical** than misclassifying a benign one, and the model is therefore more incentivized to correctly classify malware—even at the cost of increasing false positives.
The goal of this strategy is to **reduce false negatives**, i.e., cases where malware goes undetected—an outcome that is generally far more **critical and damaging** in real-world cybersecurity scenarios.

3.6 Final Experiment

After the **feature removal** and the enhancement with the **class weight** parameter, the model was retrained with a focus on a **combined score**—a custom metric that merges multiple standard metrics (such as **Recall** and **F1-score**) into a single numerical value to be optimized during model selection.Using a **combined score** is particularly effective in real-world problems like **malware detection**, since relying solely on:

- **Accuracy** is insufficient when the dataset is imbalanced;
- **F1-score** is helpful, but it doesn't indicate how many malware instances are being missed;
- **Recall (for the malware class, class 1)** is crucial, as in malware detection, a **false negative** is typically far more serious than a false positive.

Using the combined score allows prioritization of **recall on malware**, while maintaining a compromise with other objectives (such as F1-score).

For this reason, two parameters—**alpha** and **beta**—were introduced to assign specific weights to **Recall** and **F1-score**, and then combined to produce the following **combined_score**:

```python
combined_score = (alpha * recall_1) + (beta * f1) #alpha=0.7, beta=0.3
```

This is giving 70% and 30% relative importance between Recall and F1-score.

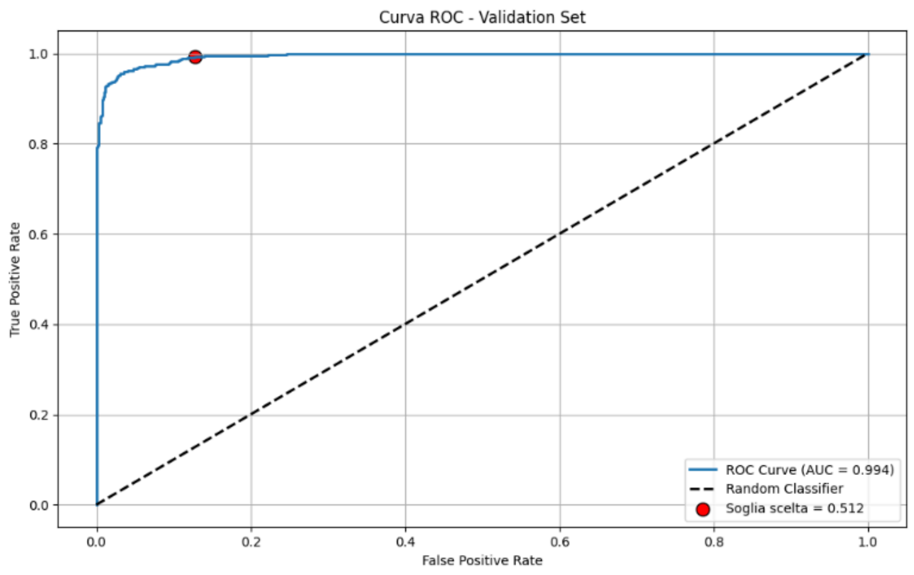
3.6.1 ROC Curve

For the evaluation, the **ROC curve (Receiver Operating Characteristics)** was also considered, which illustrates the model's behavior as the **classification threshold** varies. Specifically:

- The **x-axis** represents the **False Positive Rate (FPR)**—i.e., the proportion of benign files incorrectly classified as malware;
- The **y-axis** represents the **True Positive Rate (TPR)**—i.e., the proportion of malware correctly detected.

Each point on the ROC curve corresponds to a **different classification threshold**, representing the various trade-offs between **sensitivity (Recall)** and **specificity (1 - FPR)**.

The **threshold** is the value above which the model assigns a **positive label (malware)**. For probabilistic models, the default threshold is **0.5**, which is typically chosen to **maximize a specific metric**—in this case, the previously defined **combined_score**.



From the ROC curve image above, plotted on the **validation set**, several important characteristics can be observed regarding the quality of the **malware classifier**.
Among these, the **AUC (Area Under the Curve)** represents the area under the ROC curve and is one of the most commonly used metrics for evaluating the performance of a binary classifier. It measures how well the model can distinguish between the two classes, **regardless of the chosen threshold**.

In this case, the **AUC = 0.994**, meaning the model is **nearly perfect** at distinguishing malware from benign files. If a benign file and a malware sample are randomly selected, the model will assign a **higher score to the malware instance in 99.4% of cases**.
The chosen **threshold** is **0.512**, which is very close to the default value of **0.5**. This threshold represents the **best compromise between sensitivity (recall) and precision**, with the goal of prioritizing recall to avoid missing malware.
However, to obtain a **comprehensive view** of the model's performance, the next section—**Final Evaluation**—will also take into account **other metrics**, such as **recall** and **F1-score**.

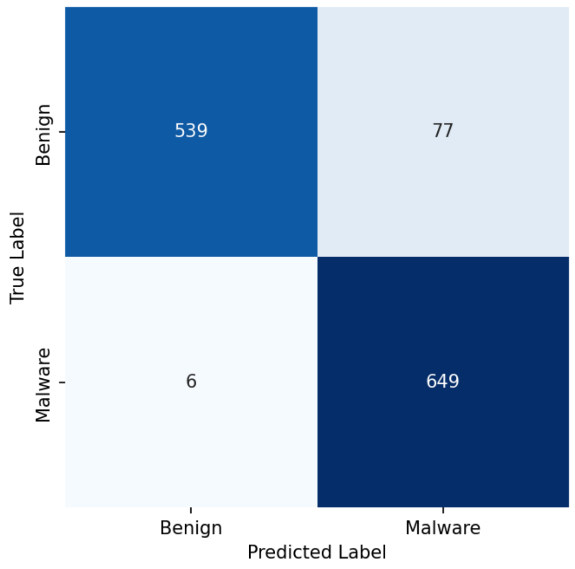
3.7 Final evaluations

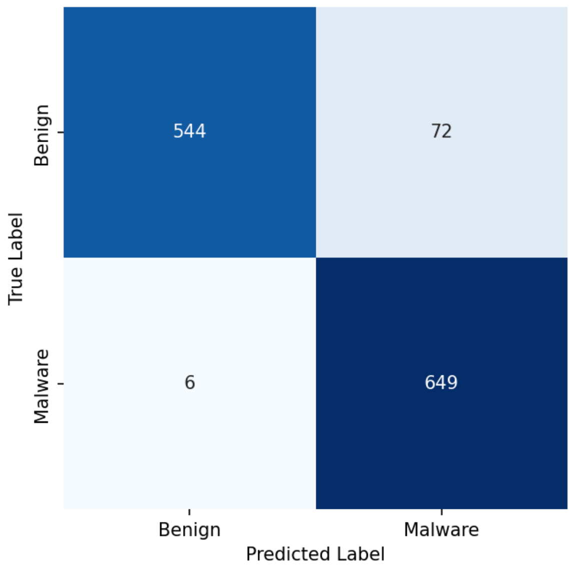
We will now proceed with the **final evaluations** on the **test set** and the **generalization set (Dataset 2)**.
As done earlier, the **confusion matrix** will be used for better **visualization and comparison** of the results obtained.

```
Python
Report sul test set:
Accuracy: 0.9346970889063729
Recall: 0.9908396946564886
F1 Score: 0.939898624185373
Classification report
              precision    recall  f1-score   support
         0.0       0.99      0.88      0.93       616
         1.0       0.89      0.99      0.94       655
    accuracy                           0.93      1271
   macro avg       0.94      0.93      0.93      1271
weighted avg       0.94      0.93      0.93      1271

Evaluation of test set (threshold 0.512):
Accuracy (0.512): 0.9386309992132179
Recall (0.512): 0.9908396946564886
F1 score (soglia 0.512): 0.9433139534883721
```

```
Classification report
              precision    recall  f1-score   support
         0.0       0.99      0.88      0.93       616
         1.0       0.90      0.99      0.94       655
    accuracy                           0.94      1271
   macro avg       0.94      0.94      0.94      1271
weighted avg       0.94      0.94      0.94      1271
```





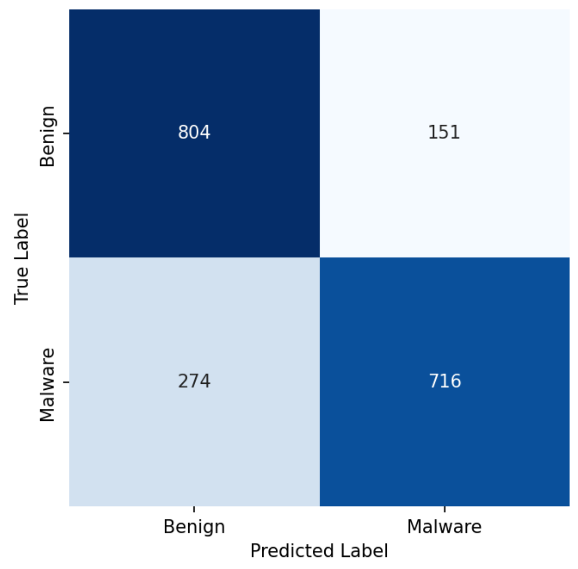
In particular, the **image on the left** represents the performance obtained on the **test set with the standard threshold (0.5)**, while the **image on the right** shows the performance with the **threshold selected from the ROC curve (0.512)**.
It can be observed that the results are **virtually identical**, precisely because the chosen threshold value is **very close to the standard threshold**. This confirms that the results described by the ROC curve represent the **best possible performance** achievable from a **Random Forest** trained with the previously defined parameters.
Compared to the initial performance, there is a **slight degradation**. This is entirely expected, as the **highly discriminative features** (such as *coff-machine*, *has_signature*, etc.)—which contained strong signals for distinguishing malware from benign files—were removed.
As a result, the model is now **more prone to confusion**, particularly with **benign files**. In fact, a **significant increase in false positives** can be observed, along with a **slight increase in false negatives**, though the latter is not particularly concerning.
All of this was done with the primary goal of **maximizing generalization** as much as possible.

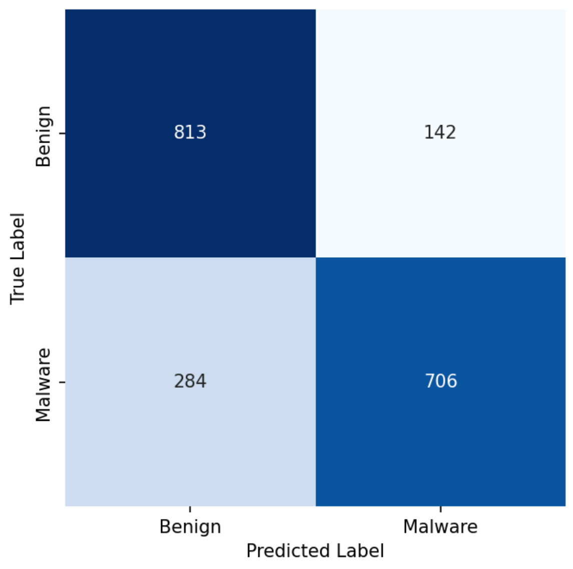
```
Python
Final evaluations on dataset2:
Accuracy on dataset2: 0.781491002570694
Recall on dataset2: 0.7232323232323232
F1 score on test set: 0.7711362412493269
Classification report
              precision    recall  f1-score   support

         0.0       0.75      0.84      0.79       955
         1.0       0.83      0.72      0.77       990
    accuracy                           0.78      1945
   macro avg       0.79      0.78      0.78      1945
weighted avg       0.79      0.78      0.78      1945

Final evaluations on dataset2 (soglia 0.512):
Accuracy on dataset2 (0.512): 0.7809768637532134
Recall on dataset2 (0.512): 0.7131313131313132
F1 score su dataset2 (soglia 0.512): 0.7682263329706203
Classification report
              precision    recall  f1-score   support

         0.0       0.74      0.85      0.79       955
         1.0       0.83      0.71      0.77       990
    accuracy                           0.78      1945
   macro avg       0.79      0.78      0.78      1945
weighted avg       0.79      0.78      0.78      1945
```





On the **left**, we see the results on the **generalization set with the standard threshold (0.5)**, while on the **right**, we see the results with the **threshold selected from the ROC curve (0.512)**. We observe that **false positives have decreased**, resulting in an **improvement in precision**, but **false negatives have increased**, meaning **recall has worsened**.
This outcome is **expected and justifiable**, since the ROC-based threshold was chosen using the **validation set**, and was therefore **optimal for its distribution**, not necessarily for the generalization set.
Nonetheless, **both results are significantly better** than those obtained initially:
Recall on malware has improved from **2% to 72%**, and **overall accuracy** has increased by approximately **23%**.

3.8 Conclusioni finali

The work carried out led to the development of a **Random Forest classifier** capable of detecting malware with excellent performance on data from the **same distribution as the training set**, but more importantly, with **significant improvements in generalization ability**.
This was achieved through a **careful removal of dataset-specific features** and **thorough hyperparameter optimization**.
In particular, it was observed that removing overly discriminative features (e.g., `coff-machine`, `has_signature`, `optional-subsystem`, `exception_table_size`) resulted in a **slight decrease in performance on the original test set**, but **prevented the model from learning artificial shortcuts**, thereby enabling **more robust behavior on external data**, as demonstrated by the results on **Dataset 2**.
The introduction of **targeted metrics**, such as the **combined score** between Recall and F1-score, allowed for **more balanced model selection**, prioritizing **malware detection** without excessively compromising **precision**.
The use of the **ROC curve** further enabled the calibration of the system to achieve **more controlled behavior** with respect to **false positives and false negatives**.
However, due to **computational limitations** imposed by the Colab environment and **time constraints**, it was not possible to:

- Explore **more complex machine learning models** (e.g., Gradient Boosting or XGBoost);
- Perform **automated threshold optimization** directly on the generalization dataset.

These limitations suggest **potential future improvements**, which could further enhance the system's robustness in real-world environments or against **advanced threats**.

Despite these constraints, the results demonstrate a **remarkable evolution** from the initial approach and lay **a solid foundation** for the application of **machine learning techniques in the cybersecurity domain**.

4 Deep Learning Model

Per affrontare il problema della malware detection da una prospettiva alternativa rispetto al machine learning classico, è stato sviluppato un classificatore basato su tecniche di Deep Learning.
A differenza del modello Ember, che si basa sull'estrazione manuale di feature, l'approccio adottato lavora direttamente sul contenuto binario dei file .exe, evitando qualunque pre-elaborazione semantica.

L'obiettivo è stato quello di sfruttare una rete neurale convoluzionale (CNN) ispirata a modelli reali utilizzati nel contesto antivirus, in grado di apprendere rappresentazioni significative dei dati senza la necessità di vedere feature estratte.
Il modello opera infatti direttamente sui raw bytes dei file eseguibili, trasformati in rappresentazioni binarie e normalizzate, ed è stato allenato a classificare ogni file come benigno o malware.

Il modello deep è stato sia addestrato che validato sul training set e validation set del Dataset 1, e infine testato sui test set di Dataset 1 e Dataset 2, con l'obiettivo di confrontare le performance e la capacità di generalizzazione.

4.1 Chosen model

Executable files (.exe) may contain complex patterns that are not easily detectable by shallow neural networks. For this reason, we chose the **AvastConv** architecture, which is based on a deep convolutional neural network (CNN). Thanks to its multiple—convolutional and fully connected layers—, if trained with a sufficient amount of data, it is capable of capturing highly sophisticated patterns that may appear in obfuscated or hashed malware files.

This architecture integrates memory management and computation optimization techniques defined in the **LowMemConvBase** class, whose logic will handle:

- **Chunking:**
  Each file is divided into fixed-size chunks to avoid loading the entire content into GPU memory at once. It is also possible to define an overlap between chunks to prevent loss of information at the boundaries:

```python
def __init__(self, chunk_size=65536, overlap=512, min_chunk_size=1024):
```

- **Processing the chunks:**
  Each chunk is processed separately by the convolutional network in an initial phase executed without gradient computation (i.e., in *no_grad* mode). In this phase, the model computes local activations (intermediate outputs of the convolutional layers) for each chunk, thereby identifying how "strong" the network's response is to each portion of the

file. This allows the importance of each segment to be evaluated without performing full backpropagation:

```python
def seq2fix():
      with torch.no_grad():
       activs = self.processRange(x_sub.long(), **pr_args)
```

- **Selecting informative segments:**
  After computing activations for all chunks, a global max pooling operation is applied to select the portions of the file that produced the highest activations across the different convolutional channels. These “winners” are considered the most informative segments of the file—those that have the greatest influence on the model’s final decision.

```python
def seq2fix():
      activ_win, activ_indx = F.max_pool1d(activs, kernel_size=activs.shape[2],
      return_indices=True)
```

- **Extracting the winning positions:**
  After completing inference on all chunks, the winning positions are extracted. For each file in the batch, the unique indices corresponding to these activation peaks are retrieved, representing the most salient points within the file:

```python
def seq2fix():
      final_indices = [np.unique(winner_indices[b,:]) for b in
      range(batch_size)]
```

- **Reconstructing the informative segments:**
  Once the most active positions have been identified, local segments around each of them are reconstructed. For every winning index, a sub-segment centered on it is extracted, with the same length as the receptive field of the convolutional network.
  This step makes it possible to isolate the regions of the file considered most relevant by the model, effectively creating a kind of 'zoom' focused exclusively on the most informative portions:

```
Python
def seq2fix():
      chunk_list
      =[[x[b:b+1,max(i-receptive_window,0):min(i+receptive_window,length)] for
      i in final_indices[b]] for b in range(batch_size)]
```

- **Preparing the final input:**
  All extracted segments are concatenated and unified for each file, generating a new compact sequence for every element in the batch. Since the segments may vary in length, automatic padding is applied to standardize the sequence lengths, ensuring compatibility with PyTorch's batch processing:

```
Python
def seq2fix():
      chunk_list = [torch.cat(c, dim=1)[0,:] for c in chunk_list]
      x_selected = torch.nn.utils.rnn.pad_sequence(chunk_list,
      batch_first=True)
```

- **Processing the final input with gradients enabled:**
  The informative segments, once unified and padded, are processed by the convolutional network (**self.processRange**) and compressed using adaptive pooling (**nn.AdaptiveMaxPool1d(1)**) to produce a compact, fixed-size vector for each file.
  Before processing, the data is transferred to the appropriate device (GPU or CPU) to ensure consistency during execution.
  Finally, the tensor is reshaped into the form (B, C) to be used in the subsequent layers (e.g., fully connected classification), where B represents the batch size and C is the number of output channels from the convolutional model:

```
Python
def seq2fix():
       if cur_device is not None:
             x_selected = x_selected.to(cur_device)
      x_selected = self.processRange(x_selected.long(), **pr_args)
      x_selected = self.pooling(x_selected)
      x_selected = x_selected.view(x_selected.size(0), -1)
      return x_selected
```

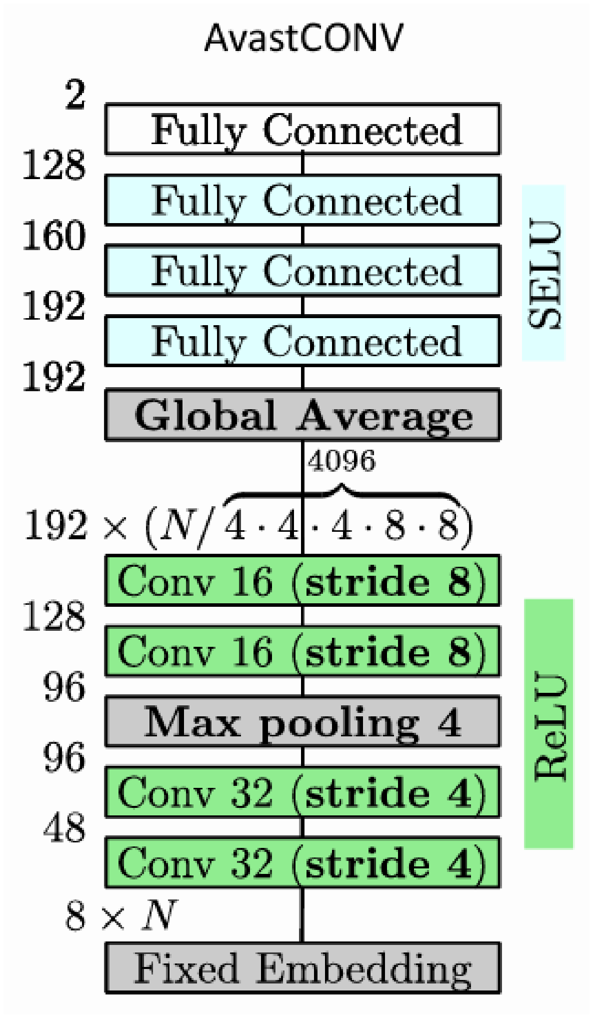
- **Zero-gradient cancellation:**
  During the backpropagation step (via the **loss.backward()** function), to further improve efficiency and avoid performing backpropagation for gradients equal to zero, a custom hook is attached to the cat function during initialization. This hook removes these unnecessary gradients:

```
Python
def __init__():
        self.cat = CatMod()
        self.cat.register_backward_hook(drop_zeros_hook)
```

  A hook in PyTorch is a function that can be attached to a module or tensor to execute custom code during the forward or backward pass of a neural network's training. In this case, it is a backward hook, meaning it is executed during backpropagation, and behaves as follows:
    - During backpropagation, PyTorch computes gradients (**grad_input**) for each input to the module.
    - This function iterates over grad_input and checks whether a gradient is entirely zero.
    - If so, it converts it to sparse format (**to_sparse()**), which is more memory- and computation-efficient (equivalent to a **NaN** in effect).
    - If the gradient contains non-zero values, it is left unchanged..

The compact sequence **x_selected**, built in the previous phase, is processed again by **AvastConv**, this time with gradient computation enabled. In this phase, the actual weight updates of the model occur during training.
The key innovation lies in the fact that backpropagation is performed only on the informative segments, avoiding the processing and propagation of gradients through irrelevant data. This approach significantly reduces memory usage and computational time while maintaining high learning effectiveness.

The standard **AvastConv** architecture used is composed of:



- **Binary embedding:** A non-trainable binary embedding layer, meaning it is not updated during training, is used to convert each byte of the file into a normalized binary representation between -1 and +1 using the `vec_bin_array` function. It transforms each byte into an 8-bit vector;
- **Convolutional layers:** The binary input transformed by the embedding is processed by four 1D convolutional layers with increasing depth, each followed by ReLU activation functions. These layers are designed to extract local patterns from the binary sequence and progressively reduce the sequence's dimensionality.;
- **Pooling layer:** A max pooling layer, applied after the initial convolutions, allows for further compression of the sequence, improving efficiency and helping the model focus on informative regions..

```python
self.conv_1 = nn.Conv1d(8, channels, window_size, stride=stride, bias=True)
self.conv_2 = nn.Conv1d(channels, channels * 2, window_size, stride=stride,
bias=True)
self.pool = nn.MaxPool1d(4)
self.conv_3 = nn.Conv1d(channels * 2, channels * 3, window_size // 2,
stride=stride * 2, bias=True)
self.conv_4 = nn.Conv1d(channels * 3, channels * 4, window_size // 2,
stride=stride * 2, bias=True)
```

- **Fully Connected Layers:** At the end of the convolutional phase, the network produces a compact tensor that is passed through four fully connected layers, each activated by a SELU function, chosen for its stability in deep models.

  The **SELU** (Scaled Exponential Linear Unit) activation function is a variant of the ReLU and ELU functions.

  $$\mathrm{SELU}(x) = \lambda \cdot \begin{cases} x & \text{se } x > 0 \\ \alpha(e^x - 1) & \text{se } x \leq 0 \end{cases}$$

  Where:

  - α ≈ 1.67326
  - λ ≈ 1.05070

  If only the ReLU function were used, the derivative would be zero for negative values, causing the neuron to stop updating. In contrast, with SELU, the derivative is never zero—even for negative values—which is important because the neuron remains 'active' at all times. The negative exponential part, with its increasing derivative, helps maintain a mean of 0 and a variance of 1, enabling self-normalization. For this reason, the AvastConv model does not include BatchNormalization layers.

```python
x = F.selu(self.fc_1(x))
x = F.selu(self.fc_2(x))
penult = x = F.selu(self.fc_3(x))
x = self.fc_4(x)
```

- **Final output:** A sigmoid function is used to obtain a binary probability (malicious or benign):

```python
return torch.sigmoid(x)
```

4.2 AvastConv model training

To train the model, the training set from Dataset 1—composed of benign and malicious executables—was used. The network was validated on a separate validation set from the same dataset, and subsequently tested on two distinct sets: the test set of Dataset 1 and Dataset 2. The latter was specifically used to assess the model’s generalization capability.

The training was conducted in a supervised manner, using:

- The Adam optimizer, particularly well-suited for models with many parameters and non-stationary gradients;
- The Binary Cross Entropy Loss (BCELoss) function, appropriate for binary classification.

The **train()** function manages the entire training loop of the model. In each epoch, it first performs training on the training set (with weight updates), followed by evaluation on the validation set (without updates). It automatically adjusts the learning rate if performance does not improve.
To prevent overfitting, an early stopping strategy is implemented through the **EarlyStopMonitor** class, which halts training if the validation loss does not improve for a predefined number of consecutive epochs (patience). If the validation loss improves, the model's best weights are saved.
Additionally, a learning rate scheduler (**ReduceLROnPlateau**) was used to adjust the learning rate during the training process. It is invoked directly within the **train** function.

```python
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, factor=0.5, patience=patience)
```

To handle the binary files used in the supervised training process, a custom dataset was developed, as the available data is not in a standard format commonly supported by deep learning libraries. Specifically, a **BinaryDataset** class was defined to explicitly specify how to read **.exe** files from disk and prepare them for processing by the model. Each file is read up to a maximum of 1 MB and converted into a byte sequence, which is then transformed into numerical tensors compatible with the neural network. File labeling is performed automatically based on the file path: if the file is located in a directory whose name contains the string 'malware', it is labeled as malicious (1); otherwise, it is labeled as benign (0).

Given the variability in file lengths, a custom function (**pad_collate_func()**) was also implemented and used by PyTorch's **DataLoader** to construct uniform batches. This function dynamically applies a padding mechanism, extending each sequence to match the length of the largest file in the batch. This ensures correct and efficient parallel processing.

```
Python
MAX_LEN = 2 ** 20  # 1 MB
train_dataset = BinaryDataset(path_list=x_train, max_len=MAX_LEN,
sorel_20m=True)
train_loader = DataLoader(train_dataset, batch_size=32,
collate_fn=pad_collate_func, shuffle=True)
```

This solution ensures that **.exe** files of varying sizes can be processed together within the same batch, while maintaining computational efficiency and structural correctness of the input.

The following parameters were chosen for training the model:

```
Python
model = AvastConv(out_size=1, channels=32, window_size=32, stride=4,
embd_size=8)
model = model.to(device)
criterion = nn.BCELoss()
train(model=model,
      criterion=criterion,
      val_loader=val_loader,
      train_loader=train_loader,
      device=device,
      patience=3,
      num_epochs=15,
      verbose=True,
      save_title='model_name',
      checkpoint_path='path-to-checkpoint-folder')
```

- **32 Channels.**
    - It should be noted that due to limited computational resources, the number of channels was reduced from 48 in the standard AvastConv architecture to 32.
- **A receptive window of 32 bytes with stride of 4.**
- **An 8-bit binary embedding for each byte of the file.**
- **15 epochs**.
    - Note that an early stopping mechanism was enabled with `patience=3`, which stops the training if the validation loss does not improve for three consecutive epochs.

4.3 Evaluation of the AvastConv Model

The model is tested and evaluated using two main functions: **predict()** and **get_accuracy()**. The **predict** function is responsible for computing the model's predictions over an entire dataset by iterating through batches provided by the DataLoader. During this phase, the model is set to evaluation mode (no weight updates), and for each batch, both the true labels and the predicted outputs are stored. Depending on whether the task is binary or multiclass classification, predictions are handled differently: in the multiclass case, the class with the highest probability is selected, while in the binary case, predictions are either rounded or passed through a sigmoid function to obtain probability scores, if requested.

```python
def predict(model, data_loader, device, criterion, apply_sigmoid=False,
to_numpy=True, multiclass=True)
```

The **get_accuracy** function leverages **predict** to obtain the model's predictions and compute the average loss over the dataset. It then compares the predictions with the true labels and returns the model's accuracy as a percentage, along with the true and predicted labels.

```python
def get_accuracy(model, data_loader, device, criterion, multiclass = True)
```

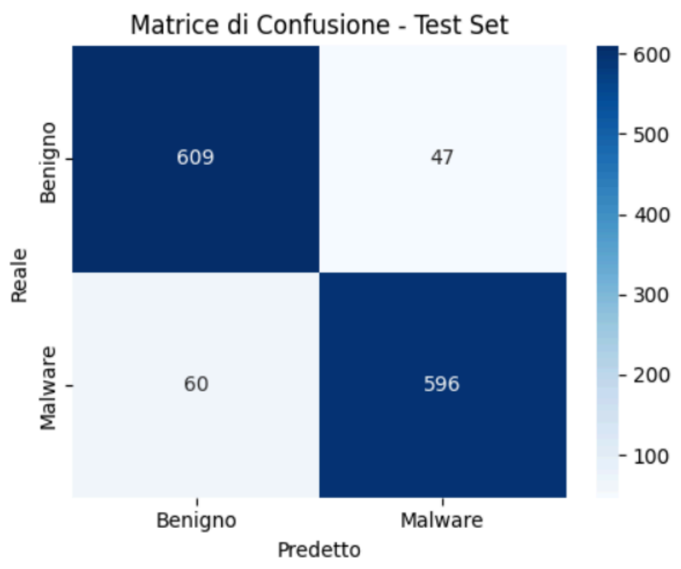
4.3 Results of the First Experiment

This experiment was evaluated using the confusion matrix computed on the available datasets.

The performance evaluation is carried out on the test set:

```python
              precision    recall  f1-score   support

     Benigno       0.91      0.93      0.92       656
     Malware       0.93      0.91      0.92       656
```

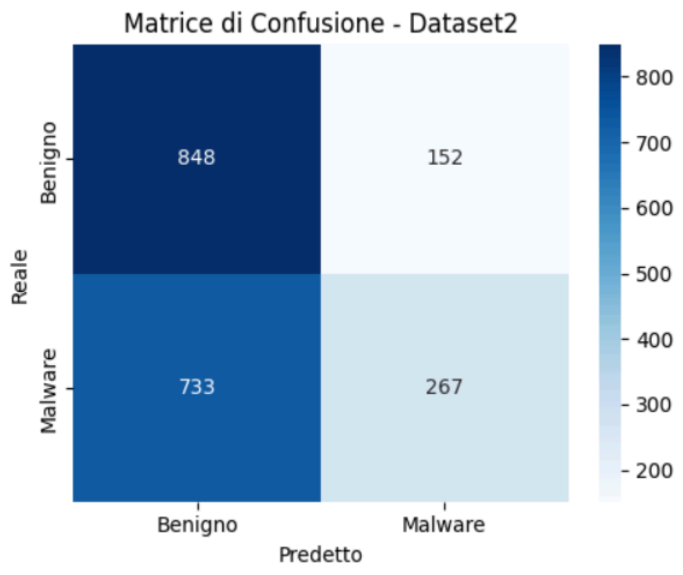


On the test set, the model demonstrates a strong ability to distinguish between benign files and malware, with balanced performance across both classes: 609 true negatives and 596 true positives, with only a few errors (47 false positives and 60 false negatives).
The **accuracy** is **91,84%** and this may suggest that the test set comes from a distribution similar to that used during training.

An analysis is then performed on Dataset 2:

```Python
              precision    recall  f1-score   support

     Benigno       0.54      0.85      0.66      1000
     Malware       0.64      0.27      0.38      1000
```



The results deteriorate significantly, particularly for the 'Malware' class: **733 false negatives** are observed, meaning malware instances were incorrectly classified as benign. This is reflected in the low **recall** for that class (**0.27**), highlighting the model's poor generalization capability on data from a different distribution with an **accuracy** of the **55%**. The model tends to adopt a conservative behavior, classifying many malware samples as benign, which poses serious risks in real-world security scenarios.

This analysis highlights the urgent need to introduce regularization techniques to improve the robustness and reliability of the detection system in unseen environments.

4.4 Improvements

In the final model, several significant modifications were made with the goal of improving the model's generalization capability, reducing overfitting, and increasing sensitivity to the malware class.

In particular:

- **Optimizer**: The Adam optimizer was replaced with AdamW, a variant that provides a more accurate handling of weight decay (L2 regularization) by decoupling it from gradient updates. This helps avoid interference between weight penalization and gradient computation, resulting in greater training stability.
  The **learning_rate** and **weight_decay** values were adjusted from their default settings to enhance stability and better control overfitting.

```python
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=1e-5,
    weight_decay=1e-3
)
```

- Loss Function: The standard **BCELoss** was replaced with **BCEWithLogitsLoss**, which internally includes the sigmoid function and provides more numerically stable handling of logits (the model's raw outputs).

Additionally, a higher weight was assigned to the malware class using the **pos_weight** parameter, increasing the penalty for false negatives. This choice reflects a conservative strategy: in cases of uncertainty, the model is encouraged to classify a file as malware rather than risk missing a real threat (false negative).

```python
criterion = nn.BCEWithLogitsLoss(pos_weight=torch.tensor([2.0]).to(device))
```

- **Dropout Layers:** In the AvastConv model, Dropout layers were added after the first two fully connected layers to reduce overfitting and increase the robustness of the network:

```python
def forward(self, x):
    x = self.seq2fix(x)

    x = F.selu(self.fc_1(x))
    x = F.dropout(x, p=0.3, training=self.training)

    x = F.selu(self.fc_2(x))
    x = F.dropout(x, p=0.5, training=self.training)

    x = F.selu(self.fc_3(x))
    x = self.fc_4(x)

    return x
```

  Note: The output is not passed directly through a sigmoid function, as **BCEWithLogitsLoss** includes it internally. This ensures numerical efficiency and stability.

- **Predict function**: With the introduction of the **BCEWithLogitsLoss** loss function, the model's output is no longer passed through a sigmoid within the model itself. As a result, the **predict** function was modified to apply the sigmoid externally only when necessary: during prediction, if the loss is **BCEWithLogitsLoss**, the sigmoid is applied to the logits to convert them into probabilities.

```python
is_logits_loss = isinstance(criterion, nn.BCEWithLogitsLoss)

if is_logits_loss or apply_sigmoid:
    probs = torch.sigmoid(raw_outputs)
    y_pred = (probs > 0.5).to(int)
```

- **Reproducibility of Experiments**: To ensure full reproducibility of the experiments conducted on the AvastConv model, appropriate strategies for seed control and computational determinism were adopted for both CPU and GPU operations. A **_seed_all(seed)** function was defined to set the seed across all modules and environments that influence random behavior.
- To also control the order of batch sampling during training, a **torch.Generator()** object with a fixed seed was used and explicitly passed to all DataLoaders.

```
Python
SEED = 42
g = torch.Generator()
g.manual_seed(SEED)

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE,
collate_fn=pad_collate_func, shuffle=True, generator=g)
```

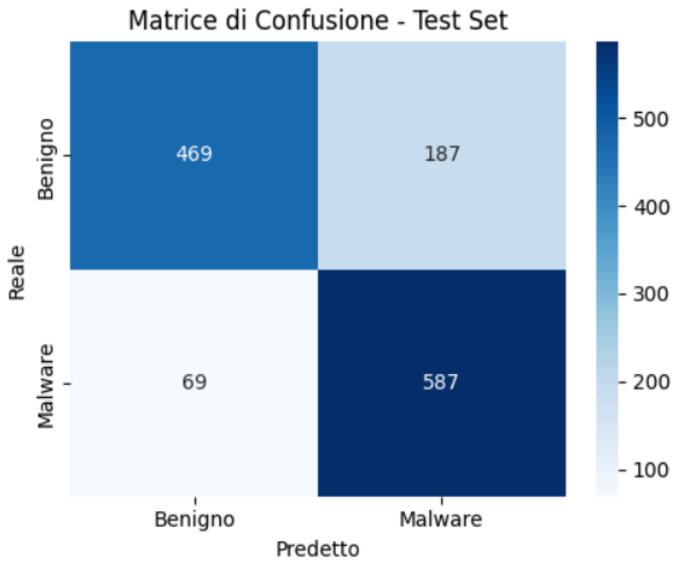
4.5 Final Experiment

The final model was saved as **Deep_Model_migliore.pkl** in the folder /content/drive/Shared Drives/AI for Cybersecurity/Consegna/Deep Model.

Le performance sul test set sono le seguenti:

```
Python
              precision    recall  f1-score   support

     Benigno       0.87      0.71      0.79       656
     Malware       0.76      0.89      0.82       656
```



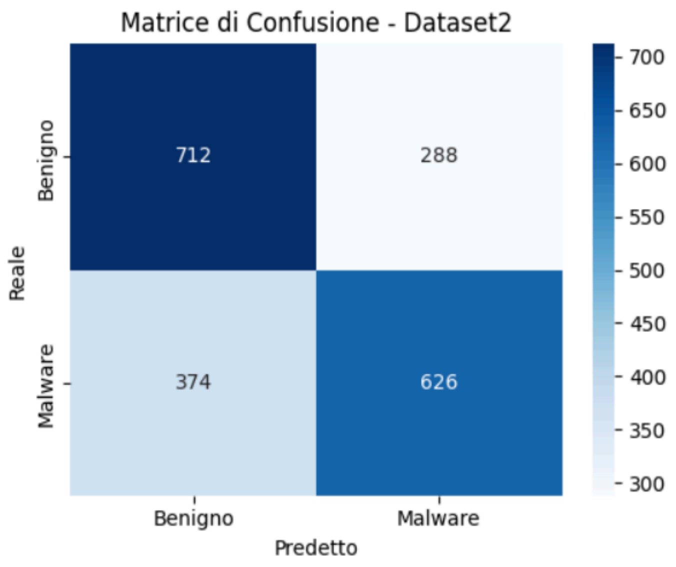
The confusion matrix for the test set highlights the model’s overall good performance, achieving—as expected—a **high recall on malware** samples and an **accuracy of the 88%**. Specifically, the model correctly identified 587 out of 656 malware files, making only 69 false negatives (i.e., cases where malware was classified as benign).

In contrast, the **'benign'** class shows a lower recall due to an increase in false positives, with 187 benign files incorrectly flagged as malware. While this behavior is not ideal, it is acceptable in the context of cybersecurity, where it is preferable to mistakenly classify a benign file as malware rather than overlook a real threat.

Finally, a verification of the model's generalization was performed on **Dataset 2**:

```
Python
              precision    recall  f1-score   support

     Benigno       0.66      0.71      0.68      1000
     Malware       0.68      0.63      0.65      1000
```



The model evaluation on Dataset 2—used to assess generalization capability—shows an overall performance drop compared to the main test set. The confusion matrix reveals a significant number of **false positives (288)** and **false negatives (374)**, which negatively affect both precision and F1-score for both classes.
Nonetheless, there is a **67% of accuracy**, the **recall** for the malware class remains acceptable **(0.63)**, indicating that the model continues to detect a good portion of threats, even on data from a distribution different from the one used during training. However, the lower **precision (0.68)** suggests that a non-negligible portion of benign files is still being misclassified as malware, increasing the false alarm rate. Overall, the model demonstrates partial generalization: it is still capable of identifying malware, but with increased uncertainty in distinguishing between benign and malicious files.
 These results point to the need for improvement strategies, such as including more diverse and representative data in the training set, to make the system more robust in real-world scenarios. These enhancements, however, are beyond the scope of this analysis.

5 GAMMA ATTACK

To evaluate the robustness of the previously trained models, a security analysis was conducted by launching adversarial attacks using the Genetic Adversarial Machine learning Malware Attack (GAMMA) framework. A total of 100 malware samples were randomly selected from the test set of our first dataset. The aim of the attack was to manipulate these malicious binaries in a way that **deceives the detection system into classifying them as benign**, while preserving their original functionality.

This adversarial technique employs a genetic algorithm that iteratively evolves candidate solutions by injecting benign code sections into malware binaries. Each candidate undergoes evolutionary steps such as crossover and mutation to optimize evasion. In this implementation, the attack leveraged a curated set of 50 benign sections—specifically from the `.data`, `.rdata`, `.idata`, and `.rodata` segments—extracted from a diverse benign dataset.

Each of these sections plays a distinct role in the structure and execution of a PE (Portable Executable) binary:

`.data` section: This section stores global and static variables that can be modified during runtime. It contains initialized writable data, which means its contents are already set when the program starts but may change as the program executes. Injecting benign data content can help preserve execution flow while subtly altering binary characteristics.

`.rdata` section: This section holds the debug directory which stores the type, size and location of various types of debug information stored in the file.

`.idata` section: The import section includes the information necessary for dynamic linking—specifically, it holds references to external libraries (DLLs) and functions that the program uses. Although more sensitive, its structure can be leveraged to introduce benign library references or metadata with low risk of interfering with program logic

`.rodata`: Although in most traditional PE binaries all read-only constants are merged into the `.rdata` section, certain toolchains—such as MinGW/Cygwin, LLVM when using long section names, or GNU ld with grouped subsections—sometimes emit specific string literals or lookup tables exclusively into a separate `.rodata` section rather than fusing them into `.rdata`.

In the experimental phase, several configurations of the GAMMA attack were explored to assess its effectiveness under different conditions. The choice of parameters aimed to strike a balance between performance and computational feasibility, while also allowing for a broad analysis of the model's vulnerability to adversarial manipulation.

The **population size**—representing the number of candidate solutions evaluated at each generation.

To evaluate the impact of the regularization term in the optimization process, the **λ (lambda) parameter**, which controls the trade-off between classification evasion and minimal perturbation, was tested across multiple magnitudes. Specifically, **λ encourages solutions with fewer injected bytes, at the expense of a reduced likelihood of misclassification as benign**. By adjusting λ, it is possible to favor stealthier attacks or more aggressive evasion strategies, depending on the objective.

The number of **iterations** for each run of the genetic algorithm was fixed at 100.

It is important to highlight that, unlike previous studies where a hard cap (e.g., 2.5 MB) was enforced on the amount of content injected into the malware, no explicit limit was applied in this work. This decision was primarily driven by the nature of the deep learning model used for classification—**AvastConv**—which processes the full binary content without truncation or fixed input sizing. As a result, file size becomes a critical factor: larger injections are not simply discarded, but can influence the entire embedding and convolution process.

Allowing unrestricted injection sizes enabled a more realistic and stress-oriented evaluation of the model's robustness, particularly in scenarios where adversarial samples become substantially larger than their original form. This approach also helped to better understand how sensitive the model is to changes in structural entropy and section distribution at scale.

5.1 Reproducibility of the Experiments

In black-box adversarial attacks based on genetic algorithms, such as the GAMMA Section Injection method used in this work, stochastic processes are fundamental. Randomness influences multiple stages of the algorithm, including initial population generation, mutation, crossover, and selection strategies. Therefore, **results can vary significantly across different runs**, even when the input sample and model remain constant.

To ensure **reproducibility** and enable **consistent comparison** across multiple experimental configurations (e.g., varying lambda values), it is essential to control this randomness by setting a **random seed**. Fixing a seed guarantees that the same sequence of random numbers is used, making the experiment deterministic and repeatable. This is particularly critical when evaluating the effect of specific parameters (such as the regularization coefficient λ) on the attack's effectiveness, as it isolates the variable under study from random fluctuations.

However, when attacking multiple samples—as in our case with a batch of 100 malware binaries—using a **single fixed seed** for all samples might bias the results by introducing correlations between different runs. To avoid this, a **distinct but deterministic seed** is generated for each sample (e.g., by combining a base seed with a counter or using a hash of the file name). This strategy maintains reproducibility for each individual attack while avoiding inter-sample dependencies.

5.2 Attack Limitations

Another important note is to ensure the practical feasibility of running the GAMMA attack within the resource constraints of the Google Colab environment, we introduced a series of runtime and sample selection controls. In particular, certain malware samples were observed to require excessive computational time—sometimes exceeding several minutes or even hours—causing a significant bottleneck in the overall experiment. Since Colab sessions are subject to execution time limits and may be automatically terminated if the process runs for too long, these prolonged runtimes pose a risk of losing progress and interrupting the entire experiment. The following chapters will introduce the mechanism chosen to mitigate this problem:

Pre-Filtering of Exception-Prone Malware Samples

During preliminary runs, certain PE binaries routinely triggered non-fatal warnings or exceptions within the LIEF parser (e.g., “Can’t read the padding content of section…” or “Export.AddressTableEntries is too large…”). Although these warnings/exceptions did not abort execution outright, they flooded the console, dramatically increasing I/O overhead, and also GAMMA attacks on those samples were much slower compared to other samples of the same size. To avoid these problems, we implemented a simple pre-filter: each candidate file is first validated with `pefile.PE()` and then parsed once by `lief.parse()`. If **any** exception is raised or **any** warning text is emitted to stdout/stderr during LIEF parsing, the sample is immediately discarded and never submitted to the GAMMA attack. This filtering stage removes the problem upfront, preventing repeated wasted attempts and ensuring that only well-formed, quickly parsable binaries proceed to the adversarial-generation phase. In accordance with the project specifications, this process continued until 100 malware samples correctly classified by our model had been accumulated for the attack experiments.

Early-Stopping

We introduced an **early-stopping criterion based on stagnation**: if the genetic optimizer fails to improve the best fitness score for five consecutive generations, the attack on that sample is terminated. This bound ensures that the algorithm does not waste further iterations exploring a local minimum, and instead moves on to the next sample.

5.3 First Experiment

In the first experiment, the following parameters were considered for generating attacks:

```python
how_many = 50
sections_to_extract = ['.data','.rdata', '.idata', '.rodata']
POPULATION_SIZE = 10
ITERATIONS = 100
QUERY_BUDGET = 800
```

Where $\lambda \in \{1e-4, 1e-5, 1e-6, 1e-7, 1e-8\}$.

5.3.1 Random Forest evaluation

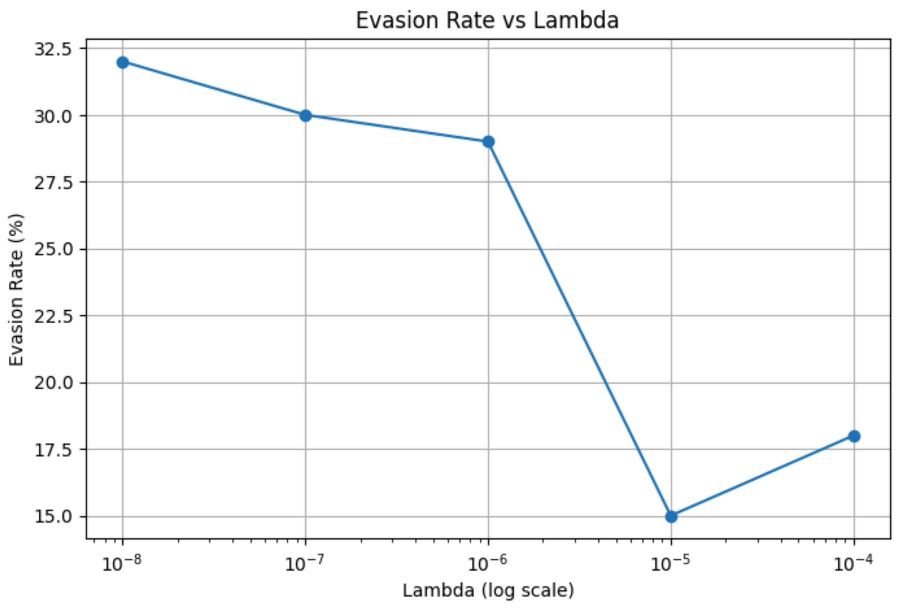
The values reported in the following table **represent the average number of bytes injected into the original malware samples and maximium number of bytes injected into the set of malware samples** as a result of the GAMMA attack, not the final file size:

λ	**Avg. Injected Bytes**	**max. Injected Bytes**
1e-4	126.29 KB	208.00 KB
1e-5	142.94 KB	220.00 KB
1e-6	152.87 KB	236.00 KB
1e-7	158.14 KB	236.00 KB
1e-8	160.11 KB	240.00 KB

In our experiments, section-injection operations consistently increased the size of each PE binary by an average of **X KB** (depending on the chosen λ). This growth can materially influence classifier performance:

- **Random Forest** relies heavily on features like byte-value histograms and overall entropy. Injecting additional sections alters these statistical distributions, which can shift the model's decision boundaries and affect its confidence.

The results of the GAMMA attack against **RandomForest** classifier are shown in the figure below:



Considering that the classifier originally had an accuracy of 100% on the original samples, the graph clearly illustrates the attack and how the parameter λ directly impacts the effectiveness of the attack. Lower λ values (e.g., 10^{-8}) promote more aggressive evasion strategies, achieving an evasion rate of approximately 32.5%. In contrast, higher values constrain the amount of modification allowed, which in turn reduces the success rate—down to around 15% with λ = 10^{-5}. This trend reflects the typical trade-off between evasiveness and minimal perturbation.

5.3.2 AvastConv evaluation

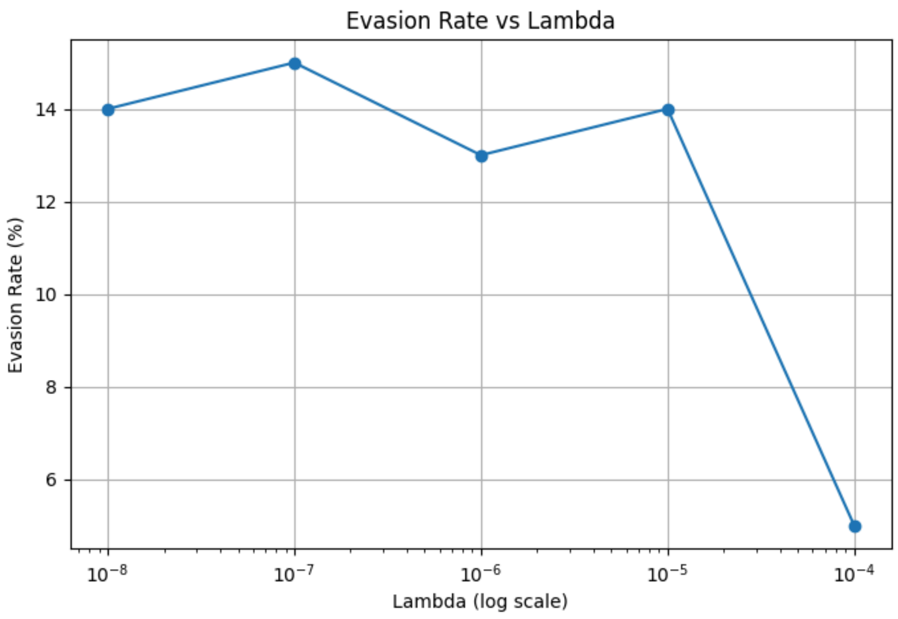
The values reported in the following table **represent the average number of bytes injected into the original malware samples and maximium number of bytes injected into the set of malware samples** as a result of the GAMMA attack, not the final file size:

λ	**Avg. Injected Bytes**	**max. Injected Bytes**
1e-4	141.46 KB	649.00 KB
1e-5	151.61 KB	661.00 KB
1e-6	164.88 KB	695.00 KB
1e-7	168.09 KB	714.00 KB
1e-8	168.58 KB	688.00 KB

In our experiments, section-injection operations consistently increased the size of each PE binary by an average of **X KB** (depending on the chosen λ). This growth can possibly influence the classifier performance:

- **AvastConv** processes the byte sequence directly. Each byte (0–255) is first mapped to an 8-dimensional binary embedding, and the model applies 1D convolutions over fixed-size chunks of the file. During training, every chunk is scanned without gradients to identify the positions with maximal activations, and backpropagation is performed only on those small “winning” segments. As a result, any increase in file length—caused by injected sections—changes the number and content of the chunks, which in turn alters the convolutional activations and can affect the final classification outcome.

The results of the GAMMA attack against AvastConv classifier are shown in the figure below:



The results show that AvastConv is considerably more resilient to GAMMA attacks compared to the Random Forest classifier. Even under the minimal value of lambda, the evasion rate does not exceed 15%. This robustness appears to stem from the model's deep architecture and the binary embedding of input bytes within convolutional layers, which collectively make it harder to successfully alter the model's decision boundaries.

5.4 Second Experiment

The following parameters were considered for generating attacks:

```python
how_many = 50 # how many section to extract from benign files
sections_to_extract = ['.data','.rdata', '.idata', '.rodata']
POPULATION_SIZE = 20
ITERATIONS = 100
QUERY_BUDGET = 800
```

Where $\lambda \in \{1e-4\ ,\ 1e-5,\ 1e-6,\ 1e-7\ ,\ 1e-8\ \}$.

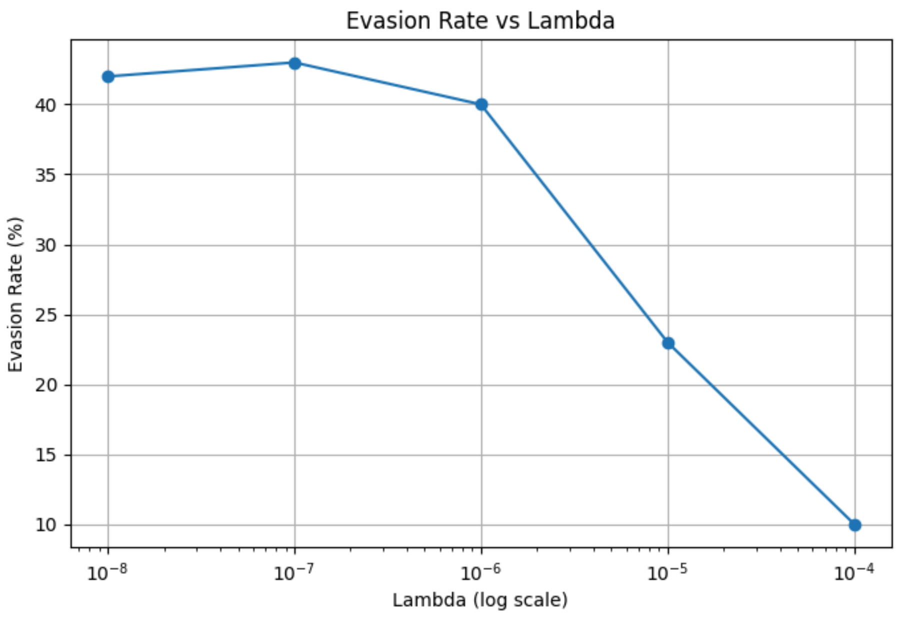
Compared with experiment 1, we want to analyse the impact of the attack as the population size parameter, which determines the number of population samples in the genetic algorithm at each step, changes.

5.4.1 Random Forest evaluation

The values reported in the following table **represent the average number of bytes injected into the original malware samples and maximium number of bytes injected into the set of malware samples** as a result of the GAMMA attack, not the final file size:

λ	Avg. Injected Bytes	max. Injected Bytes
1e-4	128.49 KB	208.00 KB
1e-5	134.77 KB	216.00 KB
1e-6	155.75 KB	232.00 KB
1e-7	164.47 KB	244.00 KB
1e-8	163.99 KB	244.00 KB

The results of the GAMMA attack against RandomForest classifier are shown in the figure below:



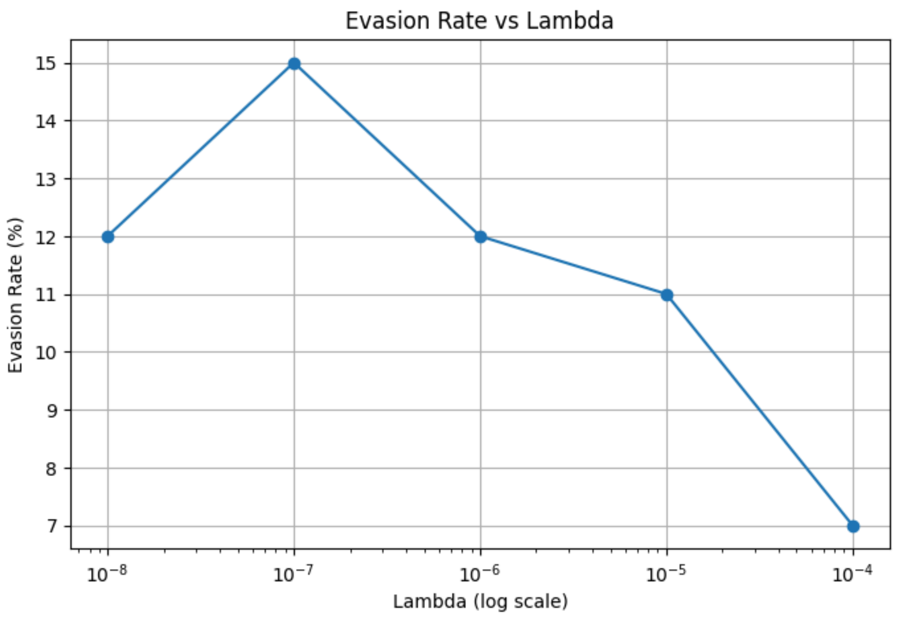
Increasing the population size in the GAMMA attack resulted in a clear improvement in evasion rate. A larger population promotes greater initial diversity, enables more comprehensive exploration of the search space, and enhances the effectiveness of selection during evolutionary iterations. These factors collectively lead to a more successful attack strategy. The previously discussed insights regarding the role of λ in balancing evasion success and perturbation magnitude remain fully valid in this context.

5.4.2 AvastConv evaluation

The values reported in the following table **represent the average number of bytes injected into the original malware samples and maximium number of bytes injected into the set of malware samples** as a result of the GAMMA attack, not the final file size:

λ	**Avg. Injected Bytes**	**max. Injected Bytes**
1e-4	131.56 KB	650.00 KB
1e-5	139.37 KB	678.00 KB
1e-6	163.32 KB	707.00 KB
1e-7	167.65 KB	698.00 KB
1e-8	166.55 KB	696.00 KB

The results of the GAMMA attack against AvastConv classifier are shown in the figure below:



Increasing the population size in the GAMMA framework yielded a slighy improvement in evasion rate against AvastConv for $\lambda = 10^{-7}$. But differently for the random forest the overall performance of the attack with an increase of the population size led to a general decrease of the evasion rate for other values of lambda.

5.5 Third Experiment

Within the third experiment, the following parameters were considered for generating attacks:

```
Python
how_many = 50 # how many section to extract from benign files
sections_to_extract = ['.data','.rdata', '.idata', '.rodata', '.rsrc','.reloc']
POPULATION_SIZE = 10
ITERATIONS = 100
QUERY_BUDGET = 800
```

Where $\lambda \in \{1e-4, 1e-5, 1e-6, 1e-7, 1e-8\}$.

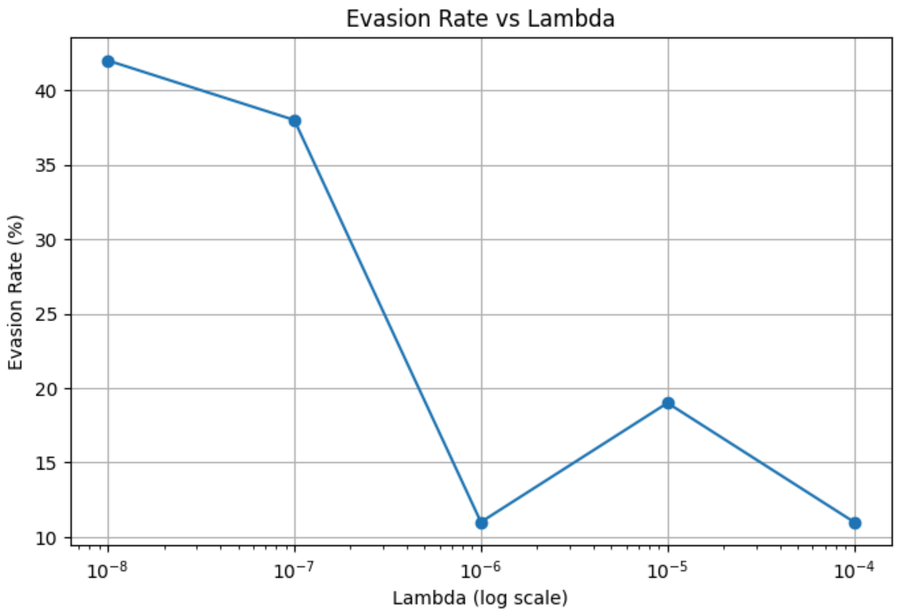
In the third experiment, we expanded the attack surface by including a broader variety of PE sections, in particular .rsrc and .reloc. We returned the genetic population size to 10 individuals and maintained 100 iterations per run to focus to understand the difference of adding new variety in the injected section.

5.5.1 Random Forest evaluation

The values reported in the following table **represent the average number of bytes injected into the original malware samples and maximium number of bytes injected into the set of malware samples** as a result of the GAMMA attack, not the final file size:

λ	Avg. Injected Bytes	max. Injected Bytes
1e-4	359.62 KB	485.0 KB
1e-5	459.73 KB	856.0 KB
1e-6	571.30 KB	1150.5 KB
1e-7	1071.18 KB	1324.0 KB
1e-8	2197.35 KB	10488.0 KB

The results of the GAMMA attack against RandomForest classifier are shown in the figure below:



This experiment shows that adding new type of sections to the binaries strengthened the GAMMA attack for the lower values of lambda, compared to the first experiment. However, increasing λ quickly diminishes the attack's effectiveness, highlighting the model's resilience to more conservative modifications. Surprisingly, when λ increases further to 10^{-5}, evasion jumps back up to around 19%, despite injecting even fewer bytes per sample (since the penalty is larger). This "local peak" suggests that at $\lambda = 10^{-5}$, there is a better balance between "enough injection to fool the model" and "not being overly penalized for each byte." In other words, the genetic search can now focus on more surgical, high-impact insertions rather than trying to inject large blocks and it finds pockets of benign content that disproportionately affect Random Forest's decision boundaries.

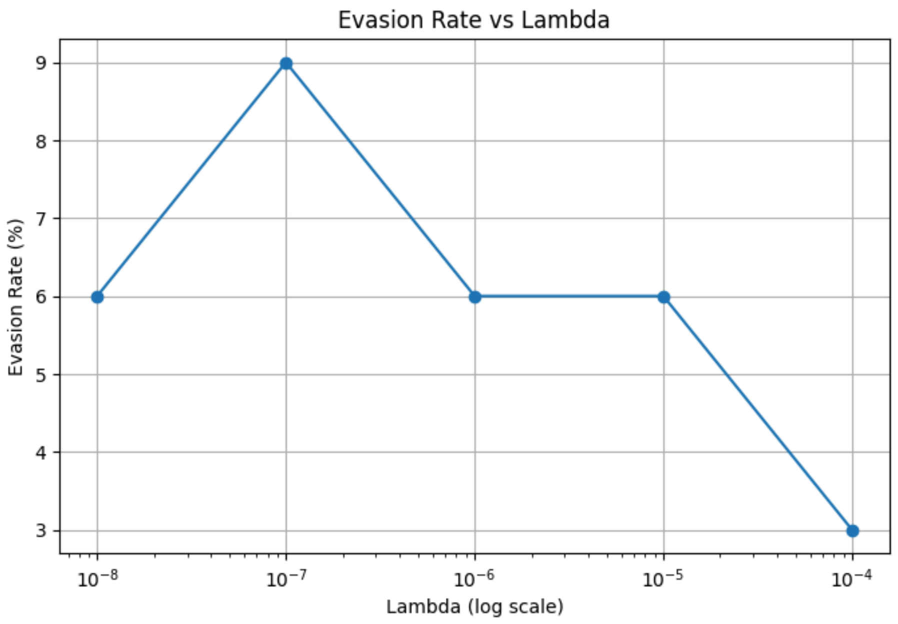
This phenomenon often happens in constrained optimization: a slightly tighter λ forces the algorithm to stop brute-forcing huge injections and instead discover a smaller set of highly strategic byte sequences that yield better evasion.

5.5.2 AvastConv evaluation

The values reported in the following table **represent the average number of bytes injected into the original malware samples and maximium number of bytes injected into the set of malware samples** as a result of the GAMMA attack, not the final file size:

λ	**Avg. Injected Bytes**	**max. Injected Bytes**
1e-4	321.94 KB	456.0 KB
1e-5	388.06 KB	624.0 KB
1e-6	644.84 KB	1096.0 KB
1e-7	843.37 KB	1144.5 KB
1e-8	847.40 KB	1232.0 KB

The results of the GAMMA attack against AvastConv classifier are shown in the figure below:



The plot shows that effect of addition of new type of section, even with low penalty on injected bytes ($\lambda = 10^{-8}$), AvastConv resists most attacks, achieving only a 6% evasion rate. When λ rises slightly to 10^{-7}, the attack becomes more surgical and peaks at 9% evasion. Beyond this point, higher λ values ($\geq 10^{-6}$) force the genetic algorithm into minimal insertions, causing evasion to drop back to 6% and eventually to 3% at $\lambda = 10^{-4}$. In short, AvastConv's convolutional processing makes it largely impervious to large-scale injections, and only a narrow λ "sweet spot" allows a modest increase in evasion.

5.6 Fourth Experiment

Within the fourth experiment, the following parameters were considered for generating attacks:

```
Python
how_many = 100 # how many section to extract from benign files
sections_to_extract = ['.data','.rdata', '.idata', '.rodata', '.rsrc','.reloc']
POPULATION_SIZE = 10
ITERATIONS = 100
QUERY_BUDGET = 800
```

Where $\lambda \in \{1e-4, 1e-5, 1e-6, 1e-7, 1e-8\}$.

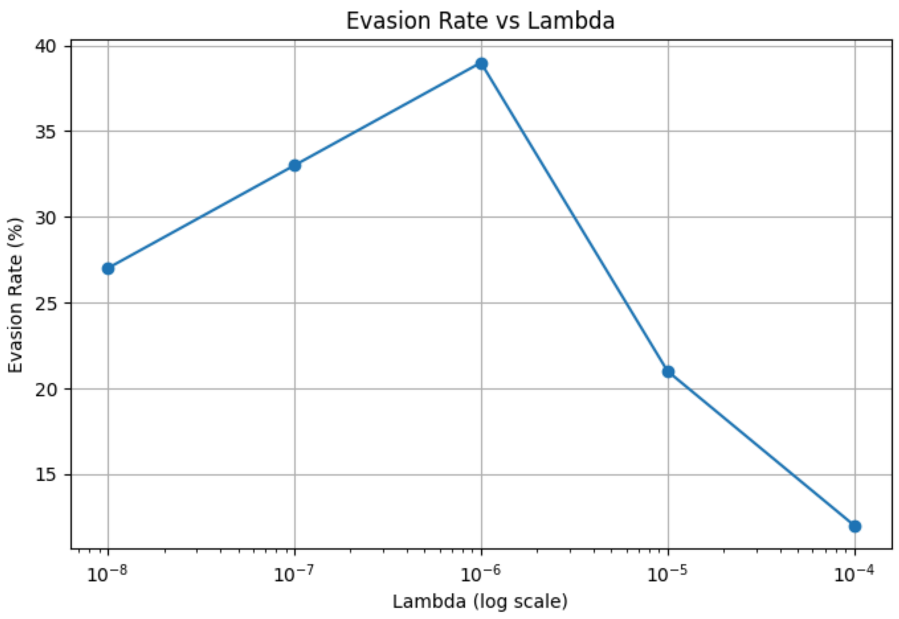
In the fourth experiment, we expanded the attack surface by including a greater number of PE sections, same as the previous experiment we maintained the extra two types of sections: .rsrc and .reloc. We still maintain the genetic population size to 10 individuals and maintain 100 iterations per run to focus to understand the difference of adding more injected sections.

5.6.1 Random Forest evaluation

The values reported in the following table **represent the average number of bytes injected into the original malware samples and maximium number of bytes injected into the set of malware samples** as a result of the GAMMA attack, not the final file size:

λ	Avg. Injected Bytes	max. Injected Bytes
1e-4	359.62 KB	485.0 KB
1e-5	459.73 KB	856.0 KB
1e-6	571.30 KB	1150.5 KB
1e-7	1071.18 KB	1324.0 KB
1e-8	2197.35 KB	10488.0 KB

The results of the GAMMA attack against RandomForest classifier are shown in the figure below:



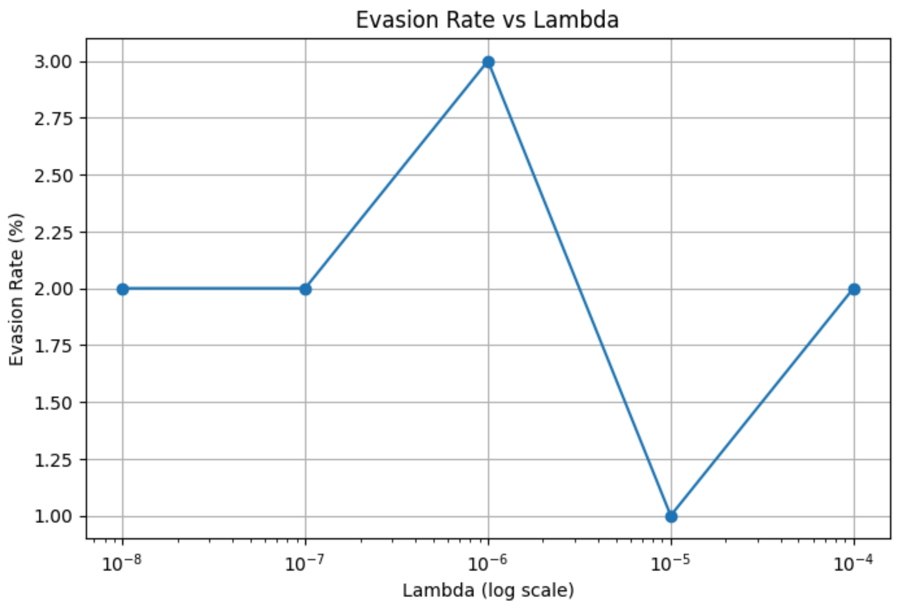
The graph shows that for Random Forest, the attack becomes most effective at λ = 10^{-6}, reaching nearly 39% evasion. As λ decreases (e.g., 10^{-8}), evasion slightly drops, suggesting that excessive injection may lead to more easily detectable anomalies. Generally the performance of the attack are still worse than the second experiment which remains the best for now.

5.6.2 AvastConv evaluation

The values reported in the following table **represent the average number of bytes injected into the original malware samples and maximium number of bytes injected into the set of malware samples** as a result of the GAMMA attack, not the final file size:

λ	**Avg. Injected Bytes**	**max. Injected Bytes**
1e-4	954.61 KB	2284.0 KB
1e-5	971.06 KB	2284.0 KB
1e-6	1078.87 KB	2514.5 KB
1e-7	1517.87 KB	4431.0 KB
1e-8	2653.51 KB	8520.0 KB

The results of the GAMMA attack against AvastConv classifier are shown in the figure below:



In the fourth experiment, AvastConv's evasion rate remained consistently below 3%—despite allowing many section types and injecting very large byte payloads. As shown in the table, average injected bytes ranged from roughly 954 KB at $\lambda = 10^{-4}$ up to over 2.65 MB at $\lambda = 10^{-8}$, with maximum injections exceeding 8 MB. Yet even these massive perturbations failed to meaningfully degrade classification accuracy. This behavior underscores that, as injected payload size increases, the attack's overall effectiveness actually decreases: AvastConv's deep convolutional layers and fixed binary embeddings dilute the impact of added sections, making it exceptionally resilient to these types of evasion attempts.

5.7 Fifth Experiment

Within the fifth experiment, the following parameters were considered for generating attacks:

```
Python
how_many = 100 # how many section to extract from benign files
sections_to_extract = ['.data','.rdata', '.idata', '.rodata', '.rsrc','.reloc']
POPULATION_SIZE = 20
ITERATIONS = 100
QUERY_BUDGET = 800
```

Where $\lambda \in \{1e-4\ , 1e-5, 1e-6, 1e-7\ , 1e-8\}$.

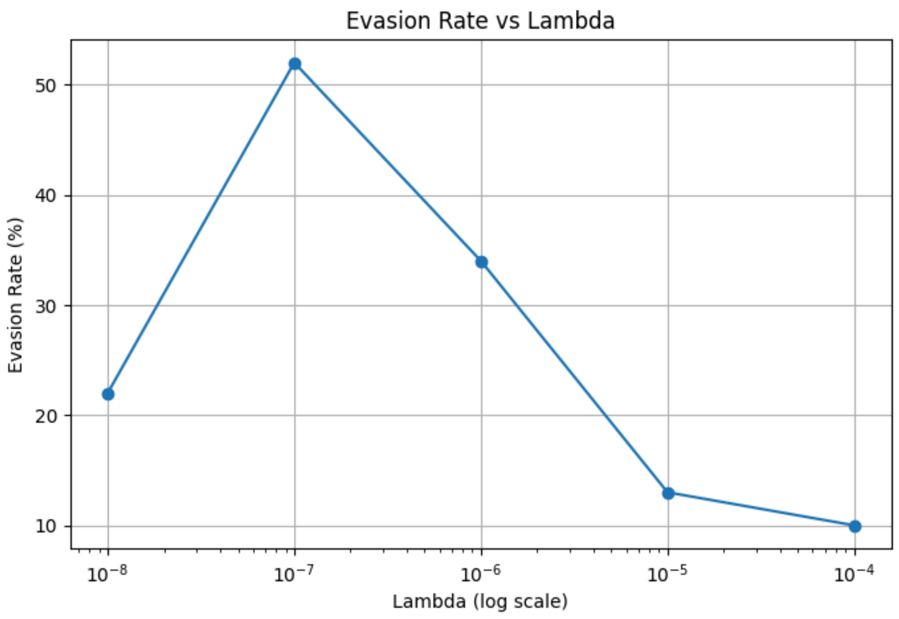
In the fifth experiment, we expanded the attack surface by including a greater number of PE sections, same as the previous experiment we maintained the extra two types of sections: .rsrc and .reloc. We increase the genetic population size to 20 individuals and maintain 100 iterations.

5.7.1 Random Forest evaluation

The values reported in the following table **represent the average number of bytes injected into the original malware samples and maximium number of bytes injected into the set of malware samples** as a result of the GAMMA attack, not the final file size:

λ	**Avg. Injected Bytes**	**max. Injected Bytes**
1e-4	832.51 KB	5876.06 KB
1e-5	836.74 KB	5868.06 KB
1e-6	928.04 KB	6078.03 KB
1e-7	1461.22 KB	6576.06 KB
1e-8	2060.07 KB	8032.06 KB

The results of the GAMMA attack against RandomForest classifier are shown in the figure below:



While earlier experiments demonstrated that tuning λ alone could create “sweet spots” where evasion peaks. Experiment 5 achieved the most pronounced increase in attack success against the RandomForest model by simultaneously **expanding the genetic population size to 20** and **doubling the benign sections** and variety type, while performing 100 iterations, we effectively increased the “vocabulary” of fragments that could be injected into the malware.

This expanded search space enabled the algorithm to discover new perturbation strategies that more potently distorted Random Forest’s statistical features. Concretely, Experiment 5’s Random Forest evasion rates (peaking near 52% at $\lambda = 10^{-7}$) exceeded those of earlier runs despite using the same λ grid because the combination of additional section types and a larger genetic population greatly *amplified the diversity of mutation patterns* and allowed for more robust recombination of high-impact benign payloads.

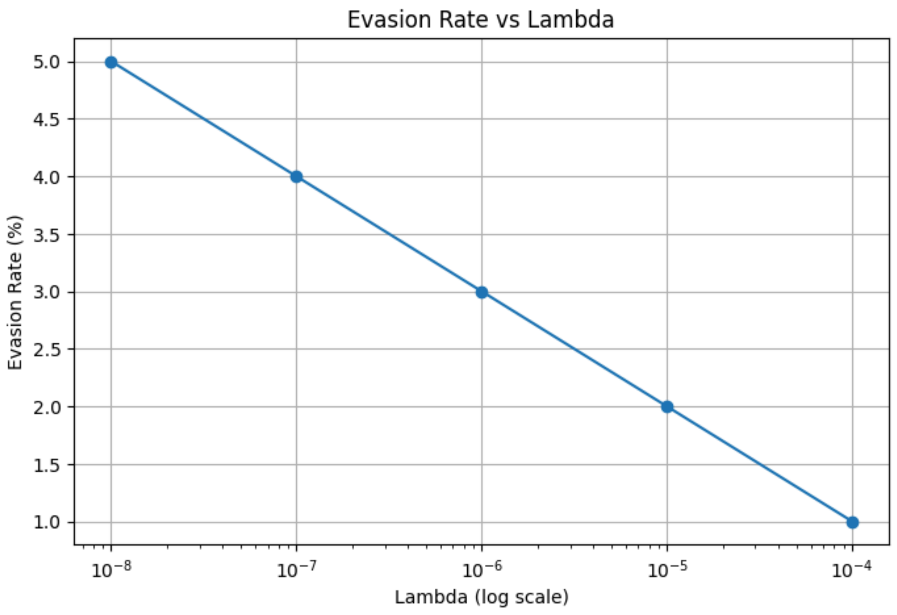
5.7.2 AvastConv evaluation

The values reported in the following table **represent the average number of bytes injected into the original malware samples and maximium number of bytes injected into the set of malware samples** as a result of the GAMMA attack, not the final file size:

λ	**Avg. Injected Bytes**	**max. Injected Bytes**
1e-4	641.80 KB	944.0 KB
1e-5	644.89 KB	960.0 KB
1e-6	959.72 KB	1258.0 KB

1e-7	1427.43 KB	2516.5 KB
1e-8	2266.62 KB	3160.0 KB

The results of the GAMMA attack against AvastConv classifier are shown in the figure below:

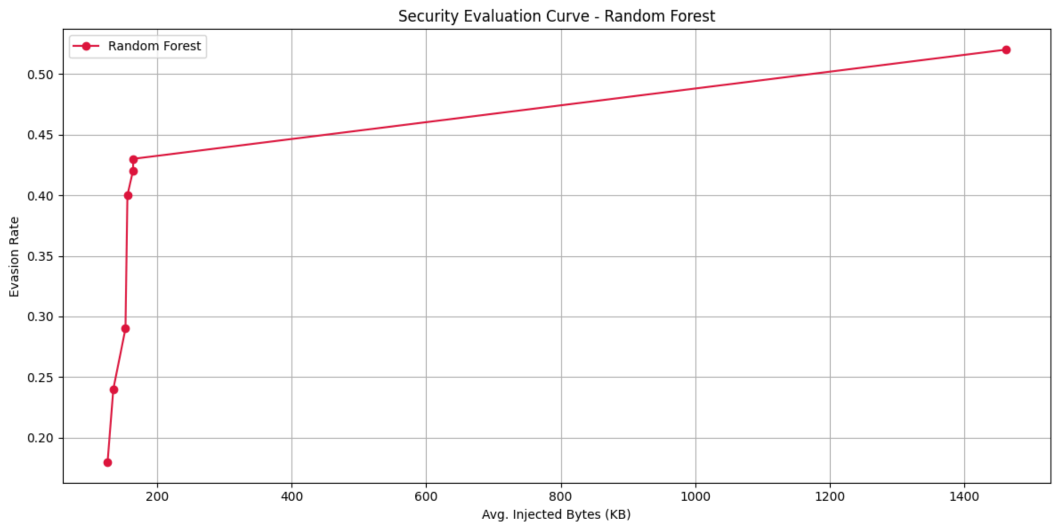


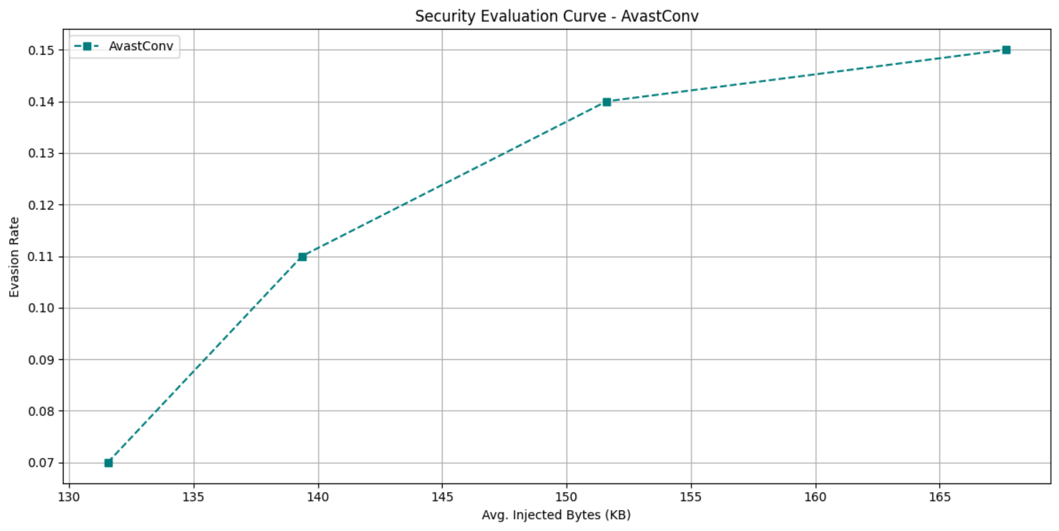
As already seen in the previous experiment, even increasing the strength of attack leading to injecting more bytes to the malware, shows a trend where the attack is very poorly efficient against this model.

5.8 Final Considerations

Our experiments with the **GAMMA attack against two malware detectors** RandomForest (78% generalization accuracy) and AvastConv (67% generalization accuracy) revealed a counterintuitive trend: the more accurate RandomForest model exhibited significantly higher evasion rates (up to 52%) compared to AvastConv (max 15%), despite larger adversarial perturbations. This suggests that generalization performance alone is insufficient to gauge robustness in adversarial settings.

To evaluate this robustness, we generated **Security Evaluation Curves (SEC)** that quantify the trade-off between perturbation effort (average injected bytes) and attack success (evasion rate). These curves were filtered to ensure monotonicity: points were sorted by injected KB, and only those showing increased evasion rates were retained. This process eliminated non-interpretable "zig-zagging" behavior, emphasizing the optimal attacker strategy for each model.





AvastConv showed a generally lower evasion rate across the same levels of injected KB, suggesting a more robust behavior compared to Random Forest while having lower generalization accuracy (lower than 70%).

AvastConv showed consistently lower evasion rates across all perturbation levels, despite its lower generalization accuracy (70%). This indicates that its architecture prioritizes robustness over nominal performance. Across all experiments, GAMMA’s success hinged not on single parameters (e.g., λ), but on the synergy between population size and benign section diversity. For example, Experiment 5 achieved peak evasion against RandomForest (52%) by combining a population size of 20, 100+ benign section types, and 100 iterations. This configuration expanded the attack’s "vocabulary," enabling it to discover perturbations that distorted RandomForest’s feature thresholds (e.g., file size, section count).

These experiments confirmed that *Random Forest models, when confronted with vastly more varied and abundant benign code, become significantly more susceptible* to manipulation because their decision boundaries rooted in handcrafted statistical features can be shifted more easily when the adversary has a large palette of payloads to explore.

In contrast, AvastConv remained highly robust throughout, and its “worst” performance in Experiment 5 was still far below Random Forest’s. Although Experiment 5 injected *on average over 1 MB of benign sections* (and up to 8.5 MB in extreme cases), **AvastConv’s evasion rate never exceeded ~15%**, remarkably, even with a population of 20. This pattern confirms that **AvastConv’s architecture is intrinsically resistant** to large-scale section-injection attacks—even when the adversary wields a maximally diverse arsenal. Furthermore a critical trend emerged in our experiments, **for AvastConv, increasing the injected payload size (KB) not only failed to improve evasion rates but often worsened the attacker’s success**. This stands in contrast with RandomForest, where larger perturbations increased the evasion rate, in fact, the average injected payload in Experiment 5 was *far larger* than in any previous run, yet AvastConv’s classification accuracy barely budged.

The main advantage of the robustness of AvastConv is thanks to its architecture that analyzes raw byte-level patterns rather than engineered features. This makes it resilient to "brute-force" perturbations like section injections: the convolutional layers detect local structures (e.g., malicious byte sequences, signatures, etc…) that are not easily disrupted by appending benign sections.

6 Defensive Methods

6.1 Adversarial training

One possible defensive approach to improve the robustness of malware classifiers against evasion attacks is **adversarial training**. This technique involves augmenting the training dataset with adversarially manipulated samples in order to expose the model to potential attack patterns during learning. Even when using simplified or less sophisticated versions of the original attacks, adversarial training can lead to noticeable improvements in resilience.

However, adversarial training is not a silver bullet. While it helps models to better handle known or similar threats, it does not fully generalize to unseen or more advanced attack strategies. Moreover, the process of generating adversarial malware samples is often computationally demanding and requires domain-specific constraints to ensure the resulting binaries remain functional. Consequently, adversarial training introduces additional computational and operational overhead compared to standard training pipelines.

6.2 GAMMA-attack detection

Given the proven ability of the GAMMA attack to produce adversarial malware samples that evade the main detector by injecting benign sections, we propose the addition of a **dedicated adversarial detector** trained specifically to recognize such modified files.

The goal of this preliminary step is **not to classify files as malware or benign**, but to **detect traces of Gamma injections** before the files are analyzed by the main classifier.

The proposed architecture consists of two components:

1. A **pre-filtering adversarial detector**, trained to distinguish between:
    - **Non-modified files**
    - **GAMMA-modified files** (i.e., adversarial samples)
2. The **main malware detector**, which operates as an antivirus classifier (e.g., RandomForest).

The adversarial detector must be placed **upstream**, before the main malware detector, so that any file flagged as “GAMMA-modified” can be:

- Blocked.
- Sent to sandbox analysis.
- Forwarded for deeper inspection.

This layered design enhances robustness by **catching GAMMA manipulations** before they reach the main detector.

To train such adversarial detector, which is a **binary classification model,** the following dataset should be used:

Label	**Class**
0	Non-modified Files
1	Gamma-modified Files

This method provides a **targeted defense** against adversarial attacks like GAMMA in a **modular and non-invasive** way: the main model is left untouched. By using this defensive layer, we aim to **filter obfuscated files** that would otherwise bypass detection, thus significantly increasing the overall security of the malware detection pipeline.

It's important to notify that this strategy **does not completely eliminate the threat** posed by adversarial attacks such as GAMMA. Several limitations and potential complications must be considered:

- **False Positives**: The adversarial detector may incorrectly flag legitimate malware (or even benign files in future versions) as adversarially modified, leading to unnecessary quarantining or investigation. This can increase operational overhead.
- **Generalization Risk**: A detector trained specifically on GAMMA variants may not generalize well to **other types of adversarial attacks** (e.g., gradient-based padding, slack byte attacks, or GAN-generated samples). Attackers can easily switch to different strategies, requiring frequent retraining of the detector.
- **Increased Complexity**: Adding a second detection stage introduces **system complexity**, both in terms of architecture and maintenance. It requires additional storage, inference time, and model management.

For these reasons, while the proposed adversarial filter is a **useful countermeasure**, it should be seen as a **complementary defense**, not a definitive solution.

6.3 Final consideration On Certified Defenses

In our context, conducting a thorough certified defense analysis proved to be particularly challenging. This is primarily due to the fact that the project itself does not impose any constraints on the strength or nature of the attacks that can be performed. Moreover, the datasets at our disposal were neither large nor diverse enough to support a broader range of experiments involving alternative attack modifications. As such, it would be premature and somewhat unjustified to attempt a formal certified defense for our models. Additionally, the limited time frame of the project, coupled with constrained computational resources, further restricted our ability to explore this aspect in greater depth.