NEXT-GENERATION MALWARE DETECTION : LIGHTWEIGHT AND INTERPRETABLE ML MODEL FOR OBFUSCATED THREATS

Report submitted to SASTRA Deemed to be University
As per the requirement for the course

CSE300: MINI PROJECT

Submitted by

ARUN J
(Reg No.: 126003031, B. Tech Computer Science and Engineering)
PRAGADHEESH R A
(Reg. No.: 126003197, B. Tech Computer Science and Engineering)
PRANUSSHRAJ M G
(Reg. No.: 126003201, B. Tech Computer Science and Engineering)

MAY 2025

SASTRA
ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION
DEEMED TO BE UNIVERSITY
(U/S 3 of the UGC Act,1956)
THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

SCHOOL OF COMPUTING
THANJAVUR, TAMIL NADU, INDIA – 613 401

SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act,1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

SCHOOL OF COMPUTING

THANJAVUR – 613 401

<u>Bonafide Certificate</u>

This is to certify that the report titled **"NEXT-GENERATION MALWARE DETECTION: LIGHTWEIGHT AND INTERPRETABLE ML MODEL FOR OBFUSCATED THREATS"** submitted as a requirement for the course, **CSE300 : MINI PROJECT** for B.Tech. is a bonafide record of the work done by **Mr. ARUN J** (Reg. No.: 126003031, B. Tech Computer Science and Engineering), **Mr. PRAGADHEESH R A** (Reg. No.: 126003197, B. Tech Computer Science and Engineering) and **Mr. PRANUSSHRAJ M G** (Reg. No.: 126003201, B. Tech Computer Science and Engineering) during the academic year 2024-25, in the School of Computing, under my supervision.

Signature of Project Supervisor	: [illegible]
Name with Affiliation	: Dr.M.SUMATHI, AP-III, SOC
Date	: 23.05.2025

Mini Project *Viva voice* held on ________________

Examiner 1 **Examiner 2**

ACKNOWLEDGEMENTS

We would like to thank our Honorable Chancellor **Prof. R. Sethuraman** for providing us with an opportunity and the necessary infrastructure for carrying out this project as a part of our curriculum.

We would like to thank our Honorable Vice-Chancellor **Dr. S. Vaidhyasubramaniam** and **Dr. S. Swaminathan**, Dean, Planning & Development, for the encouragement and strategic support at every step of our college life.

We extend our sincere thanks to **Dr. R. Chandramouli**, Registrar, SASTRA Deemed to be University for providing the opportunity to pursue this project.

We extend our heartfelt thanks to **Dr. V. S. Shankar Sriram**, Dean, School of Computing, **Dr. R. Muthaiah,** Associate Dean, Research, **Dr. K.Ramkumar**, Associate Dean, Academics, **Dr. D. Manivannan,** Associate Dean, Infrastructure, **Dr. R. Algeswaran**, Associate Dean, Students Welfare

Our guide **Dr. M SUMATHI** , Assistant Professor - III, School of Computing was the driving force behind this whole idea from the start. Her deep insight in the field and invaluable suggestions helped us in making progress throughout our project work. We also thank the project review panel members for their valuable comments and insights which made this project better.

We would like to extend our gratitude to all the teaching and non-teaching faculties of the School of Computing who have either directly or indirectly helped us in the completion of the project.

We gratefully acknowledge all the contributions and encouragement from my family and friends resulting in the successful completion of this project. We thank you all for providing us an opportunity to showcase our skills through this project.

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ML	Machine Learning
RF	Random Forest
GB	Gradient Boosting
DT	Decision Tree
SVM	Support Vector Machine
LR	Logistic Regression
KNN	K-Nearest Neighbors
NB	Naive Bayes
SHAP	SHapley Additive exPlanations
DLL	Dynamic-Link Library
CPU	Central Processing Unit
RAM	Random Access Memory
GPU	Graphics Processing Unit
OS	Operating System
CIC-MalMem-2022	Canadian Institute for Cybersecurity Malware Memory Dataset 2022
AP	Assistant Professor
SOC	School of Computing
PE	Portable Executable
CNN	Convolutional Neural Network

ABSTRACT

Malware detection is very important in the cybersecurity domain, especially with the increasing number of obfuscation techniques. Traditional methods are not anymore easy to identify new and unseen malware variants. This makes the use of machine learning (ML) highly useful. This report studies the need for an adaptive malware detection system capable of identifying both already existing and new malware threats. The system was evaluated on the CIC-MalMem-2022 dataset (58,596 memory dumps: 50% benign, 50% malware across 15 subtypes). The project focuses on building a lightweight, fast, and interpretable malware detection system using the Random Forest (RF) classifier. The RF classifier is selected here for its superior performance over six other ML models. The system was designed to detect new malware variants by training 15 separate models on different malware subtypes, using a reduced dataset and the top five most important features for each subtype. The proposed solution achieved very high accuracy, with 11 out of the 15 malware subtypes exceeding an accuracy of 99%, and the Transponder subtype reaching up to 99.84% accuracy. Additionally, the system had high classification speed (5.7 μs per instance) and compact model size (340 KB). Interpretability was done using SHapley Additive exPlanations (SHAP), explaining the details of the decision-making process and enhancing transparency. This demonstrates the system's capability to identify both known and unseen malware effectively.

KEY WORDS: Machine Learning, Malware Detection, Cybersecurity, Obfuscation Techniques, SHapley Additive exPlanations (SHAP)

TABLE OF CONTENTS

CHAPTER 1
SUMMARY OF BASE PAPER

Title	:	Detecting new obfuscated malware variants
Publisher	:	Elsevier
Year	:	2025
Journal	:	Intelligent Systems with Applications
Indexing	:	Scopus
Base paper URL	:	https://www.sciencedirect.com/science/article/pii/S2667305324001467

1.1 INTRODUCTION

In recent times, rapid growth in technologies like cloud computing, the Internet of Things (IoT), and high-speed mobile networks like 5G have changed how organizations access, process, and store data. These technologies gave rise to real-time connectivity across various sectors all over the globe, but they also introduced a growing number of security threats. As digital convenience expands, the surface area for cyber threats—particularly from malware does expand too. A malware is designed to infiltrate systems, steal sensitive data, or cause damage without the consent of the user. As the number of connected devices and reliance on digital infrastructure increase, the malware attacks also have become more frequent and more complicated. Traditional signature-based and behavior-based malware detection methods often struggle to detect the modern malware, especially ones which are designed to evade detection.

Due to these limitations, machine learning (ML) has become popular as a tool for malware detection. ML models are very much capable of learning from large datasets to identify patterns and detect anomalies in the behavior of the software. These models are trained to differentiate between benign and malicious samples which are then used to classify new files as benign or malicious. The key challenge is that most ML-based systems are trained only on known malware types, making them less effective for identifying a new, zero-day malware attack type. Malicious actors adapt and evolve their techniques in a very short interval of time, so this is a critical weakness in the ML-based systems. This project focuses on evaluating the ML models to detect malware that differs from training data that they were trained on. To achieve this an obfuscated malware dataset that mimics real-world evasion techniques is used.

1.2 RELATED WORK

Various other publications were explored that utilize ML algorithms to detect malicious activities in a particular file. Below are some of them with their respective merits and demerits:

- **"Malware Detection System Based on Adversarial Training, Liu et al. (2020)".** The authors have implemented a malware classification model which is built on adversarial training techniques applied to the static features from the Windows PE files. The model performed well with an accuracy of 97.73% and had resistance to many adversarial examples. Despite these strengths, the approach is limited because of its static-only analysis, which makes it less effective when obfuscated or dynamically adaptive malware types are present in the malicious file.

- **"Malware Classification Using a Dilated Convolutional Neural Network, Mezina and Burget (2022)".** This publication explores deep learning with the help of dilated CNNs to classify obfuscated malware from benign files. It achieved 99.89% accuracy, and the model majorly focuses on handling complex obfuscation patterns within the malware file. The model's large size and computational cost reduces its suitability for real-time or resource-constrained environments which is the major limitation.

- **"Ensemble Malware Classification Using Memory-Based Features, Carrier et al. (2022)".** The result proposed in this publication utilizes an ensemble of classifiers including Naïve Bayes, Random Forest, and Decision Tree which were trained on memory forensics data obtained from a particular file. The model achieved 99% accuracy and introduced the CIC-MalMem-2022 dataset for the purpose od obfuscated malware detection. The model does not address the challenge of detecting zero-day malware variants, which limits its capability to detect evolving threat environments.

- **"XMal: A Lightweight Memory-Based Explainable Obfuscated Malware Detector, Alani et al. (2023)".** The paper introduces a highly efficient malware detection system using Extreme Gradient Boosting and recursive feature elimination technique. The model has a compact size (575 KB), rapid processing speed, and integrated SHAP-based explainability. Its limitation is that it focuses on already-known malware subtypes, without testing generalizability to new or unseen attacks.

- **"Malware Detection Using Logistic Regression and Gradient Boost Tree, Dener et al. (2022)".** This applies traditional ML algorithms—logistic regression and gradient boosting—on memory-derived features for the purpose of malware classification. It achieved a high accuracy of 99.97% and. But, the scope of the work was confined to binary classification and did not explore performance on zero-day threats or subtype-level identification.

1.3 PROBLEM STATEMENT

As the malware variants continue to evolve with advancing obfuscation techniques, traditional malware detection methods are struggling to keep up when it comes to identifying a new or unknown malware variants creating a need for a more adaptable and efficient malware detection system. The goal is to develop a detection solution that is accurate and capable of detecting both known and novel malware and also lightweight and interpretable.

1.4 OBJECTIVE

The objective of this project work is to design and implement a lightweight and interpretable malware detection system that employes Random Forest (RF) model for classification and feature selection, to effectively detect obfuscated malware variants. By training models on a reduced number of features and employing SHapley Additive exPlanations (SHAP) for the purpose of model interpretability, the system aims to achieve a high accuracy in detecting both known and novel malware while maintaining low resource consumption. This approach solves the limitations of traditional methods discussed above by enhancing adaptability and transparency in malware detection.

1.5 PROPOSED SYSTEM

1.5.1 Work Flow Diagram

The proposed system architecture is structured into five main stages: pre-processing, baseline classification, feature selection, adaptive classification, and model interpretability. The process begins by loading and cleaning the CICMalMem-2022 dataset and followed by training the baseline models using the cross-validation and test split method. Each subtype model is evaluated against its ability to detect unseen malware variants. Finally, SHAP is implemented to interpret predictions of the through Beeswarm and Force plots. The workflow diagram is depicted in Fig 1.1.

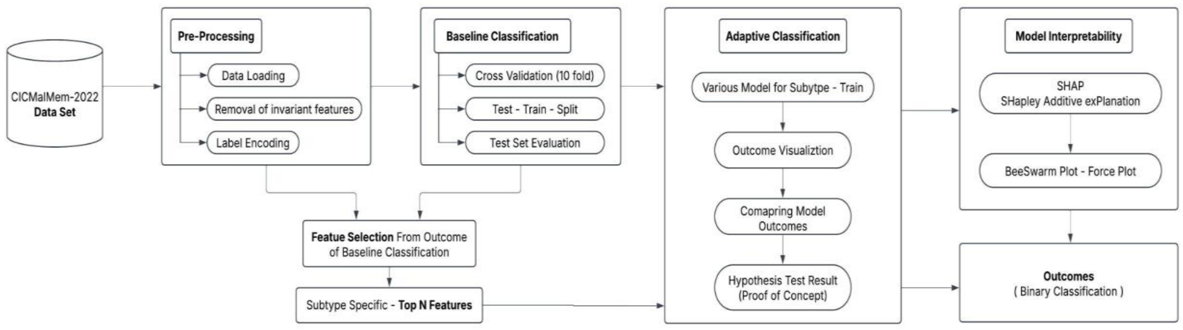

CICMalMem-2022 Data Set
Pre-Processing
Data Loading
Removal of invariant features
Label Encoding
Baseline Classification
Cross Validation (10 fold)
Test - Train - Split
Test Set Evaluation
Featue Selection From Outcome of Baseline Classification
Subtype Specific - Top N Features
Adaptive Classification
Various Model for Subytpe - Train
Outcome Visualiztion
Comapring Model Outcomes
Hypothesis Test Result (Proof of Concept)
Model Interpretability
SHAP SHapley Additive exPlanation
BeeSwarm Plot - Force Plot
Outcomes (Binary Classification)

Fig 1.1. Workflow Diagram

1.5.2 Dataset

The CIC-Malmem-2022 dataset used in this project. It is an open-source dataset introduced by Carrier et al. (2022). The dataset was created by obtaining memory dumps of recent real-life malware attack incidents. Out of 58,596 records which it has, it was evenly split as 29,298 benign and 29,298 malicious file instances. Each of those instances consisted of 55 features, which were extracted from the single memory dump file using VolMemLyzer (Lashkari et al., 2021). Some examples of those features include the number of running processes, the number of open dynamic-link libraries (DLLs), the average number of threads per process and the number of open files. The dataset further classifies each instance malicious file by its malware type (Ransomware, Spyware or Trojan Horse) and malware subtype (e.g., Zeus, Gator, Pysa, etc). The summarization of the distribution of malware types and subtypes in the dataset is visualized in Table 1.1.

DataSet URL : https://www.kaggle.com/datasets/luccagodoy/obfuscated-malware-memory-2022-cic

Table 1.1. Malware Subtype Distribution

Malware type	Malware subtype	Number of instance	Percentage (%)
Trojan Horse	Zeus	1950	3.3
	Emotet	1967	3.4
	Refroso	2000	3.4
	Scar	2000	3.4
	Reconyc	1570	2.7
Spyware	180Solutions	2000	3.4
	CoolWebSearch	2000	3.4
	Gator	2200	3.8
	Transponder	2410	4.1
	TIBS	1410	2.4
Ransomware	Conti	1988	3.4
	MAZE	1958	3.3
	Pysa	1717	2.9
	Ako	2000	3.4
	Shade	2128	3.6
Total	-	29298	50

1.6 METHODOLOGY AND IMPLEMENTATION

The methodology is segregated into three modules for proper implementation. The pictorial representation of this module segregation is given below:

Module 0 - Data Preprocessing and Feature Selection

Module 1 - Baseline Classification (Against known malware)

Module 2 - Adaptive Malware detection (Zero-day detection)

Module 3 - Model interpretability

1.6.1 Module 0 : Data Preprocessing and Feature Selection

The pre-processing phase is a very crucial step in the malware detection pipeline, which is aimed to transform raw memory dump data from the CICMalMem-2022 dataset into a clean and well structured format suitable for implementing the machine learning model.

1. **Data Loading:**
   The dataset, comprising 58,596 records (29,298 malware and 29,298 benign), is imported using a Python-based library 'Pandas'. Each of those records includes 55 features extracted from memory dumps. These features capture the runtime process behaviors such as the open handles, DLLs, services, and kernel drivers, which are useful indicators of potential malware activity.

2. **Removal of Invariant Features:**
   Invariant features are attributes that have the same value across all or nearly all samples. These are identified using variance calculations:
   $$\text{Variance} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$
   where x_i is the value of the feature in the sample, and $\bar{x}$ is the mean of that feature. If the variance is zero or below the predefined threshold, the feature is considered to be non-informative and it is removed. In this study, features like *pslist.nprocs64bit*, *handles.nport*, and *svcscan.interactive_process_services* were excluded for being invariant.

1.6.2 Module 1 : Baseline Classification

This module focuses on establishing a baseline for malware detection by training and evaluating several machine learning algorithms using the prepared fully pre-processed dataset. The primary goal of this module is to assess how well different classifiers algorithms can distinguish between malware (label = 1) and benign (label = 0) instances using all the available memory-based features (excluding the invariant ones removed in Module 0).

1. **Train-Test Split**

   The dataset is split into training (80%) and testing (20%) subsets using stratified sampling. This ensures an equal proportion of malware and benign instances in both sets, preserving class balance during training and evaluation.

2. **Classifier Training**

   2.1. Random Forest

   A random forest builds multiple decision trees on random subsets of data and combines their outputs. Improve decision tree performance by creating many trees and averaging their results. Random Forest is an ensemble learning method that constructs multiple decision trees using random subsets of the training data and random subsets of the features. Each tree makes an independent prediction, and for classification tasks, the final output is determined by a majority vote, while for regression tasks, the predictions are averaged. This randomness helps reduce overfitting and improves model accuracy by combining uncorrelated trees.

   How it works and it's uses:

   - Builds multiple decision trees on random subsets of data and features.
   - Each tree gives a vote; the majority vote (classification) or average (regression) is taken as the final output.

   Strengths:

   - Reduces overfitting compared to single decision trees.
   - Works well for large datasets with many features.

   2.2. Decision Tree

A Decision Tree is a supervised, non-parametric algorithm that represents decisions in a tree structure. Starting at a single root node, the algorithm splits the data based on feature values, creating branches and further subdivisions until the data is partitioned into homogeneous leaf nodes that represent outcomes. It works by Splitting the data at each node based on the feature that gives the best separation (e.g., using Gini Impurity, Entropy).

$$Entropy = -\sum_{i=1}^{C} p_i \log_2(p_i) \qquad Gini = 1 - \sum_{i=1}^{C} p_i^2$$

- Simple, interpretable, but prone to overfitting.

2.3. Gradient Booster

Gradient Boosting is an ensemble technique that builds a strong predictive model by sequentially adding weak learners, typically small decision trees. At each iteration, the algorithm trains a new model to predict the residual errors (the gradient of the loss function) of the combined model from the previous iterations. This process of iteratively correcting mistakes continues until a predefined stopping criterion is met, such as a maximum number of iterations or minimal improvement in error reduction.

How it works:

- Each new model is trained to predict the residual (error) of the previous model.
- Uses gradient descent to minimize a loss function.
- The final prediction is a combination of all models.
- Combines weak learners (often shallow trees) to form a strong learner.

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

hm(x) is the weak learner (a tree), and γ(m) is the step size (learning rate).

2.4. Support Vector Machine (SVM)

Support Vector Machine (SVM) is a supervised learning algorithm used for both classification and regression tasks. It works by finding the optimal hyperplane that best separates the data into classes while maximizing the margin between different class boundaries. For datasets that are not linearly separable, SVM can use kernel functions to map the data into a higher-dimensional space where a clear separation is possible.

How it works:

- Finds the optimal hyperplane that best separates different classes.
- SVM finds the line (or plane in higher dimensions) that best separates the data into two classes.
- It tries to maximize the margin (distance) between the closest points of each class and the dividing line.
- Maximizes the margin between support vectors (critical boundary points).

2.5. Logistic Regression

Logistic Regression is a statistical model used primarily for binary classification problems. Unlike linear regression, which predicts continuous outcomes, logistic regression models the probability that a given input belongs to a particular class by applying the logistic (sigmoid) function to a linear combination of the input features. The output is a probability value between 0 and 1, which can then be thresholded to assign a class label.

How it works :

- Models the probability that an instance belongs to a class using sigmoid function.
- Outputs probabilities between 0 and 1.
- Decision boundary usually set at 0.5.
- Uses Maximum Likelihood Estimation (MLE) for training.

$$P(y=1|x) = \frac{1}{1+e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_n x_n)}}$$

2.6. Naïve Bayes

Naive Bayes is a probabilistic classifier based on Bayes' Theorem. It assumes that all features in the dataset contribute independently to the probability of an outcome, which greatly simplifies the computation. Despite this strong independence assumption, Naive Bayes often performs well, especially in high-dimensional data, by calculating the posterior probability for each class and choosing the class with the highest probability. It works by calculating the posterior probability:

$$P(\text{Class} \mid \text{Data}) = \frac{P(\text{Data} \mid \text{Class}) \cdot P(\text{Class})}{P(\text{Data})}$$

- Fast and effective, especially for text classification like spam filtering.

2.7. K-Nearest Neighbors (KNN)

K-Nearest Neighbour (KNN) is an instance-based learning algorithm that classifies new data points by comparing them to the 'k' closest examples in the training data. The algorithm calculates the distance between the new data point and all examples in the dataset using a predefined metric (such as Euclidean distance). For classification, the most frequent class among the nearest neighbours is assigned, while for regression, the average of the neighbours' values is used.

How It Works:

- For a new data point, k-NN looks at the k nearest points and assigns the most common class (for classification) or the average (for regression).
- The "distance" between points is often measured by how far apart they are (Euclidean distance).
- Sensitive to distance metric and feature scaling

$$d(x, x') = \sqrt{\sum_{i=1}^{n} (x_i - x'_i)^2}$$

3. Performance Metrics

Model performance is evaluated using standard classification metrics:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \qquad \text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN} \qquad F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

4. **Why Random Forest Was Chosen:**
   - **Highest Accuracy** : Random Forest consistently outperformed other models with an accuracy of 100% during baseline testing.
   - **F1-Score** : It achieved the highest F1-Score among all classifiers, ensuring balanced precision and recall.
   - **Robustness to Overfitting** : The ensemble nature of RF makes it resistant to overfitting compared to single decision trees.
   - **Feature Importance** : RF can rank features by importance, aiding in feature selection for a lightweight model
   - **Generalization Capability** : It showed better adaptability to unseen malware subtypes during adaptive testing.
   - **Computational Efficiency** :While other models like Gradient Boosting are accurate, RF strikes a better balance between speed and accuracy, suitable for real-time applications.

By combining high accuracy, interpretability, and efficiency, Random Forest proved to be the best fit for the malware detection system.

1.6.3 Module 2 : Adaptive Malware detection

This Module is the main novelty in this system — it introduces an adaptive malware detection strategy that is designed to test a model's ability to detect unseen malware subtypes, simulating a real-world zero-day attack scenarios. Traditional models are trained on mixed malware samples but, this module trains a separate model for each one of the malware subtype and evaluates it across the entire dataset containing other subtypes.

1. **Subtype-Specific Model Training**

   For each of the 15 malware subtypes in the CICMalMem-2022 dataset, an individual model is trained using:
   - 80% of the data from a single malware subtype
   - An equal number of randomly sampled benign instances

   This results in a balanced training set for each subtype. The remaining 20% of the subtype, along with all other unseen malware subtypes and benign samples, is used as the test set.

2. **Top-N Feature Subset Selection**

   To ensure the system remains lightweight and generalizable:
   - Only the top 5 features (as ranked by Random Forest feature importance in the previous step) are used for training and testing.
   - This drastically reduces computational overhead while maintaining classification performance.

The Transponder-trained model (Spyware subtype) yielded the highest accuracy of about 99.84% when tested across all unseen subtypes. The average detection speed remained fast (e.g., 5.7 µs per instance), which highlights the model's suitability for real-time deployment.

1.6.4 Module 3 : Model interpretability

This module focuses on explaining the decision-making process of the malware detection model using SHAP (SHapley Additive exPlanations) enhancing transparency by quantifying the contribution of each and every feature to a specific prediction. Global interpretability is achieved through SHAP Beeswarm plots, which show the overall impact of features across the dataset, while local interpretability is provided via SHAP Force plots that visualize feature influence for individual predictions.

1.7 SYSTEM REQUIREMENTS

1. **Hardware Requirements:**

Processor	:	Intel(R) Core i5-1035G1 CPU @ 1.00 GHz
RAM	:	8 GB
GPU	:	Intel(R) UHD Graphics
OS	:	Windows 10 Home

2. **Software Requirements:**

   Python v3.11.5

   Libraries used :

   Pandas (v1.3.5),

   Scikit-learn (v1.2.2),

   Matplotlib (v3.6.2),

   SHAP (v0.42.1)

CHAPTER 2
MERITS AND DEMERITS OF BASE PAPER

2.1 MERITS

- **High Accuracy with Minimal Data**: Achieves over 99.8% accuracy by training on just one malware subtype.
- **Lightweight Model Design:** Uses only the top 5 features and keeps the final model size as low as 340 KB, which is useful in deploying it in resource-constrained environments
- **Fast Detection Time**: Classifies a single sample in approx. 5.7 microseconds, making suitable for real-time malware detection applications.
- **Adaptability to Unseen Malware**: The system is tested across unseen instances, simulating a zero-day attack and showcases excellent adaptive performance.
- **Explainable Predictions via SHAP:** Integrates SHAP to provide both global and local interpretability, enhancing transparency of the model's behavior.

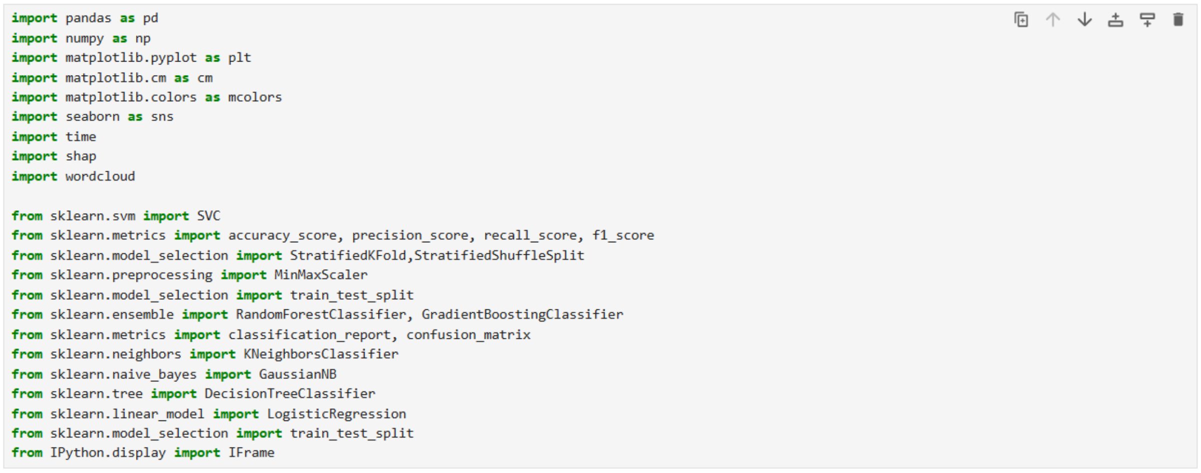
2.2 DEMERITS

- **Subtype Dependency**: The effectiveness of generalization heavily depends on which subtype it is trained on; some subtypes like Refroso or Zeus may underperform when generalized.
- **Limited Multiclass Evaluation**: The focus is on binary classification (malware vs. benign), with no multiclass classification of malware families included.
- **No Deep Learning Comparison**: Although the model outperforms traditional ML models, it doesn't directly compare its results with modern deep learning techniques under the same constraints.

CHAPTER 3
SOURCE CODE

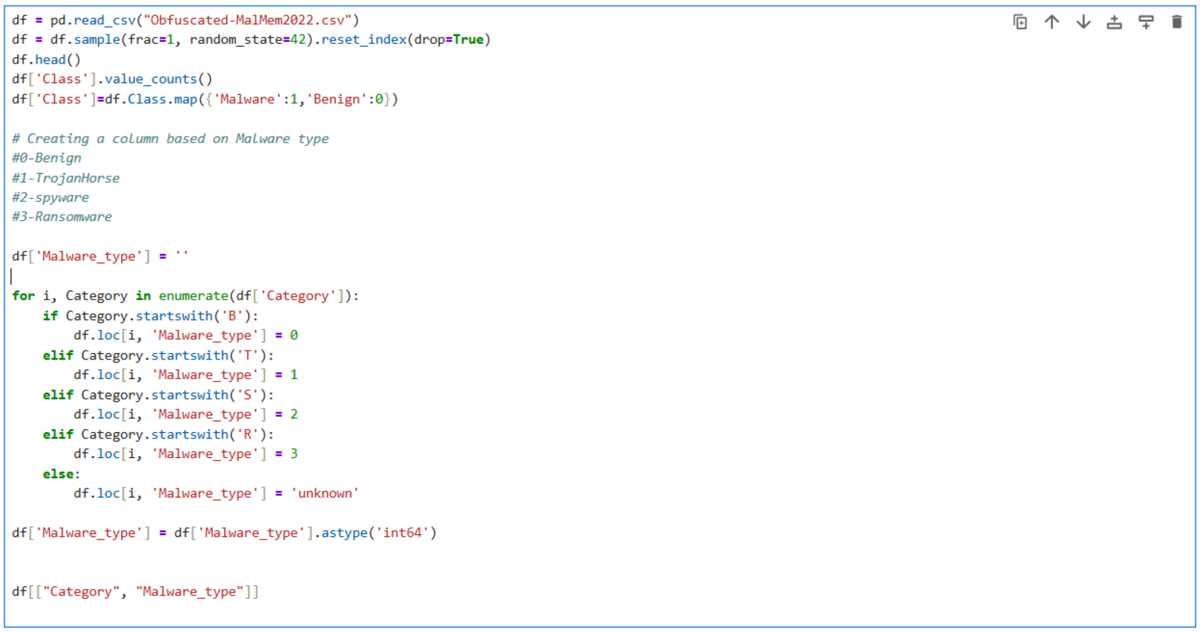
3.1 MODULE 0 : DATA PREPROCESSING

3.1.1 Making necessary imports



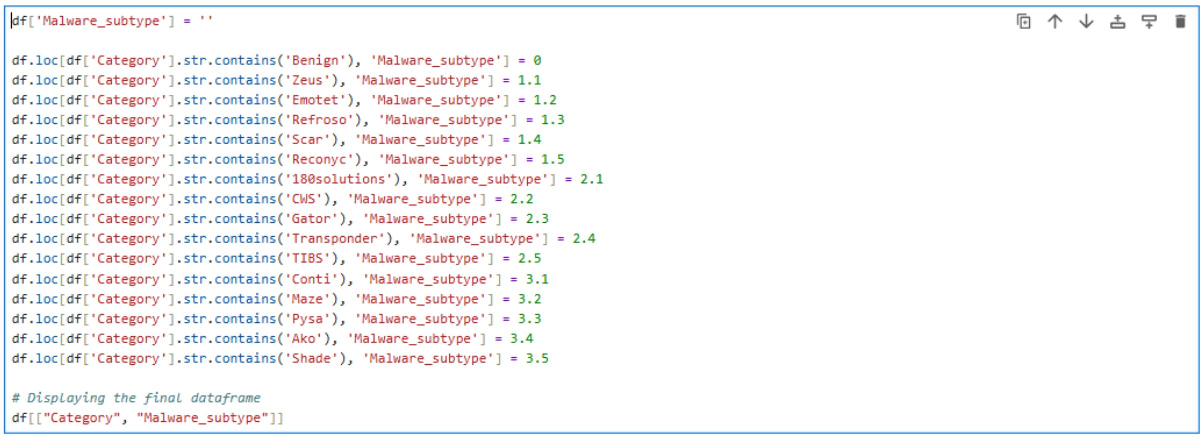
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as mcolors
import seaborn as sns
import time
import shap
import wordcloud

from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.model_selection import StratifiedKFold,StratifiedShuffleSplit
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from IPython.display import IFrame
```

3.1.2 Label encoding



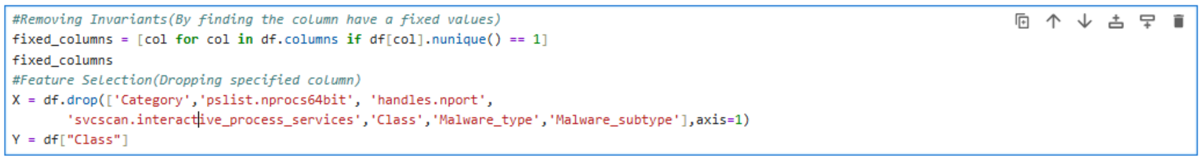
```python
df = pd.read_csv("Obfuscated-MalMem2022.csv")
df = df.sample(frac=1, random_state=42).reset_index(drop=True)
df.head()
df['Class'].value_counts()
df['Class']=df.Class.map({'Malware':1,'Benign':0})

# Creating a column based on Malware type
#0-Benign
#1-TrojanHorse
#2-spyware
#3-Ransomware

df['Malware_type'] = ''

for i, Category in enumerate(df['Category']):
    if Category.startswith('B'):
        df.loc[i, 'Malware_type'] = 0
    elif Category.startswith('T'):
        df.loc[i, 'Malware_type'] = 1
    elif Category.startswith('S'):
        df.loc[i, 'Malware_type'] = 2
    elif Category.startswith('R'):
        df.loc[i, 'Malware_type'] = 3
    else:
        df.loc[i, 'Malware_type'] = 'unknown'

df['Malware_type'] = df['Malware_type'].astype('int64')


df[["Category", "Malware_type"]]
```

3.1.3 Label encoding for subtypes



```
df['Malware_subtype'] = ''

df.loc[df['Category'].str.contains('Benign'), 'Malware_subtype'] = 0
df.loc[df['Category'].str.contains('Zeus'), 'Malware_subtype'] = 1.1
df.loc[df['Category'].str.contains('Emotet'), 'Malware_subtype'] = 1.2
df.loc[df['Category'].str.contains('Refroso'), 'Malware_subtype'] = 1.3
df.loc[df['Category'].str.contains('Scar'), 'Malware_subtype'] = 1.4
df.loc[df['Category'].str.contains('Reconyc'), 'Malware_subtype'] = 1.5
df.loc[df['Category'].str.contains('180solutions'), 'Malware_subtype'] = 2.1
df.loc[df['Category'].str.contains('CWS'), 'Malware_subtype'] = 2.2
df.loc[df['Category'].str.contains('Gator'), 'Malware_subtype'] = 2.3
df.loc[df['Category'].str.contains('Transponder'), 'Malware_subtype'] = 2.4
df.loc[df['Category'].str.contains('TIBS'), 'Malware_subtype'] = 2.5
df.loc[df['Category'].str.contains('Conti'), 'Malware_subtype'] = 3.1
df.loc[df['Category'].str.contains('Maze'), 'Malware_subtype'] = 3.2
df.loc[df['Category'].str.contains('Pysa'), 'Malware_subtype'] = 3.3
df.loc[df['Category'].str.contains('Ako'), 'Malware_subtype'] = 3.4
df.loc[df['Category'].str.contains('Shade'), 'Malware_subtype'] = 3.5

# Displaying the final dataframe
df[["Category", "Malware_subtype"]]
```

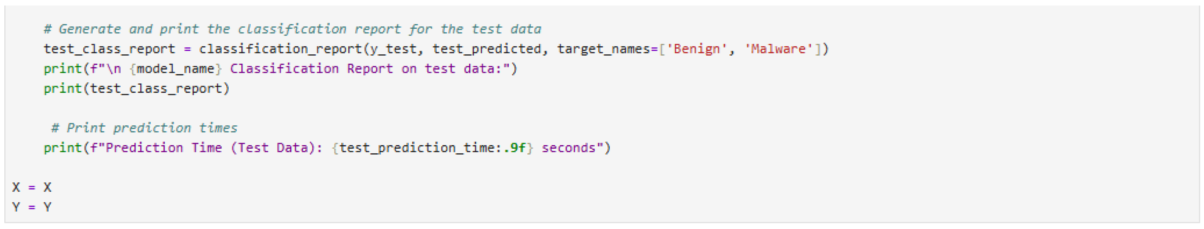
3.1.4 Removing invariant features



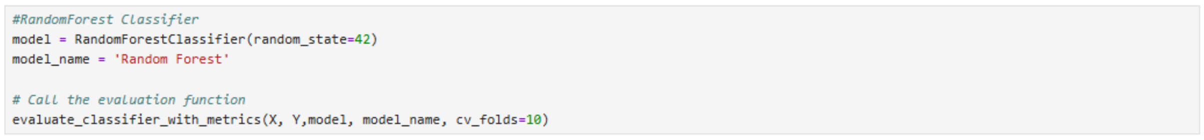
```
#Removing Invariants(By finding the column have a fixed values)
fixed_columns = [col for col in df.columns if df[col].nunique() == 1]
fixed_columns
#Feature Selection(Dropping specified column)
X = df.drop(['Category','pslist.nprocs64bit', 'handles.nport',
       'svcscan.interactive_process_services','Class','Malware_type','Malware_subtype'],axis=1)
Y = df["Class"]
```

3.2 MODULE 1 : BASELINE CLASSIFICATION

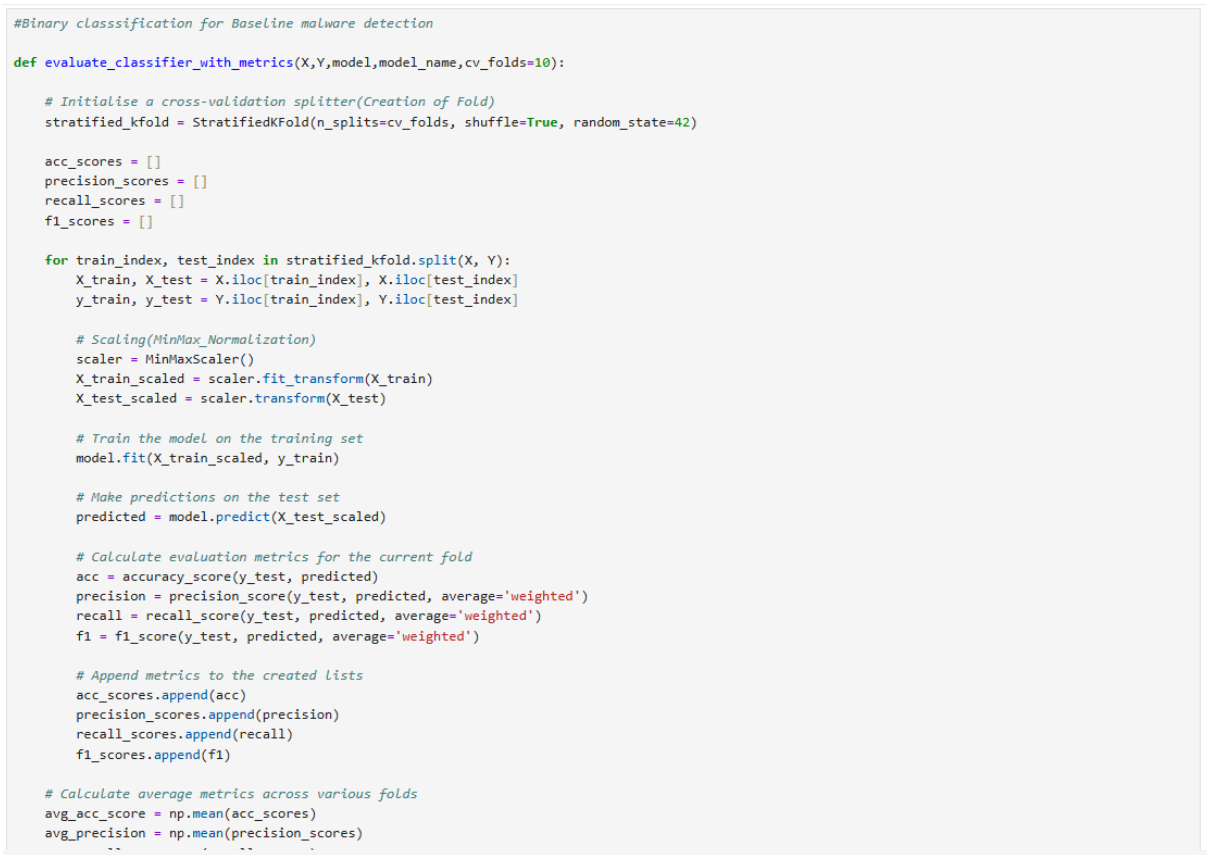
3.2.1 Evaluation of classifiers with metrics



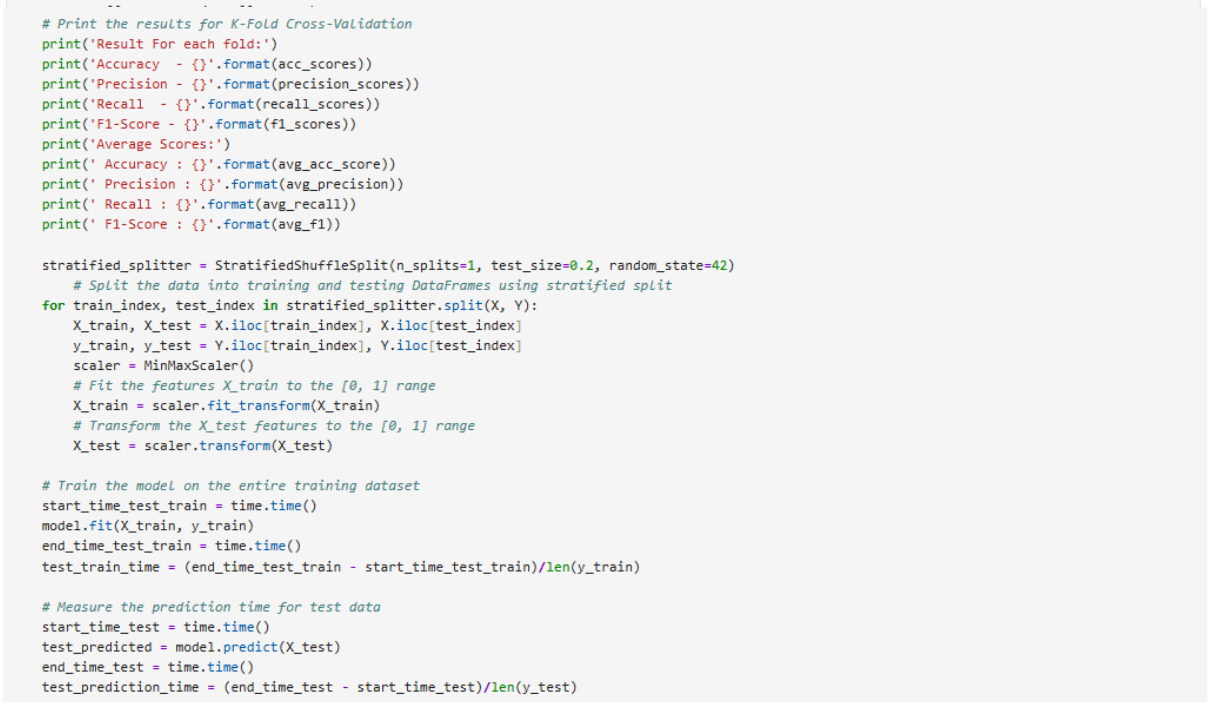
```
    # Generate and print the classification report for the test data
    test_class_report = classification_report(y_test, test_predicted, target_names=['Benign', 'Malware'])
    print(f"\n {model_name} Classification Report on test data:")
    print(test_class_report)

     # Print prediction times
    print(f"Prediction Time (Test Data): {test_prediction_time:.9f} seconds")

X = X
Y = Y
```



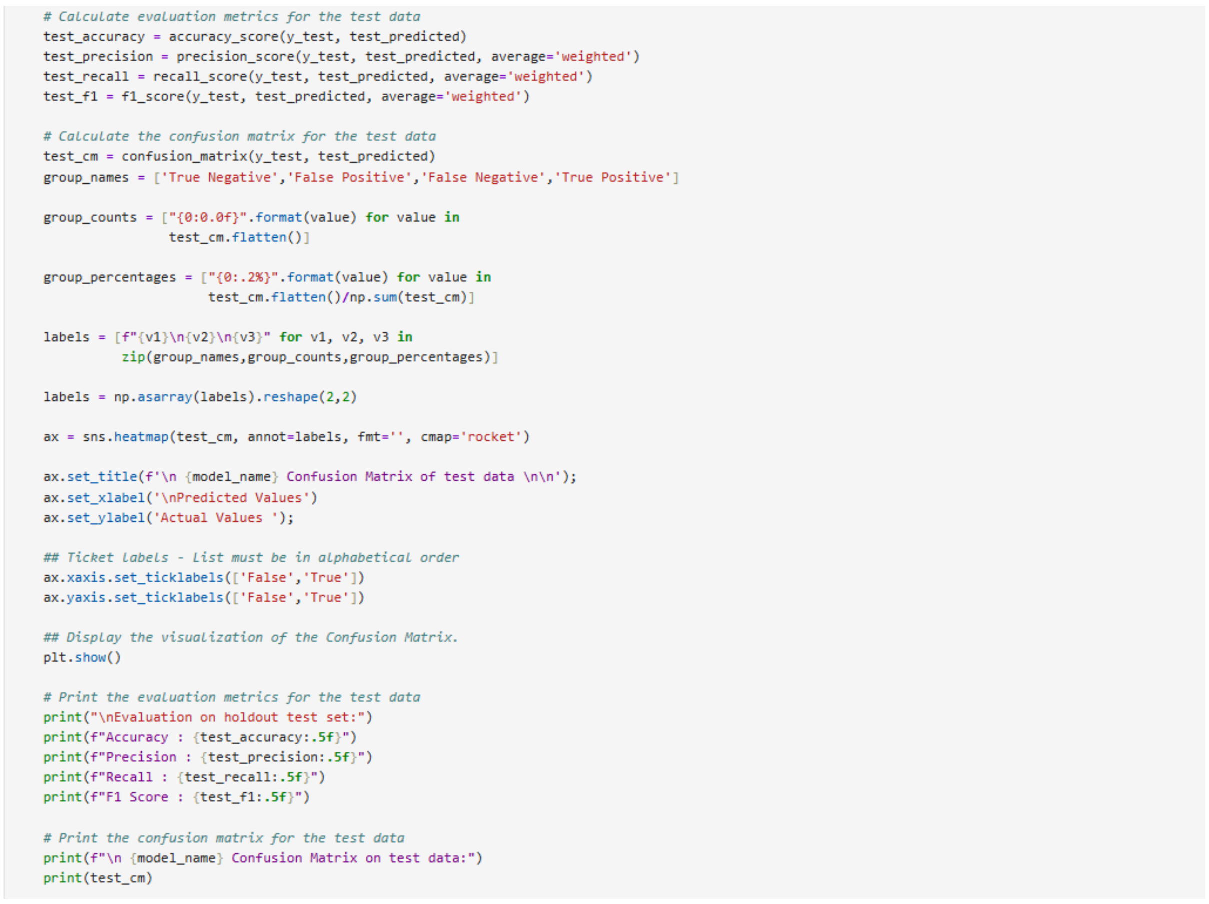
```
#RandomForest Classifier
model = RandomForestClassifier(random_state=42)
model_name = 'Random Forest'

# Call the evaluation function
evaluate_classifier_with_metrics(X, Y,model, model_name, cv_folds=10)
```



```
#Binary classsification for Baseline malware detection

def evaluate_classifier_with_metrics(X,Y,model,model_name,cv_folds=10):

    # Initialise a cross-validation splitter(Creation of Fold)
    stratified_kfold = StratifiedKFold(n_splits=cv_folds, shuffle=True, random_state=42)

    acc_scores = []
    precision_scores = []
    recall_scores = []
    f1_scores = []

    for train_index, test_index in stratified_kfold.split(X, Y):
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = Y.iloc[train_index], Y.iloc[test_index]

        # Scaling(MinMax_Normalization)
        scaler = MinMaxScaler()
        X_train_scaled = scaler.fit_transform(X_train)
        X_test_scaled = scaler.transform(X_test)

        # Train the model on the training set
        model.fit(X_train_scaled, y_train)

        # Make predictions on the test set
        predicted = model.predict(X_test_scaled)

        # Calculate evaluation metrics for the current fold
        acc = accuracy_score(y_test, predicted)
        precision = precision_score(y_test, predicted, average='weighted')
        recall = recall_score(y_test, predicted, average='weighted')
        f1 = f1_score(y_test, predicted, average='weighted')

        # Append metrics to the created lists
        acc_scores.append(acc)
        precision_scores.append(precision)
        recall_scores.append(recall)
        f1_scores.append(f1)

    # Calculate average metrics across various folds
    avg_acc_score = np.mean(acc_scores)
    avg_precision = np.mean(precision_scores)
```



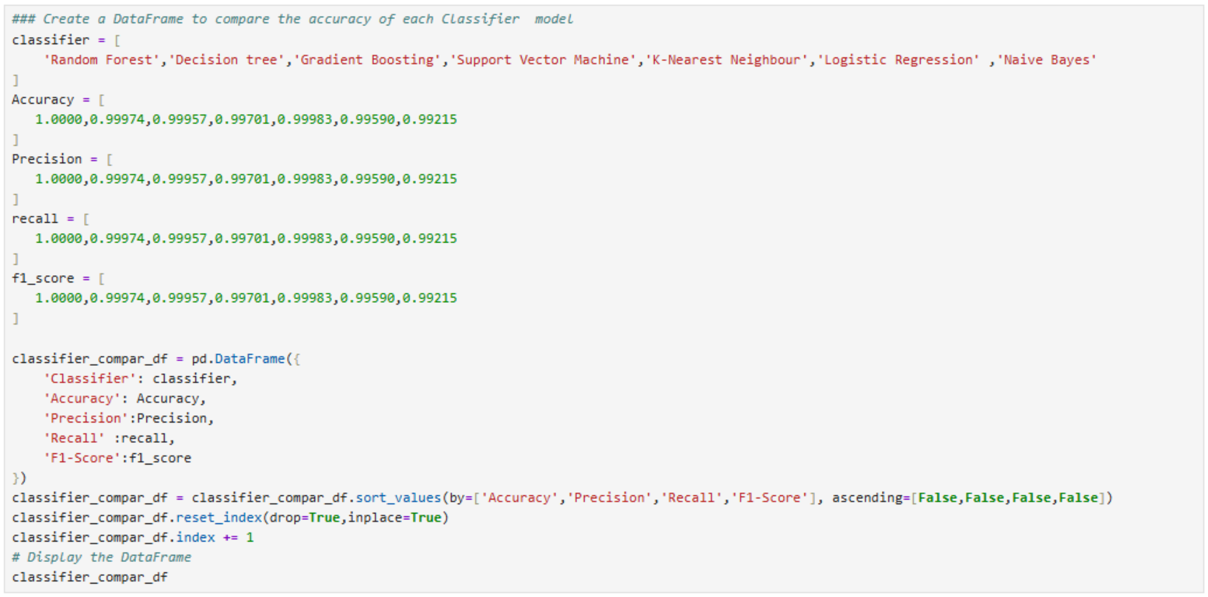
```
    # Print the results for K-Fold Cross-Validation
    print('Result For each fold:')
    print('Accuracy  - {}'.format(acc_scores))
    print('Precision - {}'.format(precision_scores))
    print('Recall  - {}'.format(recall_scores))
    print('F1-Score - {}'.format(f1_scores))
    print('Average Scores:')
    print(' Accuracy : {}'.format(avg_acc_score))
    print(' Precision : {}'.format(avg_precision))
    print(' Recall : {}'.format(avg_recall))
    print(' F1-Score : {}'.format(avg_f1))

    stratified_splitter = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
        # Split the data into training and testing DataFrames using stratified split
    for train_index, test_index in stratified_splitter.split(X, Y):
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = Y.iloc[train_index], Y.iloc[test_index]
        scaler = MinMaxScaler()
        # Fit the features X_train to the [0, 1] range
        X_train = scaler.fit_transform(X_train)
        # Transform the X_test features to the [0, 1] range
        X_test = scaler.transform(X_test)

    # Train the model on the entire training dataset
    start_time_test_train = time.time()
    model.fit(X_train, y_train)
    end_time_test_train = time.time()
    test_train_time = (end_time_test_train - start_time_test_train)/len(y_train)

    # Measure the prediction time for test data
    start_time_test = time.time()
    test_predicted = model.predict(X_test)
    end_time_test = time.time()
    test_prediction_time = (end_time_test - start_time_test)/len(y_test)
```



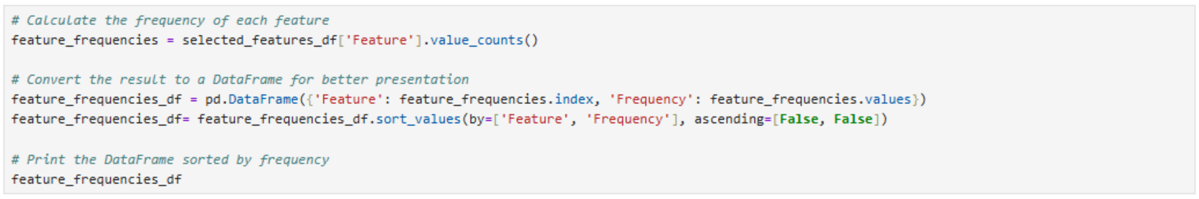
```python
# Calculate evaluation metrics for the test data
test_accuracy = accuracy_score(y_test, test_predicted)
test_precision = precision_score(y_test, test_predicted, average='weighted')
test_recall = recall_score(y_test, test_predicted, average='weighted')
test_f1 = f1_score(y_test, test_predicted, average='weighted')

# Calculate the confusion matrix for the test data
test_cm = confusion_matrix(y_test, test_predicted)
group_names = ['True Negative','False Positive','False Negative','True Positive']

group_counts = ["{0:0.0f}".format(value) for value in
                test_cm.flatten()]

group_percentages = ["{0:.2%}".format(value) for value in
                     test_cm.flatten()/np.sum(test_cm)]

labels = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in
          zip(group_names,group_counts,group_percentages)]

labels = np.asarray(labels).reshape(2,2)

ax = sns.heatmap(test_cm, annot=labels, fmt='', cmap='rocket')

ax.set_title(f'\n {model_name} Confusion Matrix of test data \n\n');
ax.set_xlabel('\nPredicted Values')
ax.set_ylabel('Actual Values ');

## Ticket labels - List must be in alphabetical order
ax.xaxis.set_ticklabels(['False','True'])
ax.yaxis.set_ticklabels(['False','True'])

## Display the visualization of the Confusion Matrix.
plt.show()

# Print the evaluation metrics for the test data
print("\nEvaluation on holdout test set:")
print(f"Accuracy : {test_accuracy:.5f}")
print(f"Precision : {test_precision:.5f}")
print(f"Recall : {test_recall:.5f}")
print(f"F1 Score : {test_f1:.5f}")

# Print the confusion matrix for the test data
print(f"\n {model_name} Confusion Matrix on test data:")
print(test_cm)
```

Similarly the metrics evaluation is done for all the classifiers including Random Forest (RF), Gradient Boosting (GB), Decision Tree (DT), Support Vector Machine (SVM), Logistic Regression (LR), K-Nearest Neighbors (KNN), Naive Bayes (NB).

3.2.2 Identifying the classifier with the best metrics



```
### Create a DataFrame to compare the accuracy of each Classifier  model
classifier = [
    'Random Forest','Decision tree','Gradient Boosting','Support Vector Machine','K-Nearest Neighbour','Logistic Regression' ,'Naive Bayes'
]
Accuracy = [
   1.0000,0.99974,0.99957,0.99701,0.99983,0.99590,0.99215
]
Precision = [
   1.0000,0.99974,0.99957,0.99701,0.99983,0.99590,0.99215
]
recall = [
   1.0000,0.99974,0.99957,0.99701,0.99983,0.99590,0.99215
]
f1_score = [
   1.0000,0.99974,0.99957,0.99701,0.99983,0.99590,0.99215
]

classifier_compar_df = pd.DataFrame({
    'Classifier': classifier,
    'Accuracy': Accuracy,
    'Precision':Precision,
    'Recall' :recall,
    'F1-Score':f1_score
})
classifier_compar_df = classifier_compar_df.sort_values(by=['Accuracy','Precision','Recall','F1-Score'], ascending=[False,False,False,False])
classifier_compar_df.reset_index(drop=True,inplace=True)
classifier_compar_df.index += 1
# Display the DataFrame
classifier_compar_df
```

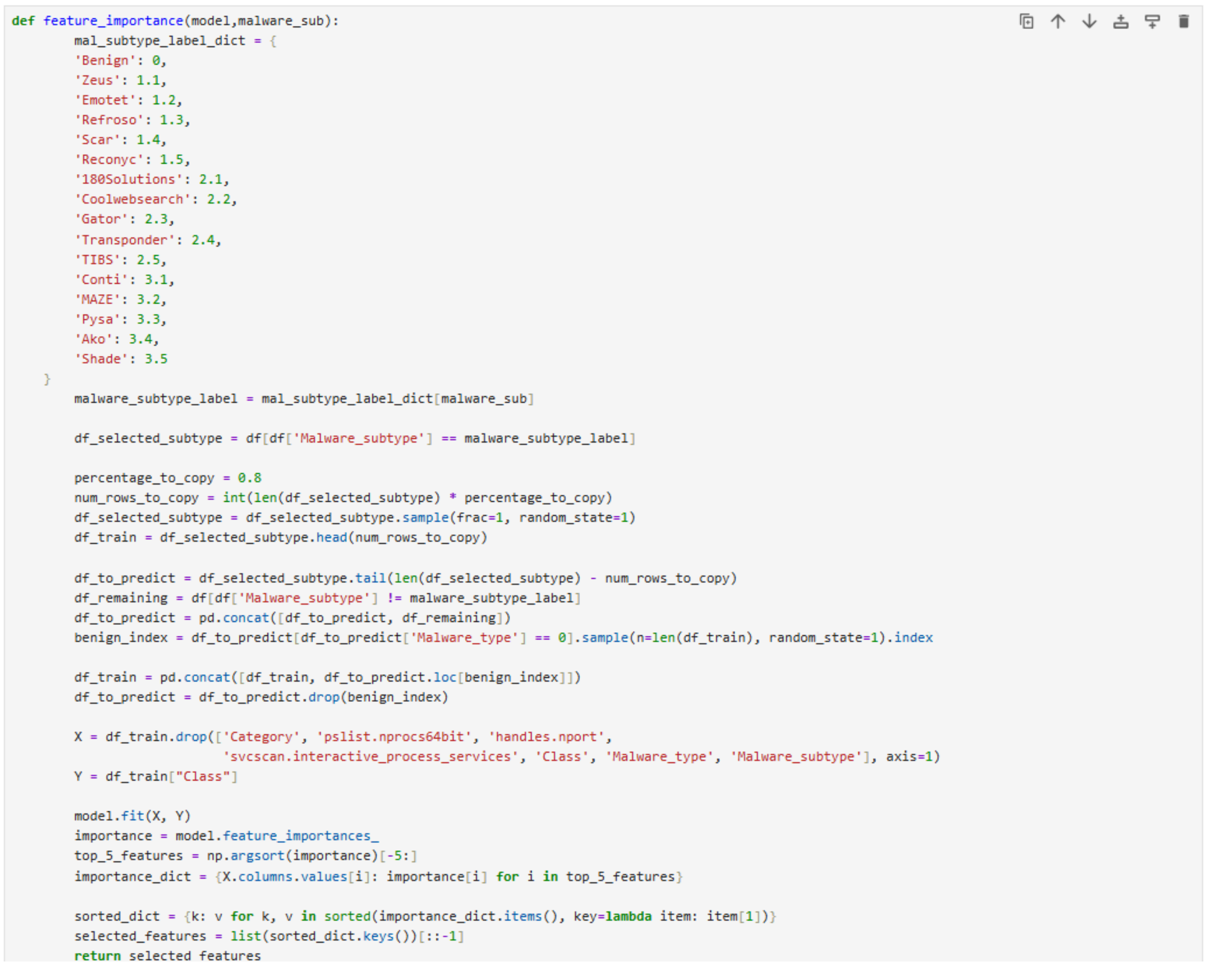


```
# Calculate the frequency of each feature
feature_frequencies = selected_features_df['Feature'].value_counts()

# Convert the result to a DataFrame for better presentation
feature_frequencies_df = pd.DataFrame({'Feature': feature_frequencies.index, 'Frequency': feature_frequencies.values})
feature_frequencies_df= feature_frequencies_df.sort_values(by=['Feature', 'Frequency'], ascending=[False, False])

# Print the DataFrame sorted by frequency
feature_frequencies_df
```

And by this analysis it was identified that the Random Forest classifer had the best accuracy compared to rest all of the algorithms

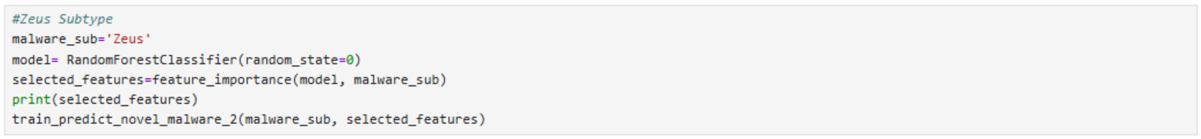
3.3 MODULE 2 : ADAPTIVE MALWARE DETECTION

3.3.1 Determining importance of each feature

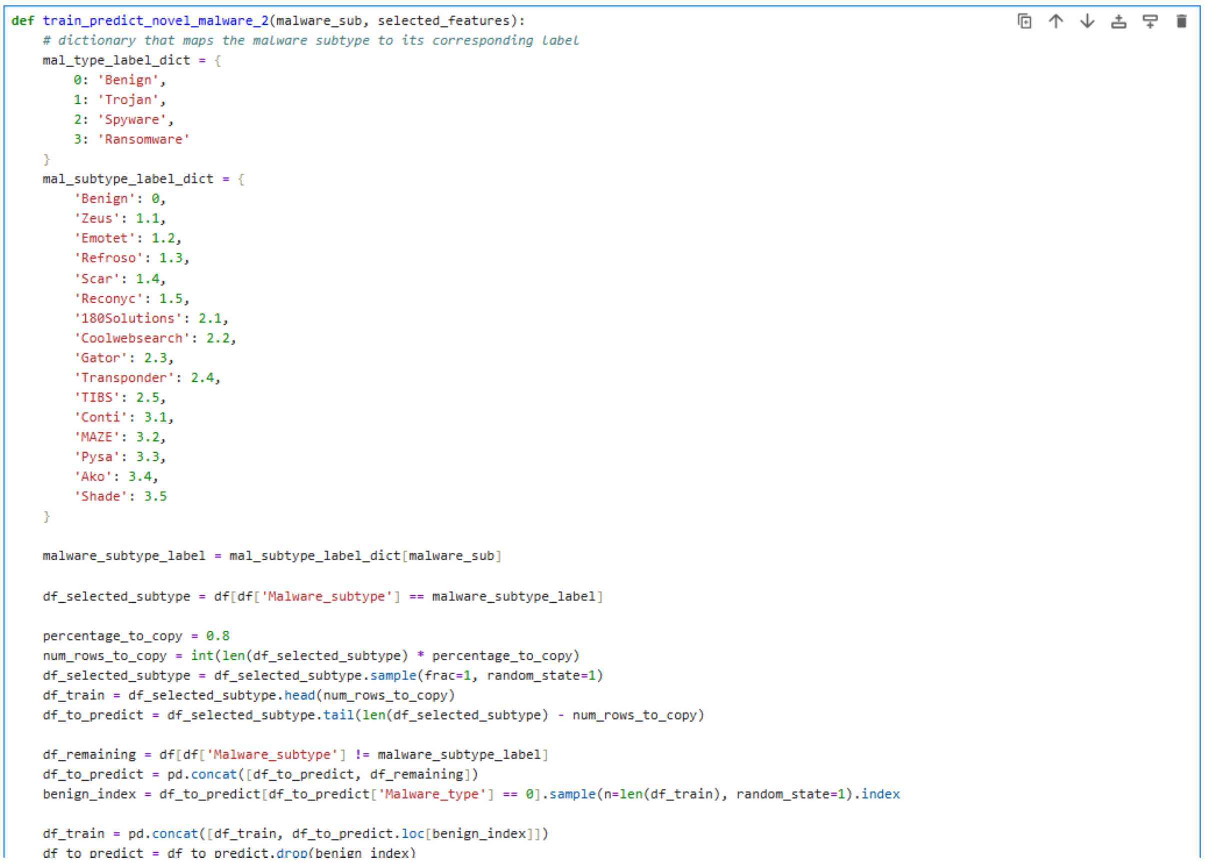


```
def feature_importance(model,malware_sub):
        mal_subtype_label_dict = {
        'Benign': 0,
        'Zeus': 1.1,
        'Emotet': 1.2,
        'Refroso': 1.3,
        'Scar': 1.4,
        'Reconyc': 1.5,
        '180Solutions': 2.1,
        'Coolwebsearch': 2.2,
        'Gator': 2.3,
        'Transponder': 2.4,
        'TIBS': 2.5,
        'Conti': 3.1,
        'MAZE': 3.2,
        'Pysa': 3.3,
        'Ako': 3.4,
        'Shade': 3.5
    }
        malware_subtype_label = mal_subtype_label_dict[malware_sub]

        df_selected_subtype = df[df['Malware_subtype'] == malware_subtype_label]

        percentage_to_copy = 0.8
        num_rows_to_copy = int(len(df_selected_subtype) * percentage_to_copy)
        df_selected_subtype = df_selected_subtype.sample(frac=1, random_state=1)
        df_train = df_selected_subtype.head(num_rows_to_copy)

        df_to_predict = df_selected_subtype.tail(len(df_selected_subtype) - num_rows_to_copy)
        df_remaining = df[df['Malware_subtype'] != malware_subtype_label]
        df_to_predict = pd.concat([df_to_predict, df_remaining])
        benign_index = df_to_predict[df_to_predict['Malware_type'] == 0].sample(n=len(df_train), random_state=1).index

        df_train = pd.concat([df_train, df_to_predict.loc[benign_index]])
        df_to_predict = df_to_predict.drop(benign_index)

        X = df_train.drop(['Category', 'pslist.nprocs64bit', 'handles.nport',
                           'svcscan.interactive_process_services', 'Class', 'Malware_type', 'Malware_subtype'], axis=1)
        Y = df_train["Class"]

        model.fit(X, Y)
        importance = model.feature_importances_
        top_5_features = np.argsort(importance)[-5:]
        importance_dict = {X.columns.values[i]: importance[i] for i in top_5_features}

        sorted_dict = {k: v for k, v in sorted(importance_dict.items(), key=lambda item: item[1])}
        selected_features = list(sorted_dict.keys())[::-1]
        return selected_features
```

3.3.2 Training and Testing against each malwate subtype



```
#Zeus Subtype
malware_sub='Zeus'
model= RandomForestClassifier(random_state=0)
selected_features=feature_importance(model, malware_sub)
print(selected_features)
train_predict_novel_malware_2(malware_sub, selected_features)
```



```
def train_predict_novel_malware_2(malware_sub, selected_features):
    # dictionary that maps the malware subtype to its corresponding label
    mal_type_label_dict = {
        0: 'Benign',
        1: 'Trojan',
        2: 'Spyware',
        3: 'Ransomware'
    }
    mal_subtype_label_dict = {
        'Benign': 0,
        'Zeus': 1.1,
        'Emotet': 1.2,
        'Refroso': 1.3,
        'Scar': 1.4,
        'Reconyc': 1.5,
        '180Solutions': 2.1,
        'Coolwebsearch': 2.2,
        'Gator': 2.3,
        'Transponder': 2.4,
        'TIBS': 2.5,
        'Conti': 3.1,
        'MAZE': 3.2,
        'Pysa': 3.3,
        'Ako': 3.4,
        'Shade': 3.5
    }

    malware_subtype_label = mal_subtype_label_dict[malware_sub]

    df_selected_subtype = df[df['Malware_subtype'] == malware_subtype_label]

    percentage_to_copy = 0.8
    num_rows_to_copy = int(len(df_selected_subtype) * percentage_to_copy)
    df_selected_subtype = df_selected_subtype.sample(frac=1, random_state=1)
    df_train = df_selected_subtype.head(num_rows_to_copy)
    df_to_predict = df_selected_subtype.tail(len(df_selected_subtype) - num_rows_to_copy)

    df_remaining = df[df['Malware_subtype'] != malware_subtype_label]
    df_to_predict = pd.concat([df_to_predict, df_remaining])
    benign_index = df_to_predict[df_to_predict['Malware_type'] == 0].sample(n=len(df_train), random_state=1).index

    df_train = pd.concat([df_train, df_to_predict.loc[benign_index]])
    df_to_predict = df_to_predict.drop(benign_index)
```

```python
X_train = df_train[selected_features]
Y_train = df_train["Class"]
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_train_scaled = pd.DataFrame(X_train_scaled, columns=scaler.feature_names_in_)
X_new = df_to_predict[selected_features]
Y_new = df_to_predict["Class"]
X_new_scaled = scaler.transform(X_new)
X_new_scaled = pd.DataFrame(X_new_scaled, columns = scaler.feature_names_in_)

rfc = RandomForestClassifier(random_state=0)
rfc.fit(X_train_scaled, Y_train)

prediction_times = []
for i in range(1000):
    start_time = time.time()
    y_pred_new = rfc.predict(X_new_scaled) #X_new = X variables from unseen dataset
    prediction_time = time.time() - start_time
    prediction_times.append(prediction_time)
print(f"Average prediction time: {np.mean(prediction_times)/len(y_pred_new)} seconds")
print(f"Standard Deviation of Prediction Time:{np.std(prediction_times)/len(y_pred_new)} seconds")

print(classification_report(Y_new, y_pred_new, target_names=['Benign', 'Malware'])) #Y_new = Y variables from unseen dataset

print(f"Accuracy: {np.mean(y_pred_new == Y_new):.5f}")
print(confusion_matrix(Y_new, y_pred_new))

cm_RF=confusion_matrix(Y_new, y_pred_new)
group_names = ['True Neg','False Pos','False Neg','True Pos']

group_counts = ["{0:0.0f}".format(value) for value in
                cm_RF.flatten()]

group_percentages = ["{0:.2%}".format(value) for value in
                     cm_RF.flatten()/np.sum(cm_RF)]

labels = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in
          zip(group_names,group_counts,group_percentages)]

labels = np.asarray(labels).reshape(2,2)

ax = sns.heatmap(cm_RF, annot=labels, fmt='', cmap='rocket')

ax.set_title(f'Random Forest Confusion Matrix: Unseen Dataset after training with only 80% of {malware_sub} with {len(X_train.columns)} Features \n\
ax.set_xlabel('\nPredicted Values')
ax.set_ylabel('Actual Values \n');
```

```python
sm = cm.ScalarMappable(cmap=colormap, norm=norm)
sm.set_array([])
cbar = plt.colorbar(sm, ax=ax)
cbar.set_label('Accuracy')
for bar in bars:
    height = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, height+0.001, f'{height:.4f}', ha='center', va='bottom',fontweight='bold')
# Add labels and title
ax.set_xlabel('Malware Type')
ax.set_ylabel('Accuracy')
ax.set_title(f"Accuracy of prediction per malware type using the {malware_sub} model\n\n")
plt.xticks(rotation=0)
plt.ylim(0.0, 1.0)  # Set the y-axis limits to focus on the accuracy range
plt.show()

plt.figure(figsize=(25,5))
Acc_per_subtype = df_to_predict[['Malware_subtype', 'Accuracy']].groupby('Malware_subtype')['Accuracy'].mean()
inv_mal_subtype_label_dict = {v: k for k, v in mal_subtype_label_dict.items()}
Acc_per_subtype.index = Acc_per_subtype.index.map(inv_mal_subtype_label_dict)
norm = mcolors.Normalize(vmin=0.9, vmax=1.0)
colors = [colormap(norm(value)) for value in Acc_per_subtype]

fig, ax = plt.subplots(figsize=(20,5))
bars = ax.bar(Acc_per_subtype.index, Acc_per_subtype, color=colors)

sm = cm.ScalarMappable(cmap=colormap, norm=norm)
sm.set_array([])
cbar = plt.colorbar(sm, ax=ax)
cbar.set_label('Accuracy')
for bar in bars:
    height = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, height, f'{height:.4f}', ha='center', va='bottom',fontweight='bold')
    #ax.text(bar.get_x() + bar.get_width()/2, height/2, f'{height:.4f}', ha='center', va='bottom',color ='black',fontsize=10,fontweight='bold')
# Add labels and title
ax.set_xlabel('Malware Type')
ax.set_ylabel('Accuracy')
ax.set_title(f"Accuracy of prediction per malware subtype using the {malware_sub} model\n\n")
plt.xticks(rotation=90)
plt.ylim(0.0, 1.0)  # Set the y-axis limits to focus on the accuracy range
plt.show()
return Acc_per_type, Acc_per_subtype
```

3.3.3 Plotting accuracy per sub type

```
def plot_accuracy_per_type(Acc_per_type, title):
    colormap = matplotlib.colormaps.get_cmap('viridis')

    norm = mcolors.Normalize(vmin=0.8, vmax=1.0)

    colors = [colormap(norm(value)) for value in Acc_per_type]

    fig, ax = plt.subplots(figsize=(10,7))
    bars = ax.bar(Acc_per_type.index, Acc_per_type, color=colors)

    sm = cm.ScalarMappable(cmap=colormap, norm=norm)
    sm.set_array([])
    cbar = plt.colorbar(sm, ax=ax)
    cbar.set_label('Accuracy')

    ax.set_xlabel('Malware Type')
    ax.set_ylabel('Accuracy')
    ax.set_title(f"Accuracy of prediction per malware subtype using the {malware_sub} model")
    plt.xticks(rotation=90)
    plt.ylim(0.5, 1.0)  # Set the y-axis limits to focus on the accuracy range
    plt.show()
```

3.4 MODULE 3 : MODEL EXPLAINABILITY

3.4.1 Prerequsites

```
import shap
from shap.plots import beeswarm
from shap.plots import bar
from wordcloud import WordCloud
shap.initjs()
rfc = RandomForestClassifier(random_state=0)
selected_features1=feature_importance(rfc, 'Transponder')
selected_features1
```

3.4.2 SHAP Force plot – Local Interpretation

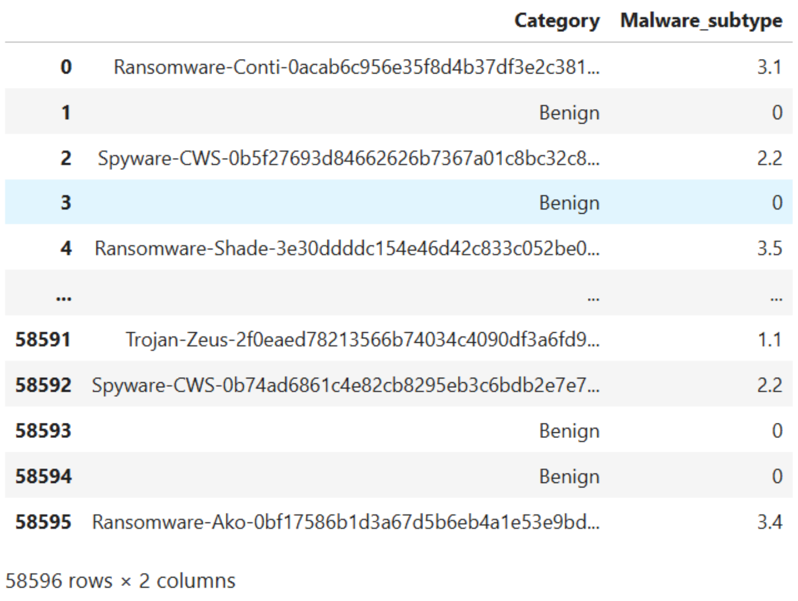
```
shap.initjs()
shap.summary_plot(
    shap_values=shap_values[:,:, 0],  # SHAP values for the specified class "0==class benign"
    features=x_test[:],  # Reshaped feature matrix
    feature_names=x_test.columns  # Feature names
)
```

3.4.3 SHAP Summary plot – Global Interpretation

```
shap.initjs()
force_plot=shap.plots.force(
    explainer.expected_value[0],
    shap_values=shap_values[index_benign][:,0],  # SHAP values for the specified class
    feature_names=x_test.columns,  # Feature names
)
shap.save_html("force_plot_benign1.html", force_plot)
from IPython.display import IFrame
IFrame('force_plot_benign1.html', width=1000, height=200)
```

CHAPTER 4
SNAPSHOTS WITH EXPLANATIONS

4.1 Data Preprocessing and Label encoding Data Preprocessing



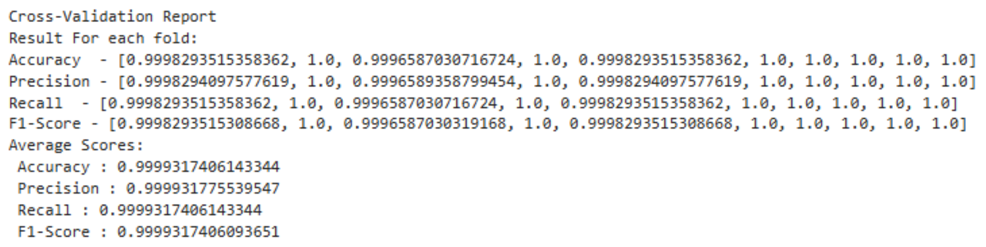
	Category	Malware_subtype
0	Ransomware-Conti-0acab6c956e35f8d4b37df3e2c381...	3.1
1	Benign	0
2	Spyware-CWS-0b5f27693d84662626b7367a01c8bc32c8...	2.2
3	Benign	0
4	Ransomware-Shade-3e30ddddc154e46d42c833c052be0...	3.5
...	...	...
58591	Trojan-Zeus-2f0eaed78213566b74034c4090df3a6fd9...	1.1
58592	Spyware-CWS-0b74ad6861c4e82cb8295eb3c6bdb2e7e7...	2.2
58593	Benign	0
58594	Benign	0
58595	Ransomware-Ako-0bf17586b1d3a67d5b6eb4a1e53e9bd...	3.4

58596 rows × 2 columns

Fig 4.1. Data Visalization after preprocessing

The above figure displays a snapshot of the dataset after preprocessing, specifically showing how each record has been cleaned, shuffled, and labeled for machine learning analysis. The dataset, sourced from CIC-MalMem-2022, contains 58,596 records evenly split between malware and benign samples. During preprocessing, textual class labels are converted to numeric codes (0 for benign, 1 for malware), and each malware instance is further labeled by type (Trojan, Spyware, Ransomware) and subtype (e.g., Zeus as 1.1, Transponder as 2.4). Invariant features (those with the same value across all samples) are removed to ensure only informative features remain.

4.2 Assessing Baseline Classifier Performance Against Known Malware Variants



```
Cross-Validation Report
Result For each fold:
Accuracy  - [0.9998293515358362, 1.0, 0.9996587030716724, 1.0, 0.9998293515358362, 1.0, 1.0, 1.0, 1.0, 1.0]
Precision - [0.9998294097577619, 1.0, 0.9996589358799454, 1.0, 0.9998294097577619, 1.0, 1.0, 1.0, 1.0, 1.0]
Recall  - [0.9998293515358362, 1.0, 0.9996587030716724, 1.0, 0.9998293515358362, 1.0, 1.0, 1.0, 1.0, 1.0]
F1-Score - [0.9998293515308668, 1.0, 0.9996587030319168, 1.0, 0.9998293515308668, 1.0, 1.0, 1.0, 1.0, 1.0]
Average Scores:
 Accuracy : 0.9999317406143344
 Precision : 0.999931775539547
 Recall : 0.9999317406143344
 F1-Score : 0.9999317406093651
```

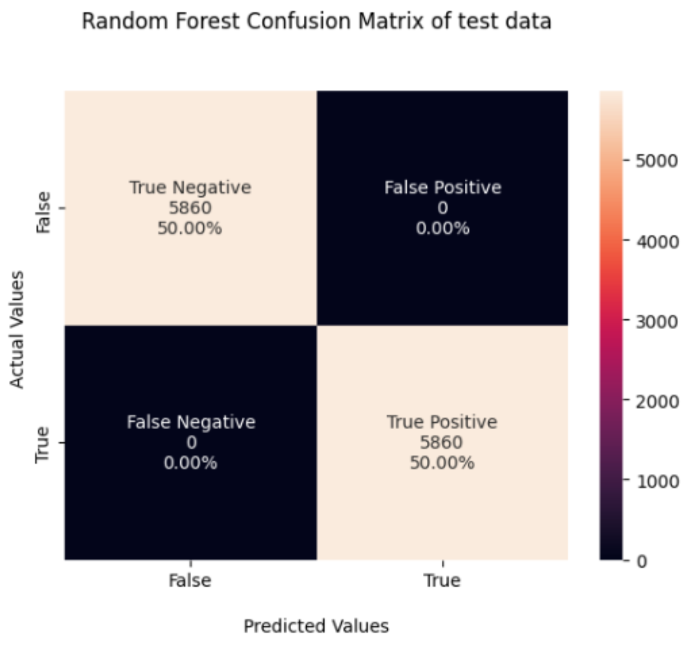


Fig 4.2. Confusion matrix for the Random Forest-based Malware Detection Model post the Baseline Classification and Evaluation

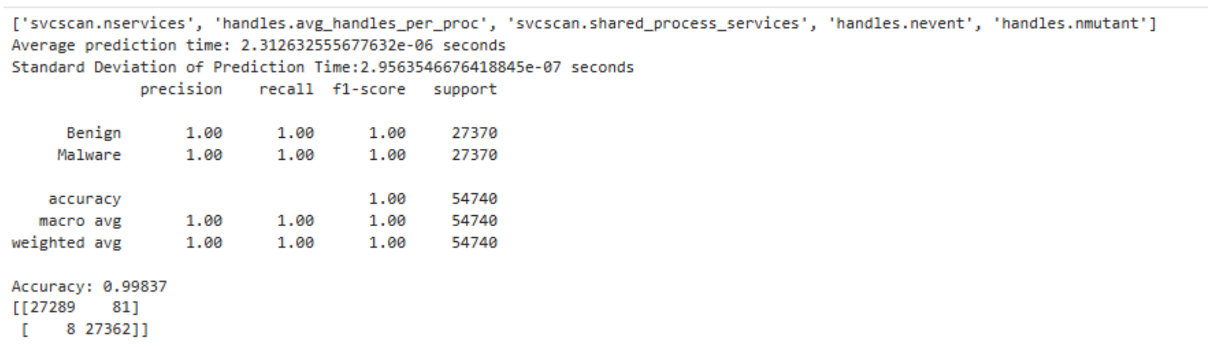
The above confusion matrix visualizes the performance of the Random Forest-based malware detection model after baseline classification and evaluation. The matrix shows how well the model distinguishes between benign and malware samples: the diagonal cells represent correct predictions (true negatives for benign and true positives for malware) and the off-diagonal cells show misclassifications (false positives and false negatives). In this classification, the confusion matrix reveals near-perfect classification, with almost all benign and malware instances correctly identified-only a negligible number of samples are misclassified, as indicated by the very low percentages in the off-diagonal cells.

4.3 Baseline Classification Result

Table 4.1. Binary classification results for models tested on previously encountered malware subtypes.

	Classifier	Accuracy	Precision	Recall	F1-Score
1	Random Forest	1.0000	1.0000	1.0000	1.0000
2	K-Nearest Neighbour	0.9998	0.9998	0.9998	0.9998
3	Decision tree	0.9997	0.9997	0.9997	0.9997
4	Gradient Boosting	0.9996	0.9996	0.9996	0.9996
5	Support Vector Machine	0.9970	0.9970	0.9970	0.9970
6	Logistic Regression	0.9959	0.9959	0.9959	0.9959
7	Naive Bayes	0.9922	0.9922	0.9922	0.9922

4.4 Adaptive Malware Detection and Feature Selection



```
['svcscan.nservices', 'handles.avg_handles_per_proc', 'svcscan.shared_process_services', 'handles.nevent', 'handles.nmutant']
Average prediction time: 2.312632555677632e-06 seconds
Standard Deviation of Prediction Time:2.9563546676418845e-07 seconds
              precision    recall  f1-score   support

      Benign       1.00      1.00      1.00     27370
     Malware       1.00      1.00      1.00     27370

    accuracy                           1.00     54740
   macro avg       1.00      1.00      1.00     54740
weighted avg       1.00      1.00      1.00     54740

Accuracy: 0.99837
[[27289    81]
 [    8 27362]]
```

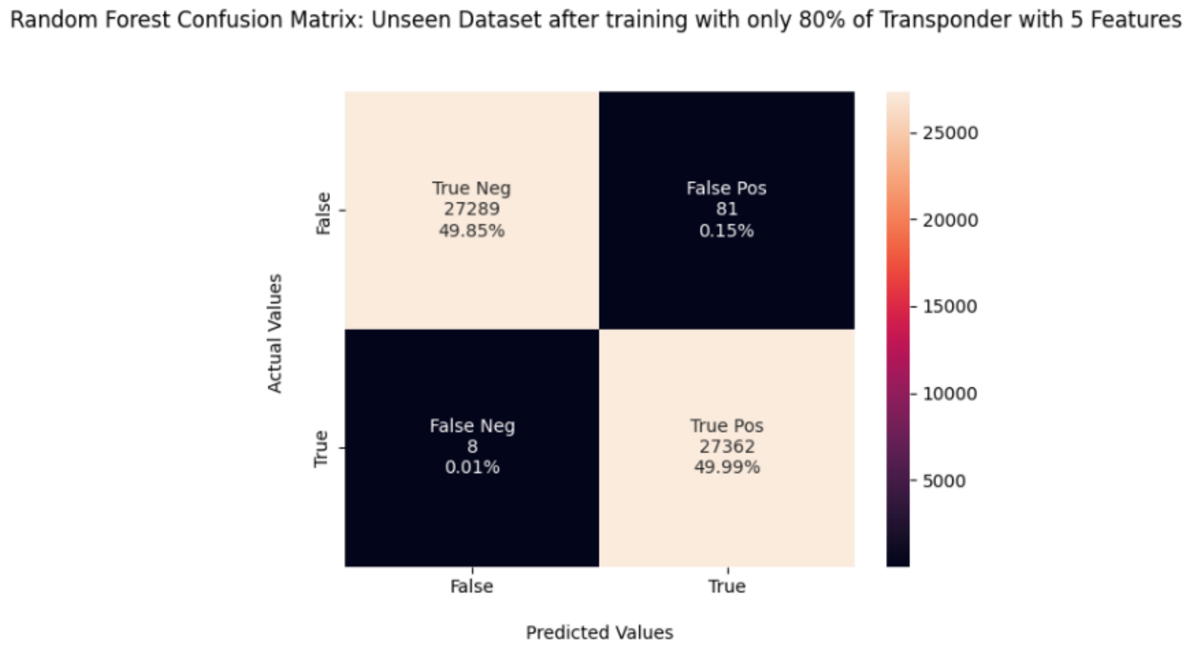


Fig 4.3. Confusion matrix for the Random Forest-based malware detection model post the train-train and validation procedure

Table 4.1 and Fig. 4.3 together highlight the superior performance and reliability of the Random Forest classifier for malware detection. Table 4.1 presents a comparison of seven machine learning models-Random Forest, K-Nearest Neighbour, Decision Tree, Gradient Boosting, Support Vector Machine, Logistic Regression, and Naive Bayes-evaluated on binary classification of malware and benign files. Random Forest achieved perfect scores across all metrics (accuracy, precision, recall, F1-score = 1.0000), outperforming the others, with even the lowest performer (Naive Bayes) maintaining over 99% accuracy. Fig. 4.3 displays the confusion matrix for the Random Forest model after training and validation, showing that the model correctly classified nearly all benign and malware samples, with a false positive rate of only 0.15% and a false negative rate of just 0.01%.

Table 4.2. Feature Selection for each Malware Subtype with their importance value

Malware_Subtype	Feature	Importance
Transponder	handles.mutant	0.07566
Transponder	handles.nevent	0.07696
Transponder	svcscan.shared_process_services	0.09663
Transponder	handles.avg_handles_per_proc	0.11051
Transponder	svcscan.nservices	0.13703

Table 4.2 presents the top five most important features selected for each malware subtype by the Random Forest model, along with their corresponding importance values. For example, for the Transponder subtype, the most influential features are svcscan.nservices (number of services running), handles.avg_handles_per_proc (average number of handles per process), svcscan.shared_process_services (number of services in shared processes), handles.nevent (number of event handles), and handles.mutant (number of mutant handles), each with a specific importance score indicating its contribution to classification accuracy.

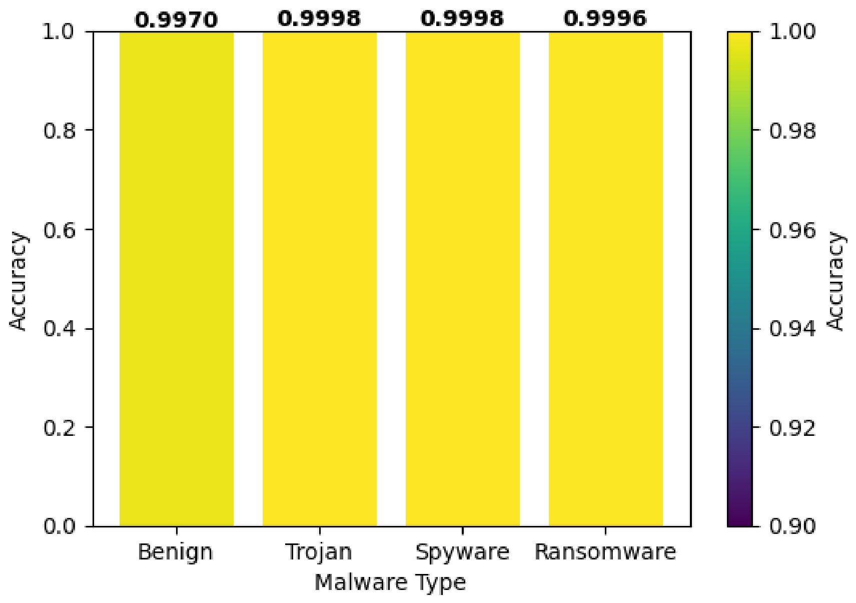


Fig 4.4. Accuracy of prediction per Malware type using Transponder Model

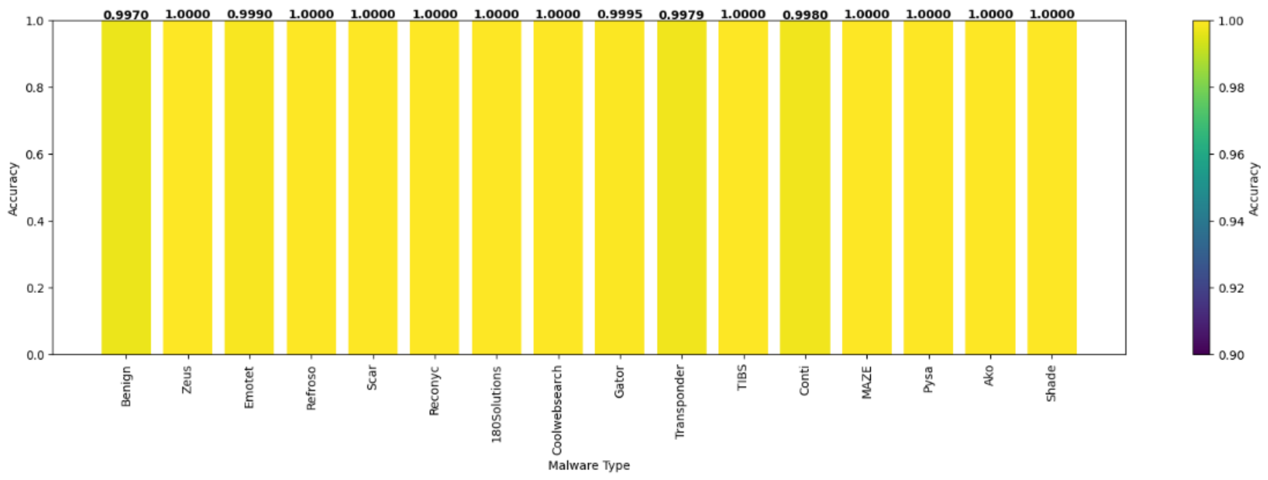


Fig 4.5. Accuracy of prediction per Malware sub-type using Transponder Model

Fig. 4.4 and Fig. 4.5 together illustrate how well the Random Forest model, trained only on the Transponder malware subtype and using just the top five most important features, generalizes to detect other malware types and subtypes. Fig. 4.4 shows the prediction accuracy for each main malware type-Trojan, Spyware, and Ransomware-when evaluated using the Transponder-trained model, with all types achieving very high accuracy, typically above 98%.

Fig. 4.5 breaks this down further, displaying the accuracy for each individual malware subtype (such as Zeus, Emotet, Conti, Shade, etc.). These results demonstrate the model's adaptability and robustness: even when trained on data from just one subtype, it can reliably detect a wide variety of unseen and obfuscated malware families, making it highly effective for real-world, zero-day malware detection scenarios.

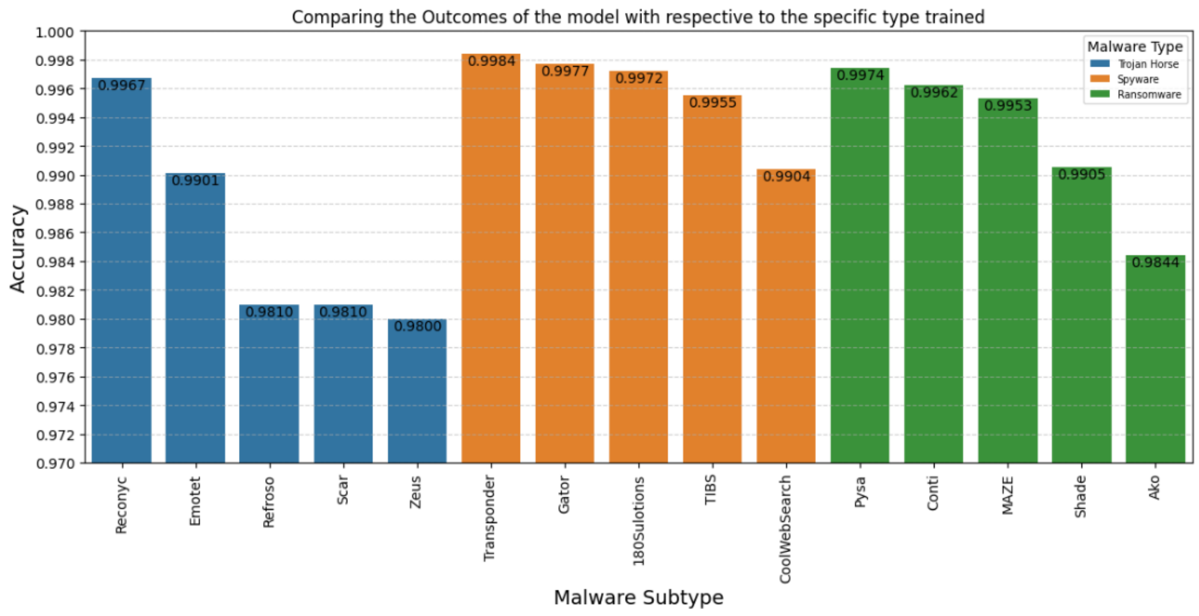


Fig 4.6. Comparison of accuracies of the different malware subtype-based models

4.5 Model explainability

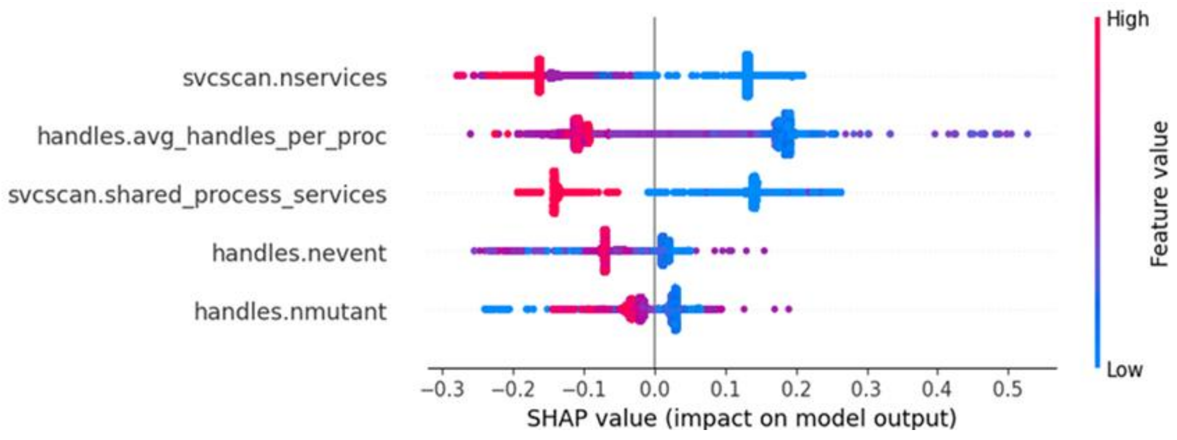


Fig 4.7. Global interpretation using SHAP Beeswarm plot of the top five features in the Transponder-based malware detection model

Fig. 4.7 shows a SHAP Beeswarm plot, which visually explains how the top five features influence the Random Forest model's malware predictions for the Transponder subtype. Each dot represents a file, with its position on the X-axis indicating whether a feature pushes the prediction toward "malware" or "benign." Features are listed by importance on the Y-axis, and color shows feature value (red for high, blue for low).

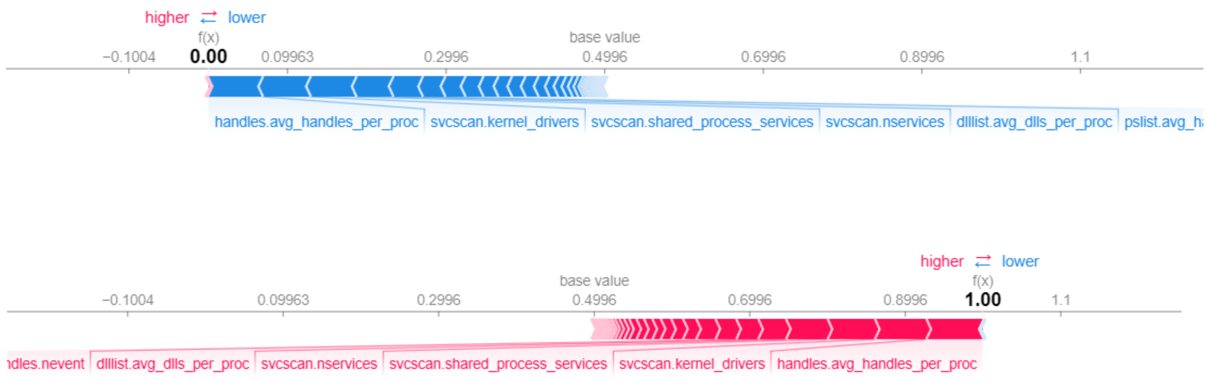


Fig 4.8. Local explanations using SHAP force plot for individual prediction cases

Fig. 4.8 presents local explanations for individual predictions made by the Random Forest-based malware detection model using SHAP force plots. In these visualizations, each feature's contribution to a specific prediction is shown as a colored arrow: blue arrows indicate features pushing the prediction toward the benign class, while red arrows push it toward malware. For example, in a benign file, features like svcscan.shared_process_services, svcscan.nservices, have strong blue arrows, meaning their values decrease the likelihood of a malware prediction. The length and direction of each arrow reflect the magnitude and influence of each feature for that particular file. This local interpretability enables security analysts to understand exactly why the model made a specific classification, providing transparency and actionable insights for incident response and forensic investigations.

```
================ Random Forest Malware Prediction ================

Choose prediction mode:
1. Manual input of feature values
2. Use instance from dataset (by entering index)
Enter your choice (1 or 2):  1

Manual Mode: Please input values for the top 5 features (between 0 and 1)
Enter value for 'svcscan.nservices' (0 to 1):  0.1
Enter value for 'handles.avg_handles_per_proc' (0 to 1):  1
Enter value for 'svcscan.shared_process_services' (0 to 1):  0.5
Enter value for 'handles.nevent' (0 to 1):  0.2
Enter value for 'handles.nmutant' (0 to 1):  0.1

✅ Prediction: The given instance is classified as: *Benign*

Visualizing input feature values and their impact...
```

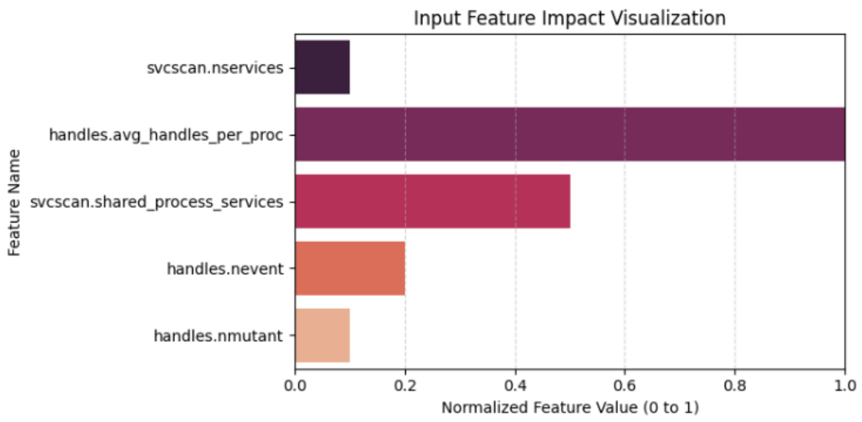


Fig 4.9. Visualizing input feature values and their impact for classification

Fig. 4.9 visualizes how the values of the top input features influence the Random Forest model's malware classification decision for individual files. In this plot, each feature's actual value for a given sample is displayed alongside its impact on the predicted outcome, highlighting which attributes most strongly push the prediction toward either "malware" or "benign". Features with their specific values for a test instance, and their corresponding SHAP values indicate whether they increase or decrease the likelihood of being classified as malware. This detailed visualization bridges the gap between raw feature data and model output, making the decision process transparent and interpretable. Security analysts can use this insight to understand exactly why a file was flagged, which is crucial for trust and effective response in real-world malware detection scenarios

CHAPTER 5

5.1 CONCLUSION

In this project, we have presented a novel machine learning-based system for detecting new and obfuscated malware attacks. The model was trained on a small dataset from a single malware subtype—Transponder—and achieved a state-of-the-art accuracy(99.84%), while maintaining high processing speeds (5.7 μs per file) which depends on the configuration of respective system as well and minimal memory usage (a model size of 340 KB) while predicting the target samples. These findings improve the generaliztion and accuracy in malware detection and also highlight the critical need for the testing and refinement of machine learning-based solutions for detecting previously unseen malware variants. This is a crucial defence solutions for ensuring that our can respond to the continuously evolving landscape of cyber threats. Malware detection is very important in the cybersecurity domain, especially with the increasing number of obfuscation techniques and this model solves exactly that problem.

5.2 FUTURE PLANS

To improve the model's explainability, future work could incorporate error analysis where explanation for when and why the model makes errors in specific cases are included. This approach would provide information about the model's weaknesses and guide more improvements. Also, future research may focus on developing a similar model on other obfuscated or non-obfuscated malware datasets. This would test the generalisability of our model across different types of malware and identify which types of attacks are more amenable to zero-shot learning. Finally, training on one dataset and testing on a different dataset could provide further validation of our model's adaptability and robustness.

CHAPTER 6
REFERENCES

[1] **Shahid Alam, R.Nigel Horspool , Issa Traore, Ibrahim Sogukpinar,** "A framework for metamorphic malware analysis and real-time detection" , *Computers & Security, Volume 48, February 2015.*

[2] **Acuto Alberto, Jack D., Maskell Simon,** "Defending the unknown: Exploring reinforcement learning agents' deployment in realistic, unseen networks", *CEUR Workshop Proceedings, Volume 3652, Pages 22 – 35, October 2023.*

[3] **Mohammed M. Alani, Atefeh Mashatan, Ali Miri,** "XMal: A lightweight memory-based explainable obfuscated-malware detector". *Computers & Security Volume 133, October 2023, 103409.*

[4] **Alazab, M., Venkatraman, S., Watters, P. A., & Alazab, M.,** "Zero-day malware detection based on supervised learning algorithms of API call signatures.", *AusDM, 11, 171–182.*

[5] **Alduailij, M., Khan, Q. W., Tahir, M., Sardaraz, M., Alduailij, M., & Malik, F.,** "Machine-learning-based DDoS attack detection using mutual information and random forest feature importance method", *Symmetry, 14(6),1–15.(2022).*

CHAPTER 7

APPENDIX – BASE PAPER

Title	:	Detecting new obfuscated malware variants: A lightweight and interpretable machine learning approach
Author	:	Oladipo A. Madamidola, Felix Ngobigha, Adnane Ez-zizi
Publisher	:	Elsevier
Year	:	2025
Journal	:	Intelligent Systems with Applications
Indexing	:	Scopus
Base paper URL	:	https://www.sciencedirect.com/science/article/pii/S2667305324001467