UNIONE EUROPEA
Fondo Sociale Europeo

PON RICERCA E INNOVAZIONE

UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

UNIVERSITA' DEGLI STUDI DI BARI ALDO MORO

DEPARTMENT OF COMPUTER SCIENCE

PHD PROGRAM IN COMPUTER SCIENCE AND MATHEMATICS

XXXVII CYCLE

SCIENTIFIC DISCIPLINARY AREA 09/IINF-05

AI-based Methods for Generating Adversarial and Synthetic Data in Cybersecurity

Dottorando: Dott. Muhammad Imran

Muhammad Imran
27.05.2025 14:00:11 GMT+02:00

Coordinator: Prof. Francesca MAZZIA

FRANCESCA MAZZIA
27.05.2025 18:39:47 GMT+02:00

Supervisor: Prof. Annalisa APPICE

ANNALISA APPICE
27.05.2025 15:31:18 GMT+02:00

Co-supervisor: Prof. Donato MALERBA

DONATO MALERBA
27.05.2025 14:39:29 GMT+02:00

Final Exam 2025

The fellowship was supported by DM 1061 del 10 agosto 2021, Dottorati PON - Bando 2021 - Ciclo 37 (XXXVII), Azione IV.4 - Dottorati e contratti di ricerca su tematiche dell'innovazione, CUP H99J21010060001, Codice: DOT1302947-1

KDDE Research Laboratory

Department of Computer Science

University of Bari ”Aldo Moro”

Abstract

During the last decade, the cybersecurity literature has conferred a high-level role to the Artificial Intelligence (AI) as a powerful security paradigm to recognise malicious software in modern anti malware systems. However, a non-negligible limitation of AI methods used to train decision models is that adversarial attacks can easily fool them. Adversarial attacks are attack samples produced by carefully manipulating the samples at the test time to violate the model integrity by causing detection mistakes. In this thesis, we analyse the performance of five realistic, literature, target-based adversarial attacks, namely Extend, Full DOS, Shift, FGSM padding + slack and GAMMA, against two AI-based, state-of-the-art models, namely MalConv and LGBM, commonly used to recognise Windows Portable Executable (PE) malware files. Specifically, MalConv is a Convolutional Neural Network (CNN) model learned from the raw bytes of Windows PE files. LGBM is a Gradient-Boosted Decision Tree model learned from engineered features extracted through the static analysis of Windows PE files performed with the LIEF library. Notably, the attack methods and AI models considered in this thesis are state-of-the-art methods broadly used in the AI literature for Windows PE malware detection tasks. In addition, we carry out an exploratory study that uses both distance analysis and SHAP analysis to explain how the considered adversarial attack methods change Windows PE malware to fool the evaluated decision models.

In addition, we explore the performance of the adversarial training strategy as a means to secure effective decision models against adversarial Windows PE malware files generated with the considered attack methods. We explain how GAMMA can actually be considered the most effective evasion method according to the performed comparative analysis and the adversarial training strategy can actually help in recognising adversarial Windows PE malware generated with GAMMA by also explaining how it changes model decisions. In addition, we generate the tabular synthetic data (TSD) with CTGAN and Focal-AuxCTGAN (Pre-training and Joint-training) and evaluated the utility of TSD generated for several classification tasks comprising problems of network intrusion detection and malware classification. Achieved results show that data-specific characteristics such as, number of classes, class distribution and features type may have an effect on the utility performance of both TSD methods.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
DT	Decision Tree
CART	Classification and Regression Tree
ID3	Iterative Dichotomiser 3
InfoG	Information Gain
OOB	Out-of-Bag
LightGBM	Light Gradient Boosting Machine
EFB	Exclusive Feature Bundling
GOSS	Gradient-based One-Sided Sampling
LR	Linear Regression
LGR	Logistic regression
NB	Naïve Bayes
RF	Random Forest
KNN	K-Nearest Neighbour
SVM	Support Vector Machine
MLP	Multi-layer Perceptron

ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent
L-BFGS	Limited Memory Broyden–Fletcher–Goldfarb–Shanno
Adam	Adaptive Moment Estimation
Tanh	Hyperbolic tangent
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
GAN	Generative Adversarial Networks
TPE	Tree-structured Parzen Estimator
AE	Autoencoder
XAI	Explainable Artificial Intelligence
LIME	Local Interpretable Model-agnostic Explanations
SHAP	SHapley Additive exPlanations
DALEX	moDel Agnostic Language for Exploration and eXplanation
PDP	Partial Dependency Plots
ALE	Accumulated Local Effects
LRP	Layer-wise Relevance Propagation
GRAD-CAM	Gradient-Weighted Class Activation Mapping
IG	Integrated Gradient
AML	Adversarial machine learning
FGSM	Fast Gradient Sign Method
BIM	Basic Iterative Method

PGD	Projected Gradient Descent
JSM	Jacobian Saliency map
ZOO	Zeroth Order Optimization
DNN	Deep neural network
RSE	Random Self-Ensemble
PixelDP	Pixel-level Differential Privacy
DP	Differential privacy
TSD	Tabular Synthetic Data
RTD	Real Tabular Data
EHR	Electronic Health Record
MSE	Mean Squared Error
IT-GAN	Invertible Tabular GAN
NODE	Neural Ordinary Differential Equations
WGAN-GP	Wasserstein Generative Adversarial Network with Gradient Penalty
CTGAN	Conditional Tabular GAN
VGM	variational Gaussian mixture model
PMF	probability mass function
CWGAN	Conditional Wasserstein GAN
PATE-GAN	Private Aggregation of Teacher Ensembles-Generative Adversarial Network
DP-GAN	Differentially Private Generative Adversarial Network
RMSProp	root mean square propagation
RDP-CGAN	Rényi Differential Privacy and Convolutional Generative Adversarial Networks
RDP	Renyi Differential Privacy

1D-CAE	one-dimensional convolutional autoencoder
PReLU	Parametric ReLU
DP-SGD	Differentially Private Stochastic Gradient Descent
Focal-AuxCTGAN	Focal Loss Auxiliary Conditional Tabular GAN
AUC-ROC	Area Under the Receiver Operating Characteristic Curve
ODML	Orange Data Mining Library
CD	Critical Difference
PE	Portable Executable
GUI	Graphical User Interface
LIEF	Library to Instrument Executable Formats
DLL	Dynamic-Link Libraries
ProcMon	Process Monitor
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
GIN	Graph Isomorphism Network
CFG	Control Flow Graph
GFE	Graph Feature Extraction
FCG	Function Call Graph
GDG	Graph Data Generation
GC	Graph Classification
SVD	Singular Value Decomposition
PSI	Printable String Information
CV	cross-validation

PCA	principal component analysis
ET	ExtraTrees
LOOCV	leave-one-out cross-validation
MSE	Mean Squared Error
MIM	Momentum Iterative Method
GBDT	Gradient Boosting Decision Tree
GAMMA	Genetic Adversarial Machine learning Malware Attack
SPM	sequential pattern mining
CloFast	Closed FAST
VMSP	Vertical mining of Maximal Sequential Patterns
ERMiner	Equivalence class based sequential Rule Miner
NBMT	Naive Bayes Multinomial Text
SGDT	Stochastic Gradient Descent Text
MCC	Matthews correlation coefficient
SMOTE	Synthetic Minority Oversampling Technique
NNI	Neural Network Intelligence
AutoML	Automated Machine Learning
HMM	Hidden Markov Model
PA	Passive-Aggressive
FPR	false positive rate
FNR	false negative rate
TPR	true positivve rate
LDA	Linear Discriminant Analysis

TF-IDF	Frequency–Inverse Document Frequency
BiLSTM	Bidirectional Long Short-Term Memory
FANN	Feature Attention-based Neural Network
CA	coordinate attention
DSC	depthwise separable convolution
GCE	global context embedding
PSO	Particle Swarm Optimization
GNN	Graph Neural Network
GINE	Graph Isomorphism Network
GAT	Graph Attention Network
FC	fully connected
FFNN	Feed-Forward Neural Network
RFE	Recursive Feature Elimination
MI	Mutual Information
URL	Uniform Resource Locator
CGAN	Conditional Generation Adversarial Nets

Chapter 1

Introduction

1.1 Research motivations

The fast growth of Artificial Intelligence (AI) has resulted in its widespread adoption in cybersecurity applications, particularly malware detection and intrusion detection. However, AI models are vulnerable to adversarial attacks, where carefully crafted malware files manipulate model decisions, resulting in classification of malware files as goodware. An attacker can craft Windows PE malware files while preserving functionalities to evade Windows PE malware detection systems. These vulnerabilities are serious concerns about the reliability of AI-based malware detectors in adversarial environments. Similarly, the effectiveness of AI-based cybersecurity systems depends on access to high-quality, diverse, and privacy-preserving datasets. Due to privacy regulations and security concerns, real-world cybersecurity datasets are often restricted or generated through controlled simulations, which may not fully capture the complexity of real-world threats. Tabular synthetic data (TSD) generation is a solution for training AI models. This thesis focuses on evaluating adversarial attacks and defence in Windows PE malware detection and producing TSD, particularly for cybersecurity applications in malware and intrusion detection.

1.1.1 Adversarial malware detection

Nowadays almost every person uses Internet daily where digital information is considered one of the most valuable assets for both persons and organizations. Due to the fast growth of the Internet, criminals often perform their crimes online rather than in the real world. Criminals use malicious software (malware) to launch cyber-attacks to steal confidential data/information, get financial benefits and, in general, to damage the target system. According to the scientific [1] and business [2] report, approximately 1 million malware files are created daily, and business report cybercrime damaged the world economy by approximately $10.5 trillion annually by

2025. Microsoft Windows is the most widely used operating system, making it a primary target for malware attacks. According to the statistics shared by the Kasperksy lab [3] at the end of 2021, Windows Portable Executable (PE) files represent the majority of malware threats, with more than 90% of daily malware detected. To protect the systems from malware attacks, a malware should be identified before it infects digital systems. In the last twenty years, AI has been proven beneficial to address complex cyber-threat detection problems, such as malware and intrusion detection. In particular, several AI studies have been recently published on Windows PE malware identification. However, effective malware detection is still a challenge because many new malware attacks occur every day, most of which are variations of previously known attacks. Hence, even advanced mechanisms face difficulty detecting these minor variations. Recent AI research studies revealed several vulnerabilities in machine learning and deep learning algorithms developed for malware detection. An attacker may take advantage of these vulnerabilities by producing an adversarial input to be incorrectly classified by machine learning and deep learning algorithms. In recent years, security researchers and practitioners have proposed numerious white-box and black-box adversarial attack methods that produce functionality-preserving adversarial Windows PE malware files, which are wrongly classified by the AI models as goodware. In addition, some research studies have proposed defensive methods against Windows PE adversarial attacks; however, Windows PE files still remain insufficiently protected to adversarial attacks. This motivates the need to systematically investigate both offensive attack methods that exploit the vulnerabilities of AI-based Windows PE malware detectors, and defensive strategies such as adversarial training aimed at improving the robustness of these detectors. A comprehensive evaluation is essential to develop a practically reliable AI-based Windows PE malware detection system in an adversarial environment.

The research described in this thesis evaluates the offensive capabilities of five state-of-the-art adversarial attack methods: Full DOS, Extend, Shift, FGSM (padding+slack), and GAMMA against Windows PE malware detectors, namely MalConv and LGBM. In addition, it investigates a defensive strategy based on adversarial training to improve the robustness of these detectors against such adversarial attacks. This approach provides valuable insights into enhancing the reliability of AI-driven cybersecurity systems, particularly within the context of Windows PE malware detection.

1.1.2 Synthetic data generation in cybersecurity

We are in an era of rising data generation, leading to a paradigm shift from manual processes to AI-based applications. However, several challenges affect the AI model development, such as data privacy, high collection costs, lack of labels, inherent biases in data, and limited data

availability. Synthetic data is artificial data that can be produced with AI systems to handle these challenges. The AI-based systems to produce synthetic data can learn the statistical distribution properties of the original data, such as variance, structure, and feature correlation. Synthetic data generation is not only focused on producing synthetic data but also requires evaluating the produced synthetic data considering fidelity, utility, and privacy dimensions. In synthetic data, achieving an optimal balance among fidelity, utility, and privacy is challenging, as an improvement in a dimension often affects the remaining dimensions.

In cybersecurity, AI has become important in addressing emerging cyber threats. However, obtaining high-quality cyber-attack datasets that accurately represent real-world attack scenarios is a challenge. Organizations cannot share their sensitive data because of data privacy regulations and security concerns. In addition, many cybersecurity datasets are generated through simulations, which often fail to capture the complexity and diversity of real-world attack scenarios. Another issue is the lack of labeled data, making it difficult to train supervised AI models effectively. Moreover, malware and network intrusions change over time, especially with the appearance of adversarial malware, requiring to mitigate overfitting during training. These constraints limit the ability to develop AI models capable of detecting cyber threats. Synthetic Data Generation (SDG) is a possible solution for producing privacy-preserving datasets that simulate real-world distributions while maintaining utility and statistical properties.

Tabular data is commonly collected in various domains, so TSD generation is a crucial task in several real scenarios. In particular, the TSD generation is crucial for cybersecurity applications of AI, such as network intrusion detection and malware identification, without compromising sensitive information. In this thesis, we focused the attention on TSD performed with a data quality-focused methodology named Focal-AuxCTGAN and defined for classification tasks. We evaluated the utility of Focal-AuxCTGAN for several classification problems comprising problems of network intrusion detection and malware classification. The TSD methodology evaluated in this thesis is able to handle mixed data types, data imbalance, and multi-classes during the TSD generation. In mixed data types, tabular data contains both numeric features (e.g., packet sizes in network traffic datasets or file sizes in malware datasets) and categorical features (e.g., protocol types in network traffic datasets or file extension in malware datasets). In data imbalanced problems, some classes are more frequent than other classes. For example, in intrusion detection datasets, normal traffic appears more frequently compared to several rare attack types such as Distributed Denial-of-Service (DDoS) attacks. Similarly, in malware detection datasets, common malware families such as Trojans are numerious, while Ransomware or Worms are less common families of malware. This class imbalance condition impacts the utility (how well TSD performs in place of Real Tabular Data (RTD) in downstream AI tasks) of the TSD task. In multi-class, malware files are commonly classified into

different families (e.g., Trojans, Ransomware, Worms). Similarly, network intrusions may belong to different network attack families (e.g., Denial-of-Service (DoS), Probe, User to Root (U2R), Remote to Local (R2L), and Man-in-the-Middle (MitM) attacks)).

In this thesis, we produce TSD by adopting a GAN-based methodology named Focal Loss Auxiliary Conditional Tabular GAN (Focal-AuxCTGAN). The class imbalance problem during TSD generation is addressed by integrating focal loss into the generation process. In addition, the utility of the produced TSD is improved by incorporating an auxiliary classifier in the Focal-AuxCTGAN methodology, evaluated under joint-training and pre-training settings. The proposed methodology can be adopted to produce utility-preserving TSD for downstream classification tasks in cybersecurity, particularly in malware classification and intrusion detection.

The data-driven TSD methodology used in the thesis is GAN-based. The motivation to focus on GAN-based methodologies is their distinct advantages over conventional TSD methods. Conventional TSD methods introduce noise into the real data to ensure privacy. In contrast, a GAN-based method produces synthetic data from the noise sampled from a Gaussian distribution, which mitigates the risk of re-identification of private information from synthetic data. In particular, the adversarial sample generated with a GAN accommodates the synthetic data to preserve the utility of the real data, making generated suitable for downstream tasks.

1.2 Goals

The research was conducted according to two goals. The first goal is to evaluate the potential of modern AI paradigms, such as adversarial learning, in creating realistic adversarial Windows PE malware and enhancing the robustness of Windows PE malware detection models to adversarial attacks. The second goal is exploring the utility of a GAN-based methodology for TSD in various downstream classification tasks comprising malware detection and network intrusion detection problems.

1.3 Contributions

The core contribution of this thesis is summarized as:

- We describe a comprehensive background in AI, Explainable Artificial Intelligence (XAI), adversarial learning in Windows PE malware detection, adversarial learning in Windows PE file analysis, adversarial attacks on Windows PE files and GAN-based TSD generation methods.

- We explored the offensive and transferability of the five, literature, attack methods for generating realistic Windows PE adversarial malware introduced in [1], that is: Extend, Full DOS, Shift, FGSM (padding+slack), and GAMMA. We also performed a distance-based analysis of adversarial Windows PE malware to explore possible relationships between the amount of changes introduced in the malware via the attack method and the evading ability of the produced adversarial malware.

  Finally, we performed an interpretative analysis of adversarial Windows PE malware generated using SHapley Additive exPlanations (SHAP) to explain how the attack methods considered in this study can change the Windows PE malware files to fool the decisions of AI models.

- We explored the performance of the adversarial training strategy introduced in [2] as a defensive strategy to train a new AI model by incorporating the realistic adversarial Windows PE malware files generated with the literature attack methods and we evaluated the accuracy and robustness of the new model on real malware and adversarial malware , respectively.

- We implemented and evaluated a TSD method, named Focal-AuxCTGAN, that combined the TSD method named CTGAN [3] and the use of an auxiliary Multi-layer Perceptron (MLP) as described in [4]. The MLP was trained in the Joint-training and Pre-training settings, respectively.

1.4 Outline

This thesis is organized as follows:

- In Chapter 2, we introduce the background on AI for classification problems with attention to machine learning, deep learning, XAI and adversarial learning literature.

- In Chapter 3, we introduce basics of and summarize literature on AI methods for Windows PE malware detection and classification. Then, we delve into literature adversarial methods to generate realistic adversarial Windows PE malware. Finally, we provide an overview of benchmark datasets commonly used in Windows PE malware analysis.

- In Chapter 4, we present a critical empirical study conducted for evaluating literature adversarial attacks and adversarial training in Windows PE malware detection. The study also includes an analysis of the performance of offensive and defensive results performed using both the Euclidean distance and SHAP.

- In Chapter 5, we introduce the background in TSD literature regarding GAN methods.Then, we describe the Focal-AuxCTGAN method and the results of the evaluation of the utility of the considered method and its baseline CTGAN in several classification problems. Finally, the chapter concludes with lessons learned and future research directions.
- In Chapter 6, we describe the conclusion of the thesis work, the limitations, and the future directions.

Chapter 2

Background in Artificial Intelligence

This chapter illustrates the learning paradigms, supervised machine learning methods, supervised and unsupervised deep learning methods, XAI methods, and adversarial learning in offensive and defensive scenarios.

2.1 Learning paradigms

The learning paradigms in AI define the learning methodologies through AI models can be learned. Each learning paradigm differs based on the type of information provided to the model during training. In particular, the learning paradigms are categorized into supervised learning, unsupervised learning, and semi-supervised learning.

The supervised learning is a learning paradigm to learn a function that maps an input into an output based on a collection of input-output examples called *training set*. It uses a collection of labeled examples to infer the mapping function [5]. The supervised learning paradigm is commonly used to formaulate several Machine Learning (ML) and Deep Learning (DL) algorithms for classification and regression problems. Specifically, the classification task involves categorizing examples into distinct classes, while the regression task involves predicting a continuous output.

The unsupervised learning is a learning paradigm to analyse an unlabeled dataset and identify patterns, trends, and structures without relying on predefined outputs [6]. It aims to recognize inherent groupings or relationships within the data, enabling the discovery of underlying structures in a data-driven manner.

The semi-supervised learning is a learning paradigm that uses both labeled and unlabeled examples. It is used in the design of several ML and DL algorithms [7]. It uses the labeled training set to guide the learning process and leverages the unlabeled dataset to capture patterns

and structures within the data. This approach is effective in scenarios where the labeled data amount is limited or costly to obtain, but a large amount of unlabeled data is available.

2.2 Machine learning methods for supervised learning

The notations used in this section are reported in the following.

- $\mathcal{D} = \{(\mathbf{x_i}, \mathbf{y_i})\}_{i=1}^{N}$: The training set, consisting of N training examples spanned over $\mathbf{X} \times Y$.
- $\mathbf{X} = (X_1, X_2, \ldots, X_m)$: An m-dimensional vector of independent features, where X_j may be either discrete or numerical for $j = 1, \ldots, m$.
- p_i is the proportion of examples of $\mathcal{D}$ labeled with class $Y = C_i$
- $\mathbf{W} = (W_1, W_2, \ldots, W_m)$: An m-dimensional vector of weights corresponding to the independent features in $\mathbf{X}$, where each W_j represents the weight associated with feature X_j for $j = 1, \ldots, m$.
- $Y \in \{C_1, \ldots, C_K\}$: The dependent feature with K distinct classes.

  This Section illustrates the theory of the following supervised algorithms for classification problems: Decision Tree, Random Forest, Support Vector Machine, Light Gradient Boosting Machine, K-Nearest Neighbour, Naive Bayes, Linear regression, and logistic regression.

- **Decision Tree (DT)** [8] is one of the most important non-parametric supervised learning algorithms. The DT algorithm is formulated for both classification and regression tasks. It is a tree-like top-down structure consisting of one root, and several decision, and leaf nodes. The root node is the starting node of a decision tree representing the entire training set. This node divides the training set into non-overlapping subsets based on a threshold, which are passed to child nodes. Following the root node, the outcome nodes are the decision nodes. Each node gets a subset of the training set from a parent node, further divided based on a logical test and passed to the child nodes. The recursive division continues until a stopping criterion is achieved, i.e., all the training examples within a node refers to a single class or the tree attains its maximum depth. Leaf nodes are the terminal nodes that contain the predicted outputs, class labels, or regression values [9,10]. The following algorithms are the most representative and well-known algorithms to train a DT. Each algorithm uses different metrics to identify the optimal condition for data splitting.

The Classification and Regression Tree (CART) [11] algorithm, produces binary splits, meaning each node possesses only two branches [12]. To produce binary splits, it behaves differently with numerical and discrete features. For numerical features, it sorts all unique values in ascending order and evaluates each midpoint between consecutive unique values as a potential condition for splitting. For discrete features, it divides the values into two groups and evaluates all possible groupings as a condition for splitting. CART employs the Gini Index as its splitting criterion for classification tasks. The Gini index quantifies the impurity of the data at a given node by assessing the probability of misclassification based on the majority class within the node. It determines the optimal condition for data splitting at each node in a decision tree. The Gini index value ranges between 0 and 1. The lower the Gini index value the better the data partitioning, while the higher the Gini index value, the greater the partitioning impurity. The goal is to select the condition expected to result in the lowest Gini Index, aiming to minimize the impurity after the split. The mathematical formulation of the Gini index is expressed in equation 2.1.

$$\text{Gini index} = 1 - \sum_{i=1}^{K} p_i^2 \tag{2.1}$$

CART uses variance reduction to identify the conditions for regression tasks. The goal is to reduce the outcome variance in each node [13].

The Iterative Dichotomiser 3 (ID3) [14] algorithm is originally designed to work with discrete features. It produces a split for each unique value of the discrete features. ID3 measures the Entropy Information Gain (InfoG) to identify the optimal condition for training set splitting. This algorithm only works for classification tasks. The entropy assesses the randomness or uncertainty of the class values through the training data at a node, accounting for the probability distribution of all classes. In the first step, ID3 computes the entropy of the training set at the root node. The mathematical formulation of the entropy is defined in equation 2.2.

$$H(\mathcal{D}) = -\sum_{i=1}^{K} p_i \log_2(p_i) \tag{2.2}$$

The entropy value ranges between 0 and 1. An uniform distribution of the data classes leads to higher entropy. In the second step, the training set is split into subsets considering all possible candidate split conditions on an independent a feature. To evaluate a split condition, the entropy of each subset is computed, and the weighted entropy is finally calculated by summing the entropies of all subsets (weighted on the amount of

training examples falling in each subset derived by the candidate split). In the third step, the InfoG of a feature is computed. The split condition achieving the maximum InfoG is selected for the tree induction. The mathematical formulation of the InfoG is defined in equation 2.3.

$$IG(\mathcal{D}, X_j) = H(\mathcal{D}) - \sum_{v \in \text{Values}(X_j)} \frac{|\mathcal{D}_v|}{|\mathcal{D}|} H(\mathcal{D}_v) \tag{2.3}$$

where Values(X_j) represents the values of feature X_j to split the training set and $\mathcal{D}_v$ is the subset of $\mathcal{D}$ for which feature X_j has the value v. Equation 2.3 shows that the InfoG quantifies the entropy reduction for a condition at each decision node. ID3 selects the feature with the highest InfoG as optimal condition for splitting the data. [15].

The InfoG may prefer conditions on features with a high number of distinct values, but these split conditions may fail in generalizing on the unseen data. C4.5 [8], an extension of the ID3 algorithm, is introduced to address this bias issue. It works with both discrete and numerical independent features. For discrete features, it produces splits for each unique value of features and for numerical features, produces binary splits similar to CART. C4.5 uses the Gain Ratio to mitigate bias. The Gain Ratio of a condition is computed by dividing the information gained by the SplitInfo. The formula of Gain Ratio is defined in equation 2.4.

$$\text{Gain Ratio}(\mathcal{D}, X_j) = \frac{IG(\mathcal{D}, X_j)}{\text{SplitInfo}(\mathcal{D}, X_j)} \tag{2.4}$$

The SplitInfo determines the degree of narrowness in splitting data for a condition. High values of SplitInfo decreases the value of Gain Ratio. The mathematical formula to calculate the SplitInfo of a condition is defined in equation 2.5.

$$\text{SplitInfo}(\mathcal{D}, X_j) = -\sum_{v \in \text{Values}(X_j)} \frac{|\mathcal{D}_v|}{|\mathcal{D}|} \log_2 \left(\frac{|\mathcal{D}_v|}{|\mathcal{D}|} \right) \tag{2.5}$$

To obtain a balance between bias and high information gain, C4.5 selects the condition with the highest Gain Ratio to split the training data [16, 17].

- **Random Forest (RF)** [18] is an ensemble classification algorithm. This algorithm employs parallel ensembling, which means multiple decision trees fit in parallel on a distinct subset of the training set. It is designed to handle both classification and regression tasks.

Generally, the RF model performs better than a single decision tree. The final prediction is determined by the majority voting in classification and average in regression tasks [19].

To learn decision tree forests, the RF algorithm uses the bagging technique [20]. The bagging technique creates training data subsets called bootstraps from the original training set by randomly picking training examples with replacements. This random sampling produces diversity in the ensemble and ensures that each decision tree is trained on slightly different data. In addition to the bagging, RF employs a random feature selection [21] technique, called randomization, to ensure that the different forest decision trees use different feature subsets. The algorithm selects a random subset of features at each node. The RF algorithm uses the Gini index or the Information Gain metrics to identify the optimal split condition at each node. Combining bagging and randomization, the RF algorithm mitigates the overfitting phenomenon, handles high-dimensional data.

- **Support Vector Machine (SVM)** [22] is used for both classification and regression tasks. The SVM algorithm is well-known for its ability to handle high-dimensional data and its effectiveness in linear and non-linear classification problems. The SVM algorithm aims to separate the training examples into two classes, identifying the best surface that maximizes the margin between them by solving an optimization problem. The margin maximization ensures the model's generalization ability on unseen data, minimizing the risk of overfitting. The main limitation of the SVM algorithm is the limited scalability. In linearly separable training data, the data has limited intersections. In this rare scenario, many candidate hyperplanes could perform the class separation. The hyperplane that separates the input data is defined in equation 2.6.

$$\mathbf{W}^T\mathbf{X} + b = 0 \tag{2.6}$$

where $\mathbf{W}$ is the weight vector, $\mathbf{X}$ is the independent feature vector, and b is biased, which adjusts the hyperplane position relative to the origin. In the linear separable case, the hyperplane margin boundaries are defined as inequation 2.7.

$$\mathbf{W}^T\mathbf{X} + b = \begin{cases} 1, & \text{for the positive class} \\ -1, & \text{for the negative class} \end{cases} \tag{2.7}$$

The optimal hyperplane linearly separating the training data is obtained by maximizing the margin distance as reported in equation 2.8.

$$\min \frac{1}{2}\|\mathbf{w}\|^2 \tag{2.8}$$

subject to:

$$y_i \left(h_i + \mathbf{W}^T \cdot \mathbf{X}_i + b\right) \geq 1, \quad \forall i \tag{2.9}$$

where i represents each example in the training set. The margin is the distance between the hyperplane and the nearest training data to the hyperplane. The goal of minimizing the value of equation 2.8 is to find the optimal hyperplane and two parallel hyperplanes (H1 and H2). By maximizing the distance between H1 and H2, some training examples lie on H1 and some on H2. These training examples are called support vectors. These support vectors directly contribute to determining the separation hyperplane. Some scenarios exist where the linear separation hyperplane could achieve satisfactory results even if the data has intersections. However, to find the optimal separation hyperplane, equation 2.9 should be modified by introducing non-negative slack variables. The equation 2.10 is the modified mathematical form obtained by modifying equation 2.9 accordingly.

$$y_i \left(\mathbf{W}^T \mathbf{X_i} + b\right) \geq 1 - \xi_i, \quad \forall i \tag{2.10}$$

where ξ_i represents a slack variable, which measures the margin violation of each training example. In non-linearly separable training data, even if the hyperplanes are determined optimally, the model may not have a well-generalized ability with this hyperplane. The Kernel function mitigates this problem by transforming the input into a high-dimensional feature space. The SVM algorithm finds the optimal hyperplane for class separation in this new feature space. The Kernel functions used in SVM include: Linear Kernel, Polynomial Kernel, Radial Basis Function, and Sigmoid Kernel. Each Kernel function transforms data differently depending on the application type and data characteristics [23, 24].

- **Gradient Boosting** is a machine learning algorithm that builds predictive models by combining multiple weak decision trees to produce a robust ensemble classifier [25]. The traditional gradient boosting technique analyzes all training examples to compute the InfoG metric of each candidate split condition. This approach makes the computational complexity proportional to the number of independent features and examples. Light Gradient Boosting Machine (LightGBM) [26] is an efficient version of the Gradient-Boosting algorithm to tackle computation time issues. The LightGBM algorithm incorporates: Gradient-based One-Sided Sampling (GOSS) and Exclusive Feature

Bundling (EFB), to improve both training efficiency and predictive accuracy. Training examples with different gradients contribute differently to compute the InfoG metric. The idea is that training examples with small gradients have small training errors, which means that the examples are already trained well. So, the GOSS method prioritizes training examples with higher gradients to concentrate more on under-trained examples. The training examples with higher gradients are more informative for computing the InfoG metric. Procedurally, the GOSS method first sorts the training examples by the absolute values of their gradients in descending order and selects the top $a \times 100\%$ examples. Subsequently, it selects the random examples $b \times 100\%$ of the remaining data. a and b are the user-defined parameters that control the proportion of high-gradient and low-gradient examples, respectively, in the training process. After sampling, to balance the data distribution in computing the InfoG metric, GOSS applies a constant multiplier term, defined as $\frac{1-a}{b}$, to the examples with small gradients. The weight adjustment is defined in equation 2.11.

$$\mathbf{W}_i = \begin{cases} 1 & \text{if } i \text{ is a high-gradient sample} \\ \frac{1-a}{b} & \text{if } i \text{ is a low-gradient sample} \end{cases} \tag{2.11}$$

The data distribution balancing ensures that the sampling does not decrease the effect of the examples with small gradients. This strategy boosts the learning process and reduces the computational complexity. Finally, the EFB technique optimizes feature selection by grouping sparse and mutually exclusive features. This grouping approach results in the size reduction of the feature matrix [27]. In addition, LightGBM converts continuous features into discrete bins, which reduces memory usage. Additionally, trees induced by LightGBM grow leaf-wise, which means LightGBM performs splits only at the optimal node, not at each node. The leaf-wise approach can cause the overfitting phenomenon. Overfitting can be mitigated by limiting the tree's maximum depth and limiting the data in the leaf node. [27, 28].

- **K-Nearest Neighbour (KNN)** predicts the unknown class of an example based on the known classes on the closest neighbour training examples. The KNN [29] algorithm is a non-parametric distance-based machine learning algorithm with no prior assumptions about the underlying data distributions. It is also known as a "lazy learning" algorithm. It can be used in classification and regression. As an instance-based learning method, the KNN algorithm does not build a generalized model during training. Instead, it stores all training examples in an n-dimensional space, where n represents the number of features in the training set. The KNN algorithm classifies new examples based on similarity with

the training data. The most commonly used distance metrics for numeric independent features are the Euclidean distance and the Manhattan distance. In contrast, for discrete independent features, the Hamming distance is the most commonly used. The Euclidean distance computes a straight line distance between examples. The mathematical form to compute the Euclidean distance between two examples $\mathbf{x_i}$ and $\mathbf{x_j}$ is defined in equation 2.12.

$$d_{\text{Euclidean}}(\mathbf{x_i}, \mathbf{x_j}) = \sqrt{\sum_{j=1}^{m}(x_{ik} - x_{jk})^2} \tag{2.12}$$

The Manhattan distance is the sum of the absolute differences between pairwise feature values. Mathematically, the Manhattan distance is defined in equation 2.13.

$$d_{\text{Manhattan}}(\mathbf{x_i}, \mathbf{x_j}) = \sum_{j=1}^{m} |x_{ik} - x_{jk}| \tag{2.13}$$

The Hamming distance is the sum of all mismatches between the corresponding independent features. Mathematically, the Hamming distance is defined in equation 2.14.

$$d_{\text{Hamming}}(\mathbf{x_i}, \mathbf{x_j}) = \sum_{k=1}^{m} \delta(x_{ik}, x_{jk}) \tag{2.14}$$

where δ shows the corresponding match or mismatch between the independent features.

In KNN, k_{NN} denotes the number of nearest neighbours to be considered for deciding the class of a query example. This is a user-defined parameter with k_{NN}. In a classification task, the class of the query example is assigned based on the majority class of neighbours. In regression, the predicted output is generally the average of the k_{NN}-nearest neighbour class values. For an unlabeled example $\mathbf{x_i}$, let $\mathbf{q_1}, \mathbf{q_2}, \ldots, \mathbf{q_{k_{NN}}}$ represents its k_{NN}-nearest neighbours in the training set. The predictive value $\hat{y}$ for $\mathbf{x_i}$ is shown in equation 2.15.

$$\hat{y}_{\mathbf{x_i}} = \mathcal{F}(y_{\mathbf{q_1}}, y_{\mathbf{q_2}}, \ldots, y_{\mathbf{q_{k_{NN}}}}) \tag{2.15}$$

where $\mathcal{F}$ represents the mode function for classification and the average function for regression. The selection of k_{NN} values is challenging in KNN. Another limitation is the high memory consumption, as the KNN algorithm stores training data and takes up the processing time for large datasets [30–32].

- **Naïve Bayes (NB)** is a simple, powerful probabilistic classification algorithm based on the Bayes-theorem. The NB algorithm computes the class likelihood of a given example assuming the conditional feature independence, meaning each independent feature contributes independently to the class likelihood [33]. The NB algorithm uses training data for computing both class and feature probabilities. For a new unlabeled example with the independent feature vector $\mathbf{x}$, it uses the posterior probability computed for each class individually on the training set and places the example in the highest probability class, as given in equation 2.16.

$$y' = \arg\max_{Y=C_i} \; P(Y) \prod_{j=1}^{m} P(x_j \mid Y = C_i) \tag{2.16}$$

  where $P(Y = C_i)$ is the prior probability of $Y = C_i$ and $P(x_j \mid Y = C_i)$ is the conditional probability of $X_j = x_j$ given that the example $\mathbf{x}$ belong to class Y_i.

  The independence assumption reduces the computation complexity, as the model can straightforwardly estimate the required probabilities, often requiring just a single pass through the training data. The NB algorithm has different variations, such as Bernoulli, Multinomial, and Gaussian, each mainly designed to deal with specific data distribution. For example the Gaussian Naive Bayes method assumes that numeric independent features follow the Gaussian distribution. The Multinomial Naive Bayes is primarily used for text classification. It classifies text by word frequency and models each feature likelihood using multinomial distribution parameters. The Bernoulli Naive Bayes is also used to classify text in a document. It assumes the Bernoulli distribution for independent features and measures the likelihood based on the presence or absence of words in a document. [34, 35].

- **Linear Regression (LR)** is used to predict the continuous output of a dependent feature based on the input values of one or more independent features. In the LR algorithm, the linear regression line represents a straight line that shows the relationship between the dependent and independent features. In the LR algorithm, a fundamental assumption is that the effect of the independent features on the dependent feature remains constant for all values of the independent feature. The main goal of the LR algorithm is to find the best line with the minimum distance between independent and dependent features. [36]. The LR algorithm is further classified as simple linear regression and multiple linear regression.

The Simple linear regression includes one continuous dependent feature and one independent feature (discrete or continuous). The Linear regression aims to fit a straight line to the data, as defined by equation 2.16.

$$Y = C + bX + \varepsilon \tag{2.17}$$

where C is a constant or intercept, b is the coefficient or slope, and ε is the error term. LR determines the C and b by minimizing the squared distances between the regression line and the training examples. The differences between the actual values of the dependent feature and the values predicted by the regression line are called residuals. Residual values help to understand how well the model fits the training data: smaller residuals mean the model explains more of the variation in the response, while larger residuals show less explanation. The linear relationship assumption between the independent and dependent features is the key limitation of the simple linear regression method. In practice, the relationship between variables is often unknown, assuming linearity is too restrictive.

Multiple linear regression is the extension of simple linear regression, which includes two or more independent variables and one dependent variable, as defined in Equation 2.17.

$$Y = C + b_1X_1 + b_2X_2 + b_3X_3 + \cdots + b_mX_m + \epsilon \tag{2.18}$$

The value of the dependent feature Y is determined by the intercept and the sum of each predictor variable multiplied by its coefficient plus error. The Multiple regression is a potent method to examine the combined impact of multiple independent features on the dependent feature, and interpreting the results is relatively easy. The Multicollinearity is the primary limitation of the multiple regression. It occurs when two predictors are highly correlated or when one predictor is a simple combination of others. As the number of independent features increases, the multicollinearity becomes more problematic [37–39].

- **Logistic regression (LGR)** [40] is a statistical model which describes the relationship between a dependent feature and one or more independent features. This model can be used for regression tasks, but it is explicitly designed for classification tasks. The LGR algorithm works like the LR. However, the outcome of the dependent feature in LGR is binary or categorical, while in LR, the outcome is continuous. Unlike LR, where training examples are arranged in a line, the LGR algorithm uses a regression formula to find a

decision line between categories. The model uses an optimization algorithm to find the best regression coefficients to classify examples accurately [41, 42].

The LGR model is built upon the concept of odds and log-odds [43, 44]. The odds represent the ratio of the probability of an event occurring to the probability of it not occurring. For instance, in a binary classification scenario like predicting rain versus no rain, the odds indicate how likely it is for rain to occur compared to not occurring. Equation 2.19 describes the mathematical formulation of the Odds value.

$$\text{Odds} = \frac{p}{1-p} \tag{2.19}$$

where p is the probability of the event happening, and $1-p$ is the probability of the event not happening. Log-odds or logits are the logarithm of odds shown in equation 2.20.

$$\text{Logit}(p) = \ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 X \tag{2.20}$$

β_0 is the log-odds when $X = 0$, and β_1 is the coefficient of the independent feature X. The LGR algorithm uses log-odds as a linear function of the independent features. This allows the predicted values to range between 0 and 1, making it suitable for classification. The actual probability from the logit function can be computed using the logistic function shown in Equation 2.21. The LGR algorithm ensures that the predicted probability values are always between 0 and 1.

$$p = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X)}} \tag{2.21}$$

LGR is further classified as simple logistic regression and multiple logistic regression. Simple logistic regression considers only one independent feature, so the relationship is between the single independent feature and the discrete outcome. Equation 2.19 is an example of a simple LGR pattern. The Multiple LGR handles two or more independent features to predict a discrete outcome. Equation 2.20 is extended to include multiple independent features, as shown in Equation 2.22.

$$\text{Logit}(p) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_m X_m \tag{2.22}$$

The coefficients $\beta_0, \beta_1, \ldots, \beta_m$ indicate the effect of an independent feature on the log-odds outcome. However, the odds ratio is a more understandable measure of this effect.

The odds ratio greater than 1 means an increase in the predictor increases the odds of the event occurring, while less than 1 means an increase in the predictor decreases the odds of the event occurring. Equation 2.23 is the mathematical formula for the Odds ratio.

$$\text{Odds Ratio} = e^{\beta} \tag{2.23}$$

The LGR algorithm has some limitations. It may overfit in high-dimensional datasets. Regularization methods such as Lasso and Ridge can mitigate overfitting. The Multicollinearity may arise, making it less reliable. One solution is to eliminate or combine correlated independent features or to employ regularization techniques. Moreover, the LGR algorithm assumes a linear association between the log odds and the independent features. If this assumption is invalid, then the model may not fit the data well. In some scenarios, dummy variables or splines can address issues due to non-linearity.

2.3 Deep learning methods

The notation used in this section is reported in the following.

- f: An activation function transforms weighted inputs to output at each neuron.

This Section includes the theory of the following supervised deep learning algorithms: MLP, CNN, MalConv, RNN, and the following unsupervised deep learning algorithms: GAN and Autoencoders.

2.3.1 Supervised deep learning methods

- **MLP** is a feed-forward neural network known as a base architectural structure in deep learning. The algorithm consists of an input layer, one or more hidden layers, and an output layer. The input layer receives the training set, the hidden layers learn intermediate patterns from the processed training set, and the output layer produces the final decision [36, 45, 46]. The architecture of an MLP is fully connected, meaning that each neuron in a layer at level l is connected to every neuron in the layer at level $l + 1$ with each connection associated with a real-valued weight. Each neuron computes a weighted sum $\mathbf{z}$ of an input $\mathbf{x}$ according to weights $\mathbf{w}$ as reported in equation 2.24.

$$\mathbf{z} = \sum_{j=0}^{m} w_j x_j \tag{2.24}$$

The algorithm to train a MLP employs the Backpropagation strategy [36]. The Backpropagation strategy computes the difference between the predicted and real output, passes the computed difference backwards through the layers, and modifies the connection weights accordingly. This is an iterative process that reduces the difference by optimizing the layer weights. The MLP model is trained by using an optimizer to modify weights effectively. The most widely used optimizers are Stochastic Gradient Descent (SGD), Limited Memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS), and Adaptive Moment Estimation (Adam) [47]. The SGD updates weights based on a single or batch of examples; L-BFGS approximates the Hessian for efficient optimization, and the Adam optimizer combines the momentum and adaptive learning rates for faster convergence. The MLP algorithm uses non-linear activation functions such as: Rectified Linear Unit (ReLU), Hyperbolic tangent (Tanh), Sigmoid, or Softmax [47], to learn complex patterns within a training set. The mathematical formulation of an activation function applied to a weighted sum of inputs is defined in equation 2.25.

$$y' = f(\mathbf{z}) \tag{2.25}$$

The ReLU and Tanh are often used in hidden layers: ReLU helps in mitigating the vanishing gradient problem, while Tanh limits training set values of range between -1 and 1. The Sigmoid and Softmax are used in the output layer. In particular, the Sigmoid is used for binary classification and produces an output ranging between 0 and 1. The Softmax is used for multiclass classification. It normalizes the output into a probability distribution across all classes, with each probability ranging between 0 and 1, and the sum of all probabilities equal to 1 [48].

The accuracy of an MLP model can be enhanced by tuning its hyper-parameters, including the number of neurons in each layer, the number of hidden layers, the number of iterations, and the learning rate. A Tree-structured Parzen Estimator (TPE) [49] algorithm is one of the hyper-parameter optimization algorithms that is commonly used to explore the hyper-parameter space efficiently. The TPE algorithm refines the search for optimal hyper-parameters using density-based modeling. Initially, the TPE algorithm selects the hyper-parameters randomly from a set of hyper-parameter configurations and evaluates their corresponding objective values. These configurations, along with their evaluations, form a configuration set. The configuration set is then split into two groups,

a better configuration group and a worse configuration group, based on a threshold. The better configuration group contains the configurations with favourable objective values (e.g., the lowest for loss functions or the highest for scores), and the worse configuration group contains the remaining configurations. The TPE algorithm uses kernel density estimators (KDE) to estimate the probability densities of hyper-parameter configurations for both the better configuration and worse configuration groups. The probability densities enable the TPE algorithm to focus its search on promising configurations within the hyper-parameter space that are likely to yield better outcomes. After sampling these promising configurations, their objective values are evaluated, and the results are added to the configuration set. The splitting the configuration set, estimating probability densities, sampling promising configurations, and updating the configuration set is iterative until the stopping criteria, such as a predefined number of trials or a time limit, are met.

- **Convolutional Neural Network (CNN)** [50] is an advanced deep neural network that consists of input, convolutional, pooling, and fully connected layers, all working together to capture and interpret complex features in the training set. The CNN handles data of various dimensions, including 1-D for sequential data, 2-D for spatial data and 3-D for volume data. In the following, we describe a CNN for 2-D data. The input layer receives the training set populated with 2-D examples. The convolutional layer extracts features from the training set by focusing on local regions. In each layer, neurons are connected only to a subset of neurons in the previous layer, called the receptive field. This receptive field enables the layer to capture localized features. The weights associated with each receptive field form a filter or kernel, which is applied over the training examples. In the 2-D training set, the filter slides horizontally and vertically over each training example and produces a feature map. This sliding process is called the convolutional operation. The convolutional operation significantly reduces the number of trainable parameters as the same filter is used across the entire training set. The convolutional layer can use multiple filters, each designed to capture a distinct feature in the training set. The diagram of an example convolutional layer and its filters is displayed in 2.1. For a convolutional layer l, the output feature map for a is defined in equation 2.26.

$$\mathbf{X_s}^{l} = f\left(\sum_{\mathbf{r}\in\mathbf{M_s}} \mathbf{X_r}^{l-1} * \mathbf{W_{sr}}^{l} + \mathbf{b_s}^{l}\right) \tag{2.26}$$

where s is the $\mathrm{s}-th$ feature map in layer l, $\mathbf{M_s}$ is the receptive field for the $\mathrm{s}-th$ feature map and $\mathbf{X_r}^{l-1}$ is the value of $\mathbf{X_r}$ in layer $(l-1)$, and $\mathbf{W_{sr}}^{l}$ are weights connecting $\mathbf{X_r}$ in layer $l-1$ to $\mathbf{X_s}$ in layer l.

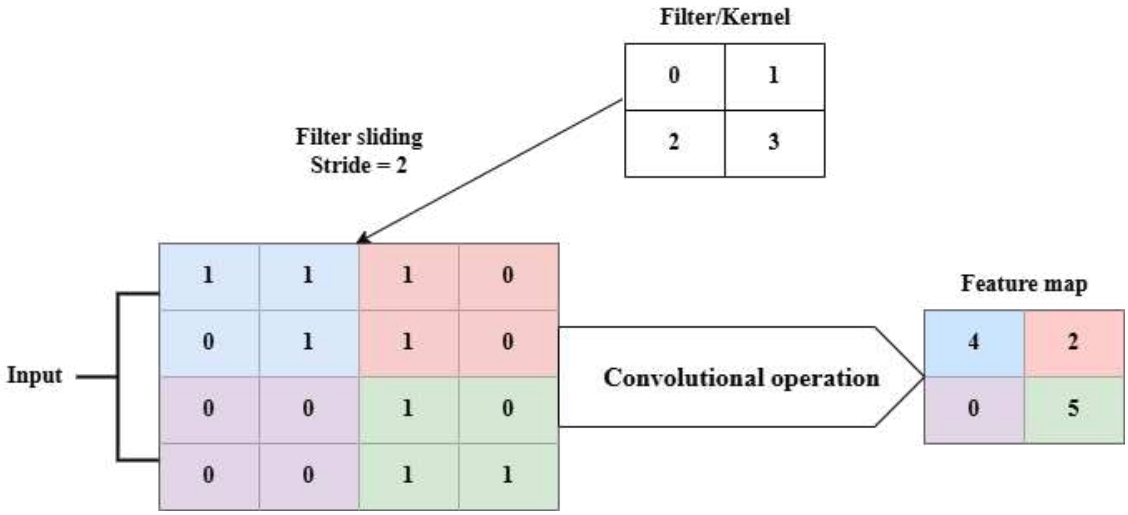


Figure 2.1: An overview of how a convolutional layer works

The pooling layer, also known as the down-sampling layer. It decreases the number of elements processed in the feature map. The pooling layer generally performs an operation of max pooling, mean pooling, or stochastic pooling. The max pooling selects the highest value from the feature elements within a neighbourhood. The mean pooling computes the average of the feature elements within a neighbourhood, and stochastic pooling randomly selects a value from the feature elements within a neighbourhood. These methods compress the feature map while preserving the crucial information, as defined in equation 2.27.

$$\mathbf{X_j}^{l} = \text{Pool}\left(\{\mathbf{X_r}^{l-1} \mid \mathbf{r} \in \mathbf{M_j}\}\right) \tag{2.27}$$

where Pool defines the type of the pooling function. A fully connected layer converts a 2-D feature vector into a 1-D output. This layer follows the same fully connected structure and computation as described for the MLP sytructure (see Equation 2.24) [51, 52].

- **MalConv** [53], is an end-to-end deep neural model that processes integer feature vectors with values ranging from 0 to 255. It is specifically designed for malware detection but can also be applied to other domains. The architecture of the MalConv is composed of an embedding layer, two parallel 1-D convolutional layers, the gated mechanism that combines the output of the two convolutional layers, a temporal max pooling, a fully connected layer and a softmax layer. The embedding layer maps each input sequence of 1-D integer values ranging among 0 and 255 into a sequence of 8-dimensional real embeddings. This is done by handling every integer value as a categorical value and

mapping it into an 8-dimensional real point of the embedded space. This embedding represents values that have semantically similar behaviour as closer points in this space. The convolutional layer includes 128 filters to iterate over disjoint windows of 500 input values each. The gated outputs of the convolutional layer integrate the global max pooling to select 128 features, achieving the highest activation value. The selected features are subsequently used to feed a fully connected layer in charge of the final soft-max classification.

- **Recurrent Neural Network (RNN)** is mainly used to process sequences of training sets [54, 55]. The algorithm memorizes the previous element's state in the sequence and uses this information to process the subsequent element. The RNN algorithm processes training set sequences and generates output sequences. Unlike the feed-forward neural networks, where the information flows unidirectionally, the RNN algorithm forms a feedback loop by putting back output as input, which means the layer's output becomes the input of the same layer. The RNN algorithm consists of input layers, hidden layers and output layers. The connections in the input and output layers are feed-forward, while the hidden layers are recurrent. The recurrent cells in the hidden layers combine the current input with the previous hidden state to compute the current hidden state. However, the RNN algorithm struggles to retain the dependent information between sequences over a long time. To address this issue, the Long Short-Term Memory (LSTM) network [56] introduces the memory cells in replacement of the traditional nodes in the hidden layers and the gates to manage the flow of information. The RNN algorithm maintains the information on the previous hidden states only, while the LSTM network maintains the information recorded in both the cell states and the previous hidden states. The cell states and the gates (input, output, and forget) are important concepts for the LSTM network. The cell states transmit information across the network. The input gate controls how much new information is stored in the memory cell. The forget gate controls the amount of the previous information that should be discarded. The output gate controls how much the internal memory of a cell contributes to the final output [55]. The mathematical formulation of the LSTM with input and output gates is defined in equation 2.28.

$$
\begin{aligned}
\mathbf{I_t} &= f_{sig}(\mathbf{W_{ih}h_{t-1}} + \mathbf{W_{ix}x_t} + \mathbf{b_i}) \\
\tilde{\mathbf{C}}_\mathbf{t} &= f_{tan}(\mathbf{W_{\tilde{c}h}h_{t-1}} + \mathbf{W_{\tilde{c}x}x_t} + \mathbf{b}_{\tilde{c}}) \\
\mathbf{C_t} &= \mathbf{C_{t-1}} + \mathbf{i_t} \cdot \tilde{\mathbf{C}}_t \\
\mathbf{O_t} &= f_{sig}(\mathbf{W_{oh}h_{t-1}} + \mathbf{W_{ox}x_t} + \mathbf{b_o}) \\
\mathbf{H_t} &= \mathbf{O_t} \cdot (\mathbf{C_t})
\end{aligned}
\tag{2.28}
$$

where $\mathbf{I_t}$ and $\mathbf{O_t}$ denote the input gate and output gate respectively. f_{sig} and f_{tan} denote the Tanh and Sigmoid activation function , respectively. $(\mathbf{t-1})$ denotes the previous time and $\mathbf{t}$ is the current time. $\tilde{C}_t, C_t$ and H_t are the candidate cell state, the cell state, and the final hidden state, respectively.

2.3.2 Deep learning methods for unsupervised learning

- **Generative Adversarial Networks (GAN)** is a network proposed in [57], for generative modeling. The GAN algorithm model the underlying distribution of a dataset and generates new examples close to the real examples based on the learned distribution. The GAN algorithm adopts a game-theoretic approach, where two neural networks, the generator $(\mathbf{G})$ and the discriminator $(\mathbf{D}_{\text{gen}})$, compete with each other. The generator $(\mathbf{G})$ receives a random noise vector $\mathbf{u}$, usually derived from a normal or uniform distribution, and generates an example that resembles the real data distribution. The mathematical formulation of the generator is defined in equation 2.29.

$$\mathbf{G}(\mathbf{u}) : \mathbb{R}^d \rightarrow \mathbb{R}^m \tag{2.29}$$

where d is the noise space dimensionality, m is the output space dimension, $\mathbf{u} \in \mathbb{R}^d$ is the noise vector and $\mathbf{G}(\mathbf{u})$ is the generated example. The discriminator behaves as a binary classifier, differentiate between the real example from the input data set and fake example generated by the generator. The discriminator produces a probability that represents the likelihood that an example is a real example. The mathematical formulation of the discriminator is defined in equation 2.30.

$$\mathbf{D}_{\text{gen}}(\mathbf{x}) : \mathbb{R}^m \rightarrow [0, 1] \tag{2.30}$$

where $\mathbf{D}_{\text{gen}}(\mathbf{x})$ is the probability that $\mathbf{x}$ is a real example. If the probability value is close to 1 then the example is a real example kept the training set. If the probability is close to 0, then the example is a fake example generated by the generator. The goal of the discriminator is to maximize the probability of correctly classifying both real and fake examples, while the goal of the generator is to minimize the discriminator ability to differentiate between real and generated examples. The objective function of both the generator and the discriminator is defined in equation 2.31.

$$\min_{\mathbf{D}_{\text{gen}}} \max_{\mathbf{G}} \mathbb{E}_{\mathbf{x}} \left[\log \mathbf{D}_{\text{gen}}(\mathbf{x})\right] + \mathbb{E}_{\mathbf{u}} \left[\log(1 - \mathbf{D}_{\text{gen}}(\mathbf{G}(\mathbf{u})))\right] \tag{2.31}$$

where $\mathbb{E}_{\mathbf{x}}$ and $\mathbb{E}_{\mathbf{u}}$ are the expected probabilities. Two challenges to address to train a GAN are: 1) the problem of the mode collapse, that is due to the fact that the generator produces examples with limited variations, and 2) the problem of the gradient saturation, which means that the discriminator may become over dominant providing limited feedback. Different architecture designs have been employed for both the generator and the discriminator, e.g. the Fully connected GAN and the Conditional GAN [3]. In a fully connected GAN, a generator produces synthetic data using random noise vector $\mathbf{u}$, while in a conditional GAN includes both the generator and the discriminator take an additional, conditional variable along with the random noise vector $\mathbf{u}$. The conditional variable is a class label, feature or any other relevant information. The generator generates synthetic data based on the specified conditional variable. The GAN algorithm can be learned with both supervised and unsupervised DL algorithms, based on whether labeled or unlabeled datasets are used [58, 59], although the GAN was originally formulated for unsupervised learning.

- **Autoencoder (AE)** [60] defines an architecture to train a representation of a dataset that allows us to reproduce a dataset accurately. The AE algorithm is composed of three components: the encoder, the latent space (code), and the decoder. The encoder compresses the dataset into a lower-dimensional latent space representation, transforming it into a code that captures the most important information of the input dataset. The decoder reconstructs the dataset from the code representation, aiming to produce an output as close to the original input as possible. The encoding and decoding process allows us to use the AE for dimensionality reduction, feature extraction and generative modelling. The AE algorithm has different variants, such as Sparse Autoencoder, Denoising Autoencoder, Contractive Autoencoder, and Variational Autoencoder, each designed to address specific dataset representation and reconstruction challenges [61,62]. The sparse autoencoder adds the sparsity constraint in the code, which means only a subset of neurons are active for the input dataset. The sparsity enables the neurons to represent distinct and meaningful representations in the code. The decoder reconstructs the data from this sparse code representation. The denoising autoencoder trains the decoder to reconstruct the dataset from the noisy code representation. This makes the model reconstruction robust to noise data. The Contractive Autoencoder creates a code representation that is robust to slight variations in the input dataset. This robustness is achieved by introducing a penalty on the Jacobian of the encoder's output concerning the input. This encourages

the model to learn stable representations. The decoder reconstructs the dataset from this robust code representation. The Variational autoencoder learns the probabilistic distribution in the code and generates synthetic data by sampling from the learned distribution. The variational autoencoder is commonly used to generate realistic faces, text, or audio and expand the dataset by generating diverse examples.

2.4 Explainable Artificial Intelligence (XAI)

The notations used in this section are reported in the following.

- $\mathbf{x}^*$: A baseline example.
- M_θ: The model to be explained as it is used in Integrated Gradients.
- ϕ: The model decision used in SHapley Additive exPlanations.
- ∂: gradient of a model decision.
- $\int$: An integral.
- $\mathbb{R}^m_{\text{input}}$: $m-$ dimensional real-valued input space.

This section includes the background on the XAI methods based on the scope and stage of the considered methods. The scope-based classification of XAI methods distinguishes between local explanations and global explanations, while the stage-based classification of XAI methods distinguishes between intrinsic and post-hoc XAI methods. The post-hoc XAI methods such as SHAP [63], moDel Agnostic Language for Exploration and eXplanation (DALEX) [64], and Integrated Gradient (IG) [65] are described in detail in the following.

In general, XAI methods make the behaviour and decision of the AI model more interpretable and understandable to humans by producing clear and meaningful explanations [66, 67].

2.4.1 Scope-based classification of XAI methods

Based on the scope, the explanations can be categorised into two types: local and global explanations.

- **Local XAI methods** The local explanations explain the model's decisions for an individual example [68, 69]. The local explanations analyse the contribution of each feature and provide the reason for the model decision for a given example. These explanations are crucial when an individual decision has consequences, such as in a medical diagnosis. The methods like IG, Local Interpretable Model-agnostic Explanations (LIME), and SHAP produce local explanations.

- **Global XAI methods** The global explanations explain the model's decisions considering the entire dataset [68, 69]. These explanations are useful to understand the overall behaviour of the model. The global explanations identify patterns, feature importance, and relationships between the input examples and the model decisions. The global explanation methods determine which features significantly influence the model's decisions and how the change in the value of these features impacts the overall model's decisions. Partial Dependency Plots (PDP) [70], and Accumulated Local Effects (ALE) [71] are examples of XAI that produce global explanations.

2.4.2 Stage-based classification of XAI methods

Based on the stage, the explainable methods are categorised into intrinsic and post-hoc XAI methods.

2.4.2.1 Intrinsic XAI methods

The intrinsic XAI methods, also known as self-explainable modelling, focus on designing models that are inherently explainable by embedding explainability into the model architecture during training. For example, constructing a decision tree or adding explainable components, such as attention layers in deep learning models, allow us to learn intrinsic explainable models [72].

2.4.2.2 Post-hoc XAI methods

The post-hoc XAI methods produce explanations for the decisions made by a trained model. These methods can be either model-agnostic, or model-specific XAI methods.

- **Model-agnostic** XAI methods explain the behaviour of any decision model, regardless of its internal structure [73, 74]. These methods focus on understanding the relationship between the input and output of a model. They handle the model as a black box, meaning

they do not require access to its internal working structure. The separation of the model-agnostic XAI method from the underlying model makes the underlying model more flexible for its task. The model-agnostic XAI methods can produce different explanations with different degrees of explainability. These methods can also produce explanations using features different from the ones used by the underlying model. The transition to a new underlying model is straightforward with model-agnostic explainability methods, as the explanation format remains consistent.
Additionally, these methods allow comparisons between models using consistent techniques and explanations. However, model-agnostic XAI methods have some limitations. These methods produce approximate explanations of model decisions, which might not be sufficient in domains where exact and more accurate explanations are required. In addition, it is challenging to customize the explanations produced by these methods according to the user's needs or feedback. Examples of model-agnostic XAI methods are: LIME [75], Counterfactuals [76], SHAP [63], and DALEX [64]. The SHAP, and DALEX are explained in detail in subsections 2.4.3 and 2.4.4, respectively.

- **Model-specific** XAI methods explain the decisions of the specific models by leveraging their internal structure and architecture [77]. So they depend on weights, activations, and gradients to produce explanations. These explanations provide a more in-depth understanding of decisions that are made and assist in identifying specific features, neurons, or layers that contribute significantly to the output. However, as these methods depend on the internal structure of the models, they lack generality and cannot be applied to different model architectures. This limits their usability in scenarios where multiple models need to be explained consistently. Some examples of model-specific XAI methods are: Guided Backprop [78], Layer-wise Relevance Propagation (LRP) [79], Gradient-Weighted Class Activation Mapping (GRAD-CAM) [80], and IG [65]. The IG is described in Section 2.4.5.

2.4.3 SHAP

SHAP [63] is a post-hoc, XAI method that produces an explanation of a model decision for a predicted class of an example. The SHAP method is based on the Shapley value from the theoretic game, measures the effect of each dimension of the input feature space on the decision produced from the model to predict the class of an example. This effect is measured as the average marginal contribution of the feature value for all alternative decisions. Let $\phi \colon \mathbb{R}^m_{\text{input}} \mapsto Y$ be the decision model, and $\mathbb{R}^m_{\text{input}}$ denotes the input space,

and Y be the set of classes. Let $\mathbf{x} \in \mathbb{R}^m_{\text{input}}$ denote the m-dimensional representation of an example to be classified with ϕ. The SHAP method measures the effect of each input dimension value $x \in \mathbf{x}$ on the decision $\phi(\mathbf{x})$ as the average marginal contribution of a feature value on the various alternative decisions. Let $\phi(\mathbf{x})[y]$ denote the confidence score according to ϕ sees $\mathbf{x}$ assigned to class $y \in Y$. ϕ assigns $\mathbf{x}$ in the class for which the highest confidence score is computed. For each input feature $X \in \mathbb{R}^m_{\text{input}}$ to explain, for each input feature sub-space $\mathbf{X} subseteq \mathbb{R}^m_{\text{input}}/\{X\}$, let $\phi_{\mathbf{X}} \colon \mathbf{X} \mapsto Y$ and $\phi_{\mathbf{X}\cup\{X\}} \colon \mathbf{X} \cup \{X\} \mapsto Y$ surrogate models with input feature spaces $\mathbf{X}$ and $\mathbf{X} \cup \{X\}$, respectively. The SHAP method computes the difference between the confidence scores determined by $\phi_{\mathbf{X}}$ and $\phi_{\mathbf{X}\cup\{X\}}$, respectively defined in equation 2.32.

$$\varphi_{\mathbf{X},X,y}(\mathbf{x}) = \phi_{\mathbf{X}\cup\{X\}}(\pi_{\mathbf{X}\cup\{X\}}(\mathbf{x}))[y] - \phi_{\mathbf{X}}(\pi_{\mathbf{X}}(\mathbf{x}))[y], \tag{2.32}$$

where $\pi_{\mathbf{X}} \colon \mathbb{R}^m_{\text{input}} \mapsto \mathbf{X}$ and $\pi_{\mathbf{X}\cup\{X\}} \colon \mathbb{R}^m_{\text{input}} \mapsto \mathbf{X} \cup \{X\}$ represent the functions to select the input values enclosed in $\mathbf{X}$ and $\mathbf{X} \cup \{X\}$, respectively. The Shapleye, finally, measured as the weighted average of the various alternative differences, that is defined in equation 2.33.

$$\Psi_{X,y}(\mathbf{x}) = \sum_{\mathbf{X}\subseteq\mathbb{R}^m_{\text{input}}\{X\}} \frac{|\mathbf{X}|!\ (m - |\mathbf{X}| - 1)!}{m!}(\varphi_{\mathbf{X},X,y}(\mathbf{x})), \tag{2.33}$$

The higher the value of $\phi_{X,y}(\mathbf{x})$, the most important the effect of X on the decision of the model of predicting $\mathbf{x}$ in the class y.

In [81], the malware files are executed in virtual machines, and the process-level information of the system is captured at 10-second intervals. Each file snapshot contains up to 120 active processes, and each process is described by the features, including CPU usage, memory usage, disk I/O activity, network behaviour, context switches, thread counts and the number of open file descriptors. The process-level classification patterns are learned by the CNN, SVM, RF, and Feed-Forward Neural Network (FFNN) models. The evaluation results show that the CNN algorithm outperforms in accuracy and F1-score. SHAP [63] is applied to explain the classification decisions of SVM, RF, FFNN, and CNN. The model-specific SHAP variants are applied (i.e., KernelSHAP for SVM, TreeSHAP for RF, and DeepSHAP for FFNN and CNN). The global and local explanation is computed using SHAP values. Global explanation is shown using summary bar plots of the top 10 most influential features for malware and goodware classification. Local explanation is shown using waterfall and force plots, illustrating the contribution

of each feature to individual classification outcomes. The explanation results are further validated through a feature removal strategy in which the top 20, 40, and 60 SHAP-ranked features are removed from the CNN input. The observed drop in performance confirms the importance of SHAP in identifying key features. This analysis supports the use of SHAP for transparency and feature importance validation in process-based malware detection.

In [82], malware detection is done based on an explainable AI-based method. The memory dump features, including the number of services, drivers, processes, threads, Dynamic-Link Libraries (DLL), and handles, are extracted using the VolMemLyzer tool. The feature set is reduced using the Recursive Feature Elimination (RFE) method, and the top five features (i.e., svcscan.nservices, dlllist.avg_dlls_per_proc, handles.nmutant, svcscan.kernel_drivers, and svcscan.shared_process_services) are selected based on RF-based ranking. Classification patterns are learned using DT, RF, NB, LR, and Gradient Boosting. The evaluation results show that Gradient Boosting achieves higher accuracy. The contribution score of each selected feature in the Gradient Boosting decision to classify a PE file as malware or goodware is computed using the TreeSHAP method and visualized through a summary bar plot. The plot shows that the svcscan.nservices feature contributes the most to the classification outcome, followed by dlllist.avg_dlls_per_proc and handles.nmutant. The SHAP explanations validate the model decision and confirm that the selected features influence the model classification decision.

2.4.4 DALEX

DALEX [64] is a model-agnostic post-hoc explainer method that produce both the global and the local explanations. In the global explanation, the DALEX method evaluates the feature importance by measuring the classification error prior to and after the permutation of a feature value. In contrast, the local explanation focuses on the individual case and identifies the unique patterns in the model decision. The DALEX method integrates various techniques for an explanation of model decisions, including PDP [70], ALE [71], and Shapley values [83]. The PDP is a global explanation technique that visualises the marginal affect of a feature on the predicted outcome by averaging predictions over the entire dataset. The ALE is a technique for global explanation that measures the effect of a feature on the model output by accounting for interactions measures with other features. The Shapley values are measured for both global and local explanations. The DALEX method is implemented in both R [84] and Python [64] programming languages, and each version offers unique functionalities. The R version of the DALEX library allow

us to perform the comparative analysis of explanations and the evaluation of multiple model decisions on the same dataset simultaneously. The Python version of the DALEX library integrates an interactive dashboard. The dashboard consists of several tabs, each focusing on how the model's decisions can be interpreted [85]. The performance metrics tab displays evaluation metrics such as accuracy, precision, recall, and F1 score. The global variable importance tab visualises the contribution of each feature to the model's decisions across the entire dataset. The individual variable importance tab explores the effect of each feature on decisions for specific examples. The accumulated local effects plot tab shows the feature contributions across observed value ranges. The residual plot tab examines the difference between the decision and actual values. The instance-level explanations tab allows users to compare model decisions with real values for specific examples.

In [86], a baseline Deep neural network (DNN) learned seperately the full feature set of the CICMalDroid20 [87] and CICMalMem22 [88] datasests. The DNN architecture consists of three fully connected layers, a dropout layer, and a batch normalization layer. The most influential features for malware classification from both datasets are identified through the permutation-based DALEX framework and the Mutual Information (MI) [89] analysis. The new DNN learned these top-N classification features ranked by each method. The DNN learns top-20 DALEX ranked features from CICMalDroid20 and achieves similar accuracy to the baseline model that learned the full set of features. For CICMalMem22, the DNN learned top-10 DALEX ranked features and achieve higher accuracy than baseline. In comparison, the MI-based approach rquires more features to achieve similar accuracy to the baseline. In addition, the adversarial examples are produced using Fast Gradient Sign Method (FGSM) attack method by modifying the DALEX and MI ranked features. The evaluation results show that adversarial examples produced by modifying DALEX ranked features achieve higher misclassification rate than those produced with MI guided features.

In [90], the cyber-threat detection is improved using an explainable guided ensemble learning method. The baseline DNN model is trained on the original training set. The adversarial examples are produced using the FGSM attack method. A set of candidate neural models is trained on subset of the original training set, each augmented with a small number of adversarial examples. The global feature relevance for each model is computed using the permutation-based DALEX method. Each model relevance vector is clustered using the k-medoids algorithm, and one medoid is selected from each cluster. The selected models are integrated into a multi-headed neural network and jointly

fine-tuned. The multi-headed ensemble model is evaluated on four multiclass cybersecurity datasets: NSL-KDD [91], UNSW-NB15 [92], CICIDS17 [93], and CICMalDroid20 [87]. The evaluation results show that the ensemble model outperforms the baseline on most datasets. In addition, examples that are misclassified by the baseline but correctly classified by the ensemble model are interpreted using local SHAP explanations.

2.4.5 Integrated Gradient

IG [65] is a model-specific (applicable only to differentiable models) post-hoc explainer method that produces explanations of the model's decisions by attributing them to its input features. The IG method is primarily designed to produce local explanations, but it can be extended to produce global explanations by aggregating measurements across multiple examples. The IG method measures the contribution of each feature to the model's decision by computing the gradients of the model's output (logit values) concerning the input features and integrating these gradients along a linear path from a baseline example to the real example. The baseline example is a neutral reference, which could be a black image for image models and a zero embedding vector for text models. The IG method ensures sensitivity and implementation invariance [94, 95]. The sensitivity ensures that if when we change the feature value of a real example we observe that the decision of the model changes accordingly, then the attribution assigned to that feature is non-zero. Conversely, if a change in the feature value does not change the decision of the model then the attribution assigned to that feature is zero. The sensitivity obtain the completeness, meaning that the sum of all feature attributions is equal to the difference between the model's output for the real example and its output for the baseline example. The sensitivity ensures that the attribution method faithfully reflects the model's behavior.

The implementation invariance ensures that two functionally equal models produce similar outputs and explanations for the same input, regardless of their differences in implementation. The mathematical formulation of the IG method to measure the contribution of a feature X_j to the model decision as the values of the feature X_j move from the baseline example $\mathbf{x}^*$ to the real example $\mathbf{x}$, is defined in equation 2.34.

$$\text{Integrated Gradient}_j(\mathbf{x}) = (\mathbf{x_j} - \mathbf{x_j^*}) \int_{\alpha=0}^{1} \frac{\partial M_\theta(\mathbf{x}^* + \alpha(\mathbf{x} - \mathbf{x}^*))}{\partial \mathbf{x_j}} d\alpha \tag{2.34}$$

where α is a scaling factor ranging from 0 to 1. $d\alpha$ is an infinitesimal change in α. The formulation 2.34 evaluates the contribution of the baseline example with $\alpha = 0$ and evaluates the contribution for the real example with $\alpha = 1$.

Notable, the calculation of the IG involves multiple evaluations of the model and its gradients along the path from the baseline example to the real example, making the computation expensive, especially for large models. Additionally, the IG method depends on a baseline, which can affect the interpretability of the explanation. Therefore, the choice of the baseline is crucial.

In [96], the static features are extracted from Android APKs using the Derbin framework. The extracted feature vectors are used to train classifiers, including LR and SVM. The vulnerability of LR and SVM is assesed by attacking adversarial examples produced using Projected Gradient Descent (PGD) attack method. The feature contribution to the malware class decision is computed using the IG method. The analysis is perfoemd using two explanation evenness metrics: (a) the cumulative ratio of top-k feature contributions, and (b) the ratio between the ℓ_1 and ℓ_∞ norms of the relevance vectors. The evaluation results show that classifiers with more evenly distributed feature contribjutions are more robust against adversarial evasion attacks.

2.5 Adversarial learning

This section includes a definition of adversarial learning, adversarial examples, transferability, and the background on the adversarial attack scenarios, offensive adversarial learning, and defensive adversarial learning. In short,

- Adversarial machine learning (AML) [97] is a sub-field of machine learning that studies attacks intended to deceive machine learning models (offensive adversarial learning) and the defense against such attacks (defensive adversarial learning).
- Adversarial examples [2, 98, 99] are carefully and maliciously well-crafted inputs to deceive a target decision model.
- Transferability [100] explores how adversarial examples produced to fool a target model can compromise the performance of another model, even if the new model is trained on a different training set or has a different architecture.

2.5.1 Adversarial attack scenerios

The adversarial scenarios can be classified according to: 1) adversary capability (evasion attacks and poisoning attacks); 2) adversary goals and objectives (availability, integrity, confidentiality); 3) adversary knowledge of the deceiving model (white-box attacks and black-box attacks).

2.5.1.1 Adversary capability-based attacks

The adversarial attacks against a model may occur either in the testing stage or in the training stage. This corresponds to two types of attacks:

- **Evasion attacks** [101] are induced by the adversary at the testing stage. The adversary produces adversarial examples by crafting the test examples, which are close to the original input examples, and alters the decision of the decision model according to its own goals.

- **Poisoning attacks** [102] are induced by the adversary in the training stage. The adversary produced adversarial examples by crafting the training examples and implanting these adversarial examples in the original training set. In the poisoning attack, the adversary forces the model to be trained to perform abnormally on specific implanting examples.

2.5.1.2 Adversary goals and objectives-based attacks

The adversarial attacks are categorised as security violation attacks and specificity attacks [103] based on the objectives and goals of the adversary. The Security violation attacks are classified as availability, integrity, and confidentiality violations. The availability violation means compromising the functionality of the decision model to stop performing legitimate tasks. The integrity violation means compromising the performance of the decision model for a specific task without disturbing the performance on remaining tasks. The confidentiality violation means revealing the private information of the decision model.

The specificity attacks are classified as target and non-target attacks. In the target attacks, the adversary forces the model to make decisions for specific classes; however, in non-target attacks, the model makes decisions for any class.

2.5.1.3 Adversary knowledge-based attacks

The adversary knowledge means the level of the adversary knowledge about the decision model. The more the knowledge of the attacker on the decision model, the more impactful the adversarial attack. The adversarial attacks produced are categorized as white-box attacks and black-box attacks based on the adversaries' knowledge.

- **White-Box** attacks assume the adversary has full knowledge about the decision model, including model type, model architecture, model parameters, and the model training data. FGSM [2], Basic Iterative Method (BIM) [104], PGD [105], DeepFool [106], and Jacobian Saliency map (JSM) [107] are white-box attack methods to produce adversarial examples described in detail in subsection 2.5.2.

- **Black-Box** attacks assume that the adversary does not know the decision model architecture or the training data used for its training. However, sometimes adversaries may have some knowledge of the decision model architecture, but not the knowledge on the weights. In black-box attacks, the adversary queries the trained decision model on the classification of various data samples. A key challenge in black-box attacks is minimizing the number of queries made to the decision model. Zeroth Order Optimization (ZOO) [108], HopSkipJumpAttack [109], and SIGN-OPT [110] are black-box attack methods to produce adversarial examples. The ZOO attack method is described in detail in subsection 2.5.2.

2.5.2 Offensive adversarial learning

This subsection includes the theory of the gradient-based attack methods FGSM, BIM, PGD, DeepFool, JSM, and ZOO for producing adversarial examples to increase the misclassification rate of a decision model.

- **FGSM** [2] uses neural network gradients to produce adversarial examples. This method computes the gradient of the model's loss function with respect to the input data. Gradients are the derivative functions that give information about which direction to move to maximize the loss. A small, flat amount of perturbation ϵ is then added to each feature (or pixel, in the case of images) that has the sign of the gradient. The FGSM aims is to minimize the maximum perturbation magnitude added to each pixel to cause misclassification. The mathematical formulation of the FGSM attack is defined in equation 2.35.

$$\mathbf{x}^{adv} = \epsilon \text{sign}(\nabla_{\mathbf{x}} J(\theta, (\mathbf{x}, y)) \tag{2.35}$$

where $\mathbf{x}^{adv}$ is the adversarial example, $\mathbf{x}$ is the original example, y is the target associated with $\mathbf{x}$, $\nabla_{\mathbf{x}}$ is the gradient computed with respect to $\mathbf{x}$, θ is the decision model parameter, and $J(\theta, \mathbf{x}, y)$ is the cost function used to train the model. FGSM is a fast adversarial examples generation method, although this methodology for producing adversarial examples is less efficient than other cutting edges.

In [111], adversarial examples are produced using the FGSM attacking method. The FGSM attack is implemented with various values of $\epsilon \in [0.25, 0.50, 0.75, 1.0]$ and [2048, 4096, 8192, 16384] bytes are modified. The malware functionality of the Windows PE malware files is reserved by injection noise inside the sections of PE files. The evasion capaboility of the produced adversairl examples is evaluated against the MalConv [53] detector, which process raw bytes of PE files for classification. The evaluation results show that the adversarial examples produced by injecting adversarial noise inside sections achives higher misclassification rate compared to the adversarial examples produced by appending adversarial noise at the end of the PE file.

- **BIM** [104] is an extended version of the FGSM. It produces adversarial examples by initializing the adversarial example with the original example and iteratively perturbing it using a small step size α. In each iteration, the adversarial example is updated by adding the perturbation in the direction of the gradient. The step size α ensures that the perturbation is applied gradually. The ϵ controls the magnitude of the perturbation to keep the adversarial example close to the original example. After each iteration, the adversarial example is clipped to ensure it remains within bounds. The mathematical formulation of the BIM attack is defined in equation 2.36.

$$\mathbf{x}_0^{adv} = \mathbf{x}, \;\; \mathbf{x}_{i+1}^{adv} = \text{Clip}_{\mathbf{x},\epsilon}\left\{\mathbf{x}_i^{adv} + \alpha \cdot \text{sign}(\nabla_{\mathbf{x}} J(\mathbf{x}_i^{adv}, y_i))\right\} \tag{2.36}$$

where $\text{Clip}_{\mathbf{x},\epsilon}$ is a function performs clipping.

In [112], the resilience of a DNN model against adversarial attacks is improved using an iterative adversarial retraining method. The model is initially trained on a dataset containing clean and Gaussian noise–augmented examples. In subsequent iterations, adversarial examples are produced using the BIM and DeepFool attack methods based on the previously trained model. The DNN is then retrained using clean, noisy, and adversarial examples, with soft labels adjusted according to perturbation magnitude. In addition,

an ensemble detection model is trained using clean and Gaussian noise–augmented examples. During the testing stage, this ensemble model compares predictions with and without added noise; if the prediction differs, the example is classified as adversarial, otherwise, the example is passed to the retrained DNN for classification. The results show that the proposed approach improves the adversarial robustness of the DNN while maintaining accuracy on clean examples.

- **PGD** [105] is an extension of the BIM method. It starts by initializing the adversarial example with a small random perturbation within the $\ell_\infty-$ball. This random initialization helps PGD escape local minima in the loss landscape. The PGD method explores different regions within the $\ell_\infty-$ball through multiple random restarts. The iterative updates in PGD are similar to those in BIM. However, producing adversarial examples with PGD is computationally more expensive than using the FGSM or BIM methods.

  In [113], a FFNN learned the behavioural feature of CIC-AndMal2020 [114] dataset. The FFNN architecture consists of six hidden layers followed by two fully connected layers and a dropout layer. The class imbalance issue is addressed using the Synthetic Minority Oversampling Technique (SMOTE) technique. The model classification performance is evaluated using accuracy and F1-score metrics. The model decision is interpreted by computing SHAP values, and the top-20 most influential features for the malware class are identified. The SHAP values are computed using DeepExplainer, and the top-20 influential features of the model decision for malware class are identified. The adversarial examples are produced by perturbing the SHAP-ranked features using the PGD attack method. The evaluation results show that modifying the most influential SHAP-guided features achieves higher misclassification rate.

- **DeepFool** [106] is an untargeted adversarial attack method that produces adversarial examples aiming to minimize the Euclidean distance between the original and perturbed examples. The attack is generated by first calculating analytically linear decision boundaries that divide examples into distinct classes, then computing and adding a perturbation that moves examples closer to or across the nearest decision boundary. If the target model is a DNN, then the decision boundaries are often nonlinear, so DeepFool performs the attack iteratively by adding a perturbation to the example at each iteration. The algorithm stops as an adversarial example has been generated or a predefined maximum number of iterations or maximum perturbation is reached. Specifically, the DeepFool attack method considers a decision model and an example $\mathbf{x}$. At each iteration i, a perturbation $\boldsymbol{\epsilon}_i$ is added to the current example $\mathbf{x}_i$, resulting in the next perturbed example $\mathbf{x}_{i+1}$. The process ends when the model decision for $\mathbf{x}_{i+1}$ differs from its decision for the

original example $\mathbf{x}$. Finally, the total perturbation $\boldsymbol{\epsilon}$ is computed as the sum of all perturbations $\boldsymbol{\epsilon} = \sum \boldsymbol{\epsilon}_i$, resulting in the final perturbation value to produce the adversarial example $\mathbf{x}^* = \mathbf{x} + \boldsymbol{\epsilon}$. The DeepFool attack method produces adversarial examples with less perturbation than FGSM, but this method is computationally more expensive than FGSM.

In [115], the adversarial resilience of MLP classifier is assessed against adversarial attack methods including FGSM, Carlini Wagner, and DeepFool. The MLP classifier consists of two hidden layers with 256 units each, followed by ReLU activations and dropout regularization. The MLP classifier learned the classification patterns from the NSL-KDD [91] dataset. The adversarial examples are produced using all attacking methods, and the classifier is evaluated on the original and adversarial examples. The results show that the adversarial examples produced using DeepFool achieve higher misclassification rate than those produced with other attack methods.

- **JSM** [107] is a target adversarial attack that produces adversarial examples of a specific class. The JSM method produces adversarial examples by modifying a minimum number of features, measured using $L0$ norm. The JSM method computes saliency maps, which record the saliency values for each feature of an input example. These values show how much each feature perturbation will impact the model decision. The saliency values are computed using the gradients of the decision model output with respect to its input features. The Jacobian matrix of the decision model output with respect to its input is analyzed to assign saliency scores to features. The input features with the higher saliency score for the perturbation is discovered using the JSMA method iteratively. The features are ordered by decreasing the saliency value, and each selected feature is perturbed by a value θ, which either increases or decreases its value to move the model decisions closer to the target class. The process stops when a threshold number of modified features is reached or misclassification occurs. Although this technique is computationally expensive compared to FGSM since it requires the computation of the saliency values, it dramatically reduces the perturbed feature amount that is impacted for adversarial examples generation that appears to be more like the original sample.

- **ZOO** [108] is a black-box attack method to produce adversarial examples in scenarios where we can query the target model without having any knowledge of the internal structure of the model. Instead of directly accessing the gradients of the target model, the ZOO attack approximates gradients by iteratively adding small perturbations to the input and observing changes in the target model decisions probabilities. The ZOO attack transforms the adversarial noise into a lower-dimensional space for optimization and

then maps it back to the original input size using bilinear interpolation transformation. This dimensionality reduction and transformation improve the computational efficiency of adversarial noise optimization. Furthermore, hierarchical attacks perturb groups of features simultaneously, while importance sampling focuses on high-impact features.

In [116], malicious Uniform Resource Locator (URL) detection is done using a combination of lexical and Web-scrapping features. Lexical features include linguistic indicators (e.g., URL length, domain length, special character ratio) and human-engineered features (e.g., domain names, bag-of-words, n-grams). Web-scrapped features are collected through Google search results, including domain typo-squatting analysis, Levenshtein distance, Shannon entropy, WHOIS, IP, and content-based indicators. Several classifiers, including RF, Gradient Boost, AdaBoost, and XGBoost, learned these combined features. The evaluation results show that XGBoost achieves the highest classification accuracy. In addition, the adversarial examples are produced using ZOO black-box attack method, and evaluation results show that the adversarial examples against the RF classifier achieved higher evasion rate.

2.5.3 Defensive adversarial learning

This section includes the theory of the adversarial training, ensemble adversarial training, defensive distillation, randomization, denoising, and regularization.

- **Adversarial Training** [2] technique enhances the robustness of the decision models against adversarial attacks. It combines clean examples with adversarial examples and trains the decision model on this combined training set to increase the accuracy and robustness to adversaries. The adversarial training technique regularizes the decision model, thanks to the presence of adversarial examples in the training stage helping the model to learn more stable decision boundaries. Using adversarial training against FGSM on the MNIST dataset, the authors of [2] decrease the error rate from 89.4% to 17.9%.

  In [117], Android API calls are replaced by the system API sequences using depth-first search (DFS). Each system API sequence is mapped to a fixed RGB color value using a pre-constructed feature database, resulting in images of Android applications. The adversarial examples are produced using the Conditional Generation Adversarial Nets (CGAN) [118] method and added to the training set. The CGAN and LeNet-5 network, which classifies the images into malware and goodware are trained together to improve classification performance. The classification performance is evaluated using

accuracy and F1-score metrics. The evaluation results show that the adversarial training improved the classification performance of the LeNet-5.

In [119], the DNN processes the raw byte features extracted from Windows PE files. The architecture of DNN consists of three hidden layers, each with 1200 neurons. The adversarial examples are produced using the FGSM [2] attack method. The adversarial examples most likely to evade the DNN are selected based on feature maps extracted from the last fully connected layer. These selected adversarial examples are added to the training set, and the DNN is retrained on the augmented training set. The evaluation results show that the classification performance of the model is improved.

- **Ensemble Adversarial Training** [120] technique combines clean examples with obtained attacking multiple pre-trained models and trains the decision model on this combined training set. The training of the decision model on the combined training set, enriched with diverse adversarial examples, helps mitigate vulnerabilities associated with single-attack methods such as FGSM. The diversity of adversarial examples addresses the issue of sharp curvature in the loss landscape, which adversaries can leverage in their attack. The empirical evaluations shows that decision models trained with ensemble adversarial training increase their robustness against both single-step and multi-step adversarial attacks and outperform the models trained with traditional adversarial training methods.

- **Defensive distillation** [121] is used as a defense technique in adversarial settings for deep neural models. It improves the model capability of generalizing examples outside the training set and improves the smoothing of the trained model. Defensive distillation-trained models are more suited for deployment in security-sensitive environments because they are less vulnerable to adversarial examples. As shown in Figure 2.2, the defensive distillation includes a Teacher and Student network having the same architecture. The teaching network outputs have less confidence in the ground truth label and exhibit a more excellent dispersion across classes when the softmax layer of the neural network is trained at a higher temperature T. The temperature T is a parameter that controls the dispersion of class probabilities p_i produced by the softmax function. When $T = 1$, the Softmax function produces sharp probabilities, emphasizing the most likely class. When $T > 1$, it produces a more dispersed probability distribution across all classes, making softer the output. The mathematical formulation of the softmax function with temperature T is defined in equation 2.37.

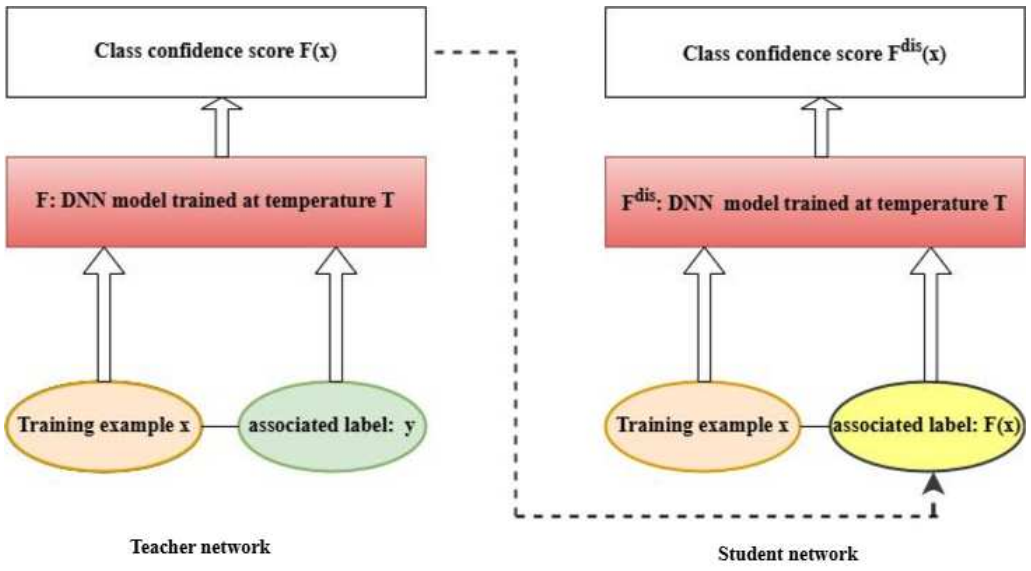


Figure 2.2: An overview of how defensive distillation works.

$$p_i = \frac{\exp(\text{logit}_i/T)}{\sum_j \exp(\text{logit}_j/T)} \tag{2.37}$$

where p_i is the probability of the $i-th$ logit, logit_i is the logit of the $i-th$ class, logit_j is the logit for each class, and $\sum_j$ ensures probabilities sum for all classes is equal to 1. The teacher network is first trained on the training set using the ground truth labels at a higher temperature T. It produces soft labels, which are a probability distribution across all classes, rather than hard labels. The student network is trained with the same training set using the soft labels produced by the teacher network at temperature T. Training the student network with soft labels should, in principle, make the student network more robust against adversarial examples.

- **Randomization** techniques mitigate the impact of adversarial examples on the model decision. The author of [122] applies random transformations, such as random resizing and random padding, to adversarial examples before training a CNN. The randomized transformations disrupt the adversarial perturbation pattern, and this makes more challenging for adversarial examples to deceive the CNN. The random transformation increases the CNN robustness to adversaries. The Random Self-Ensemble (RSE) [123] technique integrates random noise layers before each convolutional layer within a DNN. During the model training, each input example is processed once per step, with random noise added by the layers. At the inference time, the trained model passes the same input example (clean or adversarial) multiple times. Each pass applies a different random noise pat-

tern, resulting in a set of decisions for the same input example. The final decision is obtained by ensembling (i.e., averaging all decisions), with the effect of increasing the model robustness to adversaries. The Pixel-level Differential Privacy (PixelDP) [124] is a Differential privacy (DP)-based mechanism that of integrates a noise layer into the DNN, ensuring that small, norm-bounded perturbations of the input examples result in only minimal changes to the decision distribution. By bounding the sensitivity of decisions to input variations, the PixelDP provides certified robustness.

- **Denoising** removes unnecessary noise or perturbations from the input examples before feeding them to DNN model. The feature squeezing [125] is a denoising technique that squeezes the less critical features from the input example. The input example can be squeezed with the Bit-Depth reduction method and the Spatial smoothing method. The Bit-depth reduction method reduces the intensity levels per pixel by combining similar pixel values. The spatial smoothing method applies filters, such as median filtering, to reduce noise and remove small-scale perturbations. The feature squeezing technique passes both the original input example and the squeezed example of the same input example, comparing the model decision on the original input example with its decision on the squeezed example. The input example is adversarial if the difference in both decisions exceeds a threshold.

- **Regularization** Regularization enhances the generalization of DNN models by introducing constraints or penalties during the training stage. The input gradient regularization [126] technique trains the DNN model by applying a regularizer term to the input gradients along the objective function to produce smooth input gradients. The input gradient regularization technique penalizes extreme values in the gradients with respect to the input, by making the DNN model more stable to small perturbations, and increasing its robustness. The authors demonstrate that the input gradient regularization technique is equivalent to adversarial training, providing an effective alternative defense strategy.

Chapter 3

Background in Windows PE malware analysis

This chapter illustrates the Windows PE file format, static and dynamic analysis methods for elaborating Windows PE files, AI-based methods for malware detection and classification focusing the attention on Windows PE files, realistic adversarial attack methods targeting AI-based Windows PE malware detection methods, and benchmark datasets commonly used in Windows PE malware research.

3.1 Windows PE format

A Windows Portable Executable (PE) malware is a PE file with a malicious intention, e.g., an executable accessing the system without user permissions, stealing private or confidential information, or requiring a ransom [127]. A Windows PE malware is classified into various types based on its execution behaviour and intended functions, including viruses, worms, backdoors, trojan horses, spyware, botnets, rootkits, adware and ransomware. [128]. A virus infects healthy files by performing destructive actions like modifying or deleting system and user data. A worm spreads through networks or physical devices (e.g., USB drives). It damages the system by consuming resources. A backdoor creates unauthorized access points, often through the open network ports, enabling attackers to control the system remotely. A trojan horse pretends to be legitimate software installed on the victim machine with the user's knowledge while executing malicious activities in the background. A Spyware secretly monitors the user's activities. The botnet consists of multiple compromised machines controlled remotely by an attacker. Botnets perform denial-of-service attacks, spam campaigns, and coordinated malicious activities. The rootkit hides malicious activities by altering system-level functions and data structures. An Adware is downloaded with third-party software downloads and displays unwanted ads. Some adware files serve as trojans targeting user privacy, but all adware files are not malicious. A Ransomware encrypts files and demands a ransom.

The PE file is the standard binary format of DLL and Executables in the Microsoft Windows family [129]. Its layout contains a Header area, a Section area and the Unmapped Information area. As shown in Figure 3.1, the Header area contains the DOS Header, DOS Stub, PE Header, PE Optional Header and Section Table.

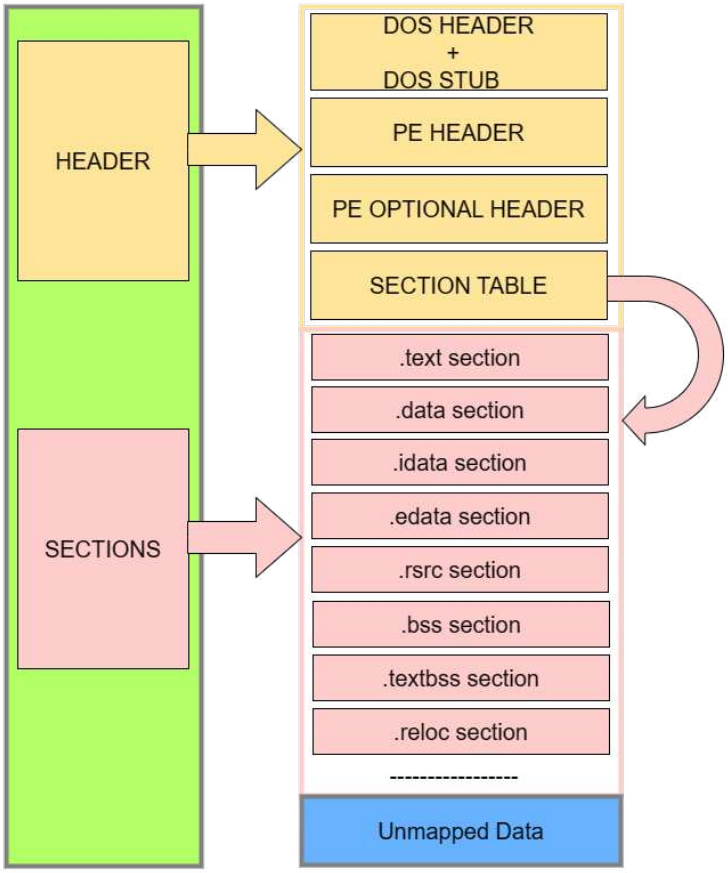


Figure 3.1: The structure of a PE file. The elements in yellow are included in the Header area. The elements in pink are included in the Section area. The elements in blue are included in the Unmapped Data Section area.

Both the DOS Header and DOS Stub are used for compatibility with the MS-DOS system. The DOS Header contains the unique file signature identifying the file as a valid PE format, and the PE Header starts offset. The PE Header contains the global information about the PE file. For example, it contains information about the type of processor needed to run the given file, the number of sections, and the section creation time. The PE Optional Header contains information to describe the alignment of the binary data. Notice that the binary data must be recorded in the executable code by satisfying the constraints that each section of the PE file must start at an offset multiple to that field, and the size of each header must be a

multiple of the file alignment. The PE Optional Header contains the offset pointers to find the functions in the Import Table or the Export Table. The Section Table provides global information about Sections such as names, offsets, and sizes. The Section area contains the consecutive sections with the executable code (.text) and required data (.data), which include global and static variables. The number of sections in the Section area is not fixed a priori. Additional sections can be used to report the information regarding the address and size of the import table (.idata) and export table (.edata), resource data such as icons and menus (.rsrc), uninitialised data (.bss), extended text on additional linking (.textbss) or relocation information (.reloc). The Unmapped Data Section contains all data that are recorded in the PE file, although these data are never loaded into memory during the program execution. Examples of unmapped data include uninitialised data and debug information.

3.2 Static and dynamic analysis of Windows PE files

Malware analysis is a process of identifying the functionalities of the Windows PE malware file, including understanding how it works, which data are targeted or destroyed, and which systems or programs are affected. Static and dynamic analysis are two primary approaches for extracting features from Windows PE files. The static analysis of a Windows PE file involves examining the structure, the metadata, and the code without executing it. The dynamic analysis observes the behaviour (e.g., the creation of new files, URL access, registry key changes, new log entries, execution of API calls, malware downloading) of the Windows PE files during the execution in a virtual controlled environment. The virtual environments may be created using virtualization tools such as VirtualBox [4] or Vmware [5]. Both static and dynamic analysis aim to recover the malware functionalities, such as how it operates, what data it targets, and which systems or programs it affects [127, 130].

In the static analysis of the Windows PE malware file, various tools are used to extract meaningful features from the file structure without executing it. These tools are classified into Graphical User Interface (GUI)-based tools and programming libraries. The GUI-based tools provide a visual interface to inspect the Windows PE malware file. Tools, such as PeView [131], provide the PE file header information, while PEid identifies the packer tool in case of malware obfuscation. Similarly, CFF Explorer [132] provides detailed header information and metadata for the PE file. Tools like Radare [133] perform reverse engineering for deeper inspection, and Disassemblers (e.g., IDA Pro, Ghidra) convert PE files into assembly code for manual analysis. Yara identifies malware variants using pattern matching across PE files [134].

For the large datasets of Windows PE files, the manual static analysis using GUI tools becomes impractical. The Library to Instrument Executable Formats (LIEF) [6], is an open-

source Python library that allows us to parse, manipulate and analyse executable file formats, including Windows PE files. The LIEF parse the Windows PE file structure starts from the DOS Header, following the PE Header, Optional Header, and Section Table. In addition, it allows us to extract specific features, such as imported libraries and functions, byte histograms, byte-entropy distributions, header properties, and section-level metadata. LIEF also provides functionalities for modifying the PE files, including the functionalities for the addition of new sections, the modification to existing headers, or the payload injection.

The advantage of the static analysis is that it is fast and less time-consuming. The disadvantages of the static analysis include high false positive rates, less effectiveness in detecting behavioural anomalies, and vulnerability to obfuscation techniques. In addition, machine learning models trained on features extracted using static analysis are often less robust against new malware variants.

In the dynamic analysis, various manual and automated tools provide the activity information of the Windows PE file during runtime in an isolated environment. In the manual analysis scenario, Process Explorer [135] provides several information, including the running processes, CPU usage, memory consumption, and the loaded DLL. For example, Process Monitor (ProcMon) [136] detects real-time activities, including file system interactions, registry modifications, and network calls. TCPView [137]shows the connection created by the malware through active User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) endpoints. Regshot [138] records snapshots of the system registry before and after malware execution, which shows registry modifications. Wireshark and Tshark [139] analyse the network activity of Windows PE malware by capturing incoming and outgoing traffic. Memoryze [140] captures complete memory dumps after running the malware and provides information about the hidden and running processes. Volatility [141] is a Python-based framework that analyses memory dumps, extracting process details, DLL lists, and specific strings embedded in malware. An automated sandbox tool such as Cuckoo Sandbox [142] is the most suitable for analysing a large number of Windows PE malware files. Cuckoo Sandbox provides an isolated environment for executing the Windows PE malware files. It generates a detailed report indicating the registry changes, network communications, and system calls observed during malware execution [143].

The dynamic analysis has several advantages. It is more robust against obfuscation techniques. It can identify hidden functionalities of malware that execute only under specific conditions or time delays. In addition, the dynamic analysis can detect malware using packing or encryption techniques by observing its behaviour during runtime when the malware unpacks and executes itself. However, the dynamic analysis has also some disadvantages. It is slower than the static analysis, as it requires time for malware execution to observe its behaviour. Ad-

vanced malware files can recognize that they are run in sandbox environments and alter their behaviour to evade detection. In addition, the dynamic analysis requires significant computational and memory resources to run Windows PE malware files in isolated environments.

3.3 Windows PE malware detection and classification

This section illustrates the state of the art of the signature-based, behaviour-based, and hybrid-based malware detection with attention to the use of AI in approaches for Windows PE malware detection and classification.

The Windows PE malware detection approaches are divided into three categories: signature-based, behaviour-based, and hybrid-based.

The Signature-based malware detection approach identifies the PE malware file based on its static properties, such as specific signatures or patterns derived from the structure or content. This approach is one of the most traditional and widely adopted methods for malware detection, particularly in antivirus software. The hash algorithms, such as MD5 [7] and SHA1 [8], produce signatures (unique hash values) for each Windows PE malware file. These signatures are stored in a centralized database. The signature of an unknown Windows PE file is produced and compared with the signatures stored in a database. If a match is found, the PE file is flagged as malware; otherwise, it is considered goodware. The signature-based approach has some limitations. A single-byte modification in a Windows PE malware file changes the file signature. Every variant of the Windows PE malware file requires a new signature to be produced and added to the database to detect it. The signature-based detection approach is less effective against obfuscated Windows PE malware files, where PE malware variants dynamically change their code to evade detection [144].

The Behaviour-based malware detection approach identifies the Windows PE malware files based on the behaviour of each PE file. The activities of the Windows PE malware file, including new file creation, modification, deletion, registry changes, system calls, network activities, and process interactions, are monitored in an isolated environment. The Windows PE file is detected as a malware if its behaviour differs significantly from the normal one. The PE file behaviour can be observed using manual or automated tools described in Section 3.2. The behaviour-based detection approach can detect unseen or obfuscated Windows PE malware file variants [145]. The main shortcoming of this approach is the high false-positive rate. In addition, the computational cost is also a challenge in the actual implementation of the behaviour-based detection method.

The Hybrid-based malware detection approach combines signature-based and behaviour-based approaches to detect Windows PE malware files. This approach detect known Windows

PE malware by comparing the file signature with the signature of known malware recorded in an existing database. The Windows PE file is declared as a malware if a match is found. If no match is found, then the second stage is performed by using a behaviour-based detection approach that uses dynamic analysis to detect the Windows PE file as malware or goodware. The hybrid-based approach increases the capability of detecting known, unknown, and obfuscated Windows PE malware variants and minimizes the false positive rate [130].

With regard to the use of AI in Windows PE malware detection, various studies have investigated the performance of machine learning and deep learning methods used to recognize malware from a feature-vector representation of Windows PE files. This feature vector representation is commonly obtained with an engineering phase that includes also static and dynamic analysis.

3.3.1 Machine learning-based malware detection methods

In [146], the behavioural analysis of Windows PE files is performed using the Cuckoo Sandbox to extract API system calls, Printable String Information (PSI), file operations, registry changes, network activities, dropped files, and mutexes. Extracted tokens are converted in count features. The Singular Value Decomposition (SVD) [9] is used for feature selection, while various supervised machine learning algorithms (e.g., the ML algorithms, such as RF, DT, SVM, KNN, NB, Gradient Boosting, and AdaBoost) are used to train a classification model that classifies Windows PE files as malware or goodware.

In [147], an ensemble learning method is used for Windows PE file classification. The method is two-stepped. In the first step, three classification models, i.e., a MLP with one fully connected hidden layer, a MLP with two fully connected hidden layers, and a 1-D CNN, are trained using static features to produce initial decisions. In the second step, several classification models including DT, SVM, RF, KNN, MLP, AdaBoost, ExtraTrees (ET), and NB are trained using the modified dataset to produce the final decisions. The principal component analysis (PCA) is used to remove collinearity and reduce the size of the data used for the training stage.

In [148], an ensemble learning method is formulated. It considers four data views composed of; DLL, API functions, PE Header, and PE Sections. In addition, it considers two further views: Combined Set1, which combines DLL and API functions, and Combined Set2, which combines PE Header and PE Section features. A feature selection step is performed by using IG and PCA. Several classification models are learned by using the following classification algorithms: NB, DT, SVM, RF, KNN, and Gradient Boosting, which are individually trained

and evaluated on original features, selected features, and integrated features. Moreover, an ensemble is obtained using the majority rule.

In [149], the Windows PE metamorphic malware files are executed in a sandbox environment, and API call sequences are extracted and converted into a positive integer code. The sequential pattern mining algorithm CM-SPAM is used for discovering frequent API call patterns. The sequential rules are explored using ERMiner. Compact representations (e.g., closed and maximal patterns) are discovered using Closed FAST (CloFast) and Vertical mining of Maximal Sequential Patterns (VMSP). The discovered patterns in integer code are learned by the five algorithms: RF, SVM, KNN, NB, J48, and in string format are learned by the Stochastic Gradient Descent Text (SGDT) and Naive Bayes Multinomial Text (NBMT). The experimental results confirm that string-based classifiers achieve higher accuracy compared to integer code-based classifiers.

In [150], the malware detection is done based on the static sequential opcode features. Opcode graphs are extracted from Windows PE files. The graph size is reduced by selecting only high-weight edges and nodes with the highest degrees. The nodes show the unique opcodes, and the edges show their sequential relationship. The short opcode sequences are extracted from the reduced graphs using the weighted random walk. Each opcode in a short sequence is converted into a low-dimensional numerical vector using the Skip-Gram model [151], and the vectors are concatenated to represent the full sequence. The classifification patterns are learned using SVM, CNN, KNN, and LSTM algorithms, and the classification performance is evaluated with accuracy, true positive rate, and false positive rate metrics.

In [152], MD5 hash, Optional Header size, and Configuration Load Size features are extracted from Windows PE files through static analysis. Supervised machine learning methods, including LR, DT, RF, Gradient Boosting, AdaBoost, XGBoost, and unsupervised methods such as PCA, and K-means clustering learn patterns to detect PE files as malware and goodware. The class imbalance issue is addressed by applying several balancing methods (e.g., undersampling, oversampling, balanced bagging classifier, and SMOTE). The classifier's evaluation performance is evaluated with accuracy and F1-score metrics. The RF classifier achieved the highest accuracy.

In [153], the LightGBM classifier is used to learn the static features extracted from Windows PE files, which are extracted using the LIEF library. The LightGBM classifier performance is evaluated for detecting malware with default parameters and optimized parameters using two Automated Machine Learning (AutoML) frameworks: AutoGluon-Tabular and Microsoft Neural Network Intelligence (NNI). The results show that LightGBM with optimized parameters outperforms the version with default parameters.

In [154], Windows PE files are executed in the Cuckoo Sandbox, and the API call sequences are extracted, grouped by process ID. They are converted into integer-mapped sequences. The malware files belong to six malware families, and an individual Hidden Markov Model (HMM) model is trained on the API sequences of each malware family. An α-pass algorithm [155] is used to compute the log-likelihood at each observation point in the sequence, producing a log-likelihood curve. The sub-curves are extracted based on the slope discontinuities of the curve using a window-based thresholding. Each sub-curve is scored using two features: the log-likelihood per observation, and the area between curves, which measures the deviation between a sub-curve from the test sequence and the corresponding region of the curve produced from the training files of the same HMM. Several classification algorithms, such as J48, RF, and SVM, are trained using the sub-curve features, and the performance is evaluated on classifying Windows PE files as malware or benign. The RF classifier achieves the highest accuracy.

In [156], five low-dimensional static features (i.e., 2-gram byte sequences, 2-gram matrices of frequent opcodes, API features, API-DLL features, and a window entropy map) are extracted from Windows PE files. The Opcode feature is extracted using the Radare2 disassembler. The API and DLL features are produced by frequency analysis. The window entropy map feature is a byte-level entropy feature obtained using sliding windows and entropy quantization. The extracted features are processed to train five tree-based ensemble classifiers, using AdaBoost, XGBoost, RF, Extra Trees, and Rotation Forest. The classification performance is evaluated using accuracy, precision, recall, and AUC metrics. The experimental results confirm that tree-based ensemble classifiers trained with low-dimensional features outperform classifiers trained with high-dimensional features.

In [157], API call sequences are extracted from Windows PE files by performing dynamic analysis in a sandbox environment. The contextual relationships between API functions of malware files and goodware files are obtained using the word2vec word embedding [158] technique. Similar API functions are grouped using the k-means clustering algorithm. The clusters form states of a Markov chain model, and the transition probabilities between these states are calculated for malware and goodware files using the maximum likelihood estimation [159]. This results in two distinct transition matrices: malware transitions and goodware transitions. Any unseen Windows PE file is classified as goodware or malware based on the likelihood of state transition of the file against transition metrics. The accuracy performance analysis shows that classification with contextual and clustering methods outperforms without these techniques.

In [160], API call sequences with associated arguments are extracted from Windows PE files using the Cuckoo Sandbox environment. API call features are extracted using two methods: (a) considering each API call and its list of arguments as a single feature, and (b) con-

sidering each argument of each API call as an individual feature. The extracted features are encoded into fixed-length bit vectors using a Hashing Vectorizer function. Five machine learning classifiers are learned from the extracted features using: SVM, XGBoost, RF, DT, and Passive-Aggressive (PA). The evaluation study shows that malware classification using these feature extraction methods outperforms API sequence-based or frequency-based feature extraction methods, particularly achieving better performance against argument manipulation, sequence tampering, and mutation attacks.

In [161], the API call features are dynamically extracted from the Windows PE files using the Cuckoo Sandbox environment. In addition, after executing the files, import address table entries are statically extracted from the memory dumps by applying the Impscan plugin of the Volatility framework. The API call features and memory-based artifacts are combined, and several machine learning classifiers are trained with SVM, RF, DT, NB, and KNN from this vector of features. The classification performance is evaluated with accuracy, detection rate, false positive rate (FPR), and false negative rate (FNR) metrics. The SVM classifier achieves the highest accuracy. The results show that classifiers trained with combined features outperform classifiers trained using only API features or memory-based artifacts only.

In [162], the frequency of API calls is recorded for each Windows PE file by executing it in a Cuckoo Sandbox environment. Several machine learning classifiers, including LR, KNN, CART, NB, SVM, DT, RF, and Linear Discriminant Analysis (LDA) are used to learn a classifier from the API call frequency features and classify PE file as malware or goodware. The classifiers are evaluated using accuracy, true positivve rate (TPR), and FPR metrics. The SVM and KNN classifiers achieve the highest accuracy. The results confirm that classifiers outperform on API call frequencies compared to approaches relying only on the existence of API call features.

In [163], static API call sequences are extracted from the PE file structure, and the dynamic API sequences are extracted by executing Windows PE files in a Cuckoo Sandbox environment. The same behavior type API calls are grouped into semantic blocks (e.g., registry operations, file operations, process operations, and network activities). The common APIs from static and dynamic analysis are assigned higher scores, which show their reliability. The importance of each API call is computed using the Frequency–Inverse Document Frequency (TF-IDF) technique. The APIs with higher contribution scores are selected, and a fixed-size feature vector is produced by combining static and dynamic information. The machine learning classifiers, including DT, RF, KNN, and XGBoost, are trained using these combined feature vectors to classify PE files as malware or goodware. The classification performance is evaluated using accuracy and F1-score metrics. The evaluation results show that classifiers trained with com-

bined static and dynamic features outperform classifiers relying only on static or dynamic API sequences.

3.3.2 Deep learning-based malware detection methods

In [164], the binary contents of the Windows PE files are transformed into two-dimensional matrices, resulting in grayscale images where each pixel represents a byte value. The produced grayscale images are resized to a fixed size. The class imbalance issue in malware families is addressed by producing additional images using data augmentation techniques (e.g., shifting, rotation, and flipping). In this study, a combination of a pre-trained VGG16 network and a Bidirectional Long Short-Term Memory (BiLSTM) architecture is used for classification. VGG16 extracts high-level spatial features from the malware images, and the BiLSTM layers capture sequential dependencies within the extracted features. The outputs from the BiLSTM layers are concatenated and passed through the dense layers, followed by a softmax activation function. The evaluation shows that the ConRec method outperforms by combining convolutional feature extraction and sequential modeling, compared to relying solely on a single CNN architecture.

In [165], the static features: machine type, section entropy, image base, sections size, import and export directory attributes are extracted by parsing the PE headers of Windows PE files. Similar PE files are grouped by applying the k-means clustering algorithm, and pseudo labels are produced. The number of clusters is obtained using the Elbow method and the Silhouette analysis. A Feature Attention-based Neural Network (FANN) is trained using the pseudo-labeled data. The FANN architecture consists of three dense hidden layers with an embedded attention block after the first hidden layer. The attention block assigns higher weights to important features by using sigmoid and ReLU activations in parallel. The evaluation show that the classification of PE files is improved by combining feature attention with clustering-based pseudo-labeling.

In [166], the API call sequences and their run-time parameters are extracted from Windows PE files through dynamic analysis. Each API call is labeled, showing the risk level of its run-time parameters. The labels are assigned using a rule-based classification method for known parameters and a clustering-based method for unknown parameters. The semantic embeddings of API calls are produced based on their contextual relationships, and label embeddings are produced by encoding the risk level of run-time parameters. Semantic embeddings and label embeddings are fused into a hybrid embedding vector. Text-CNN and BiLSTM classifiers are trained on hybrid embeddings to classify PE files as malware or goodware. Evaluation

shows that integrating run-time parameters improves the classification performance, particularly against adversarial attacks.

In [167], the byte values of Windows PE files are mapped to pixel intensities, resulting in grayscale images. The input size effect on classification performance is evaluated by resizing the images to dimensions ranging from 32×32 to 224×224. The proposed IMCLNet algorithm for malware family classification consists of three key components: coordinate attention (CA), depthwise separable convolution (DSC), and global context embedding (GCE). The CA module localizes discriminative regions in the image by encoding spatial and channel-wise information. It captures long-range dependencies through average pooling along the horizontal and vertical axes, followed by a shared transformation and separate attention weights for each spatial direction. The DSC module reduces the parameters and computational cost by factorizing standard convolutions into a depthwise convolution followed by a pointwise convolution. The GCE module combines shallow, mid-level, and deep feature maps from different network layers and forms a unified global representation. The classification performance of IMCLNet is evaluated using accuracy and F1-score metrics.

In [168], the performance of the MalConv and TextCNN algorithms is evaluated using time-evolved Windows PE malware files collected from VirusTotal. MalConv learns a classification pattern from the raw bytes of PE files, while TextCNN learns a classification pattern from opcode features extracted through disassembly using the IDA tool. Both algorithms are trained on malware samples collected in 2017 and tested on samples from 2018 and 2019. The results show a significant drop in classification performance on the newer files, indicating poor generalization to time-evolving files. To address this issue, a two-headed neural network architecture is appended to the frozen feature extractor of each base algorithm. Each head outputs class probabilities using the softmax function, and the difference between the two outputs is used as a measure of sample abnormality. If this difference exceeds a predefined threshold, the file is considered abnormal (i.e., time-evolved). The detected abnormal files are filtered from the test set, and the classification performance of the base algorithms is re-evaluated on the remaining testing files using accuracy, F1-score, and AUC metrics. The results show that the classification accuracy performance is improved by identifying and filtering time-evolved files without retraining the original algorithms.

In [169], a parallel DL algorithm named PDL-FEMC is proposed to classify Windows PE files as goodware or malware. The architecture of PDL-FEMC consists of five hidden layers, where the first three layers work as an unsupervised autoencoder to extract low-dimensional features, and the remaining two layers work as a supervised classifier. The autoencoder extracts features from Windows PE files, which are then learned by the classifier to classify a PE file as malware or goodware. The training of the PDL-FEMC algorithm is optimized using a hybrid

method that combines backpropagation and Particle Swarm Optimization (PSO) [170]. Backpropagation updates local parameters using gradient descent, while PSO performs a global search to avoid local minima and enhance convergence. The computational efficiency is improved by dividing the training set across multiple processors, each of which trains a separate replica of the algorithm on its assigned data partition. PSO is then applied to the output obtained from these parallel replicas to determine a global optimal solution. The accuracy performance of the PDL-FEMC algorithm is evaluated on five datasets: DREBIN, VIRUSSHARE_TOP, ANDROID_ML, BODMAS, and CIC-AndMal-2020. The results show that optimizing the algorithm using the combined backpropagation and PSO strategy improves the malware classification accuracy.

In [171], static features are extracted from Windows PE malware files using hexadecimal and assembly source code views. The hexadecimal view extracts byte-based features (e.g., metadata, byte unigrams, entropy statistics, local binary patterns, and Haralick). The assembly source code view extracts assembly-based features (e.g., metadata, opcode unigrams, register usage, symbol frequency, and pixel intensity). In addition, deep features are extracted using CNN-based algorithms trained individually on different input modalities, including byte N-gram, opcode N-gram, grayscale images, and entropy. The output from each algorithm is converted into a fixed-length vector. The hand-crafted and deep features are concatenated into a single combined representation using an early fusion strategy. The classifier learns these combined features using the XGBoost algorithm to classify PE files into malware families. The evaluation show that combining hand-crafted and deep features allows the proposed approach to achieve higher accuracy than ensemble-based and post-processing approaches.

In [172], API call sequences are extracted from Windows PE files using the Cuckoo Sandbox environment. Each API call is represented by its name and its arguments, where arguments are either strings or integers. The API names are embedded using the Word2Vec method, integer arguments are encoded using feature hashing, and string arguments such as file paths, DLLs, registry keys, URLs, and IP addresses are encoded using a similarity encoding method. In addition, statistical features are extracted from printable strings, including counts, average length, entropy, and "MZ" header occurrences. These encoding strategies result in fixed-size vectors. An API call graph is produced, where nodes represent API names, edges represent call order, and API argument features are edge attributes. The semantic and structural information of the API call graph is processed to train a Graph Neural Network (GNN) algorithm, composed of two modules named Graph Isomorphism Network (GINE) [173] and Graph Attention Network (GAT) [174]. The GINE layer aggregates node and edge information, while the GAT layer captures structural relationships through attention mechanisms. A gated pooling layer [175] is applied after each module to retain only the top nodes based on the learnable

scores. The global graph representations are obtained by concatenating max and mean pooling, and fed into an MLP classifier for binary and multiclass classification. The evaluation results show that combining API arguments and structural features contributes to gain classification accuracy.

In [176], the bytes of Windows PE malware files are converted into grayscale images using the bytes to image (B2IMG) method. In this conversion, the hexadecimal values are parsed from PE files, invalid entries are removed, and the remaining byte values are mapped to pixel intensities ranging from 0 to 255. The class imbalance issue is addressed by producing synthetic PE malware images of underrated classes using the Cycle-GAN method and adding them to the training set. The DenseNet-121 classifier with two fully connected layers is trained on the PE images. The evaluation results show that integrating the B2IMG transformation and CycleGAN-based augmentation improves the accuracy of the classifier, particularly in handling imbalanced malware classes.

In [177], four types of features, that is, PE-Header, PE-imports, PE-image, and API call sequence are extracted from Windows PE files. PE-Header and PE-imports features, are extracted using Cuckoo Sandbox reports. A PE-image feature is produced by transforming raw bytes of PE files into 32×32 grayscale images. The API call sequences feature is produced by executing the PE files in the Cuckoo Sandbox environment. An individual DNN is specified for each feature type. A DNN architecture, that includes five fully connected (FC) layers, is trained using PE-Header and PE-Imports features. A 1-D CNN, that includes three convolutional layers and a max pooling layer, is trained using the PE-image feature type. A LSTM algorithm with global attention mechanism is trained using API call sequences. In addition, three fine-tuned EfficientNet networks (B0, B1, and B2) are used in parallel to enhance image-based malware detection. The output of all models is concatenated and fed to FC layers for both binary and multiclass classification. The accuracy of the classification process results show that combining static, dynamic, and image-based features improves the accuracy of the classification process.

In [178], the assembly instructions are extracted from Windows PE files using the objdump disassembler. The resulting assembly instructions are treated as sequences structured as a sentence or a full document. The three datasets are produced from these sequences (e.g., a document-level dataset, an entire assembly listing of executables, a sentence-level dataset, assembly instruction sequences, and an unlabeled instruction dataset). A BiLSTM architecture is trained on the document-level dataset. Both BiLSTM, DistilBERT, and GPT-2 models are trained on the sentence-level dataset. The GPT-2 general purpose and GPT-2 domain specific are trained on unlabeled instruction sequences. The evaluation shows that BiLSTM trained on the full document outperforms all sentence-level algorithms in terms of accuracy performance.

In [179], 324 bytes are extracted from the PE Header and converted into integer values ranging between 0 and 255. This feature vector representation is used to train a deep neural network. The K-means clustering with $k = 2$, is finally, used on the embedding representation outputted by the last layer of the deep neural network; to group similar examples into clusters. Based on the majority class associated to the examples grouped in each cluster, a cluster is labelled as goodware and the other cluster is labeled as malware. These cluster centres represent the malware and goodware prototypes, respectively. A new Windows PE file is classified according to the class associated to closest cluster centre by conmputing the Euclidean distance in the deep embedding space used for the clustering step.

In [180], a Graph-based method is presented. It basis on the analysis of the Control Flow Graph (CFG) structure of Windows PE files. This method consists of three modules: Graph Feature Extraction (GFE), Graph Data Generation (GDG), and Graph Classification (GC). The GFE module performs the following tasks: (1) extracting CFG structural information from Windows PE files using the open-source Python library Angr [10]; (2) preserving the CFG structure and assigning unique names to each block; and (3) transforming the CFG into a Function Call Graph (FCG). The CFG is represented as a directed graph that excludes the assembly code of each block. The GDG module maps the text information of each node of the FCG into a 384-dimensional vector using the pre-trained Large Language Model MiniLM [181], provided by the SentenceTransformer [11] library. Then, it embeds the 384-dimensional vectors into the corresponding nodes to produce complete graph data using the NetworkX [12] library. The GC module produces the fixed-length graph-level representations using a Graph Isomorphism Network (GIN) model consisting of three GINConv (GIN Convolutional) layers. The 384-dimensional vectors produced by the GDG module serve as input to the first GINConv layer. The classification model is, finally, trained to learn graph-level representations for classifying Windows PE files as malware or goodware.

3.4 Evasion Windows PE methods

As the use of AI-based systems for detecting Windows PE malware continues to grow, adversarial learning has become an important area of research. Attackers may leverage the vulnerabilities of AI-based detection models to create adversarial Windows PE files by injecting carefully chosen content into unused or modifiable places within the Windows PE files. This allows attackers to obtain adversarial malware to evade the AI-based detection system while preserving the malicious functionality of the PE file.

The adversarial attack methods synthesised for imagery and tabular data [182] cannot be used for Windows PE files since realistic adversarial Windows PE malware files must pre-

serve a Windows PE-compatible format (format-preserving property), obtain an executable file (executable-preserving property) and maintain the malicious nature of the executable file (maliciousness-preserving property). The format-preserving property implies that adversarial modifications must generate a binary file that complies with the standard format of Windows PE files. The executable-preserving property ensures that adversarial Windows PE malware loads all the necessary data and can be executed. The maliciousness-preserving property means that adversarial Windows PE malware maintains the same malicious behaviour (e.g., modifying registry items and deleting or encrypting files) as the original Windows PE malware.

This section illustrates the theory of the evasion methods to generate adversarial Windows PE white-box attack methods, including Kolo-padding, Full DOS, Extend, Shift, FGSM (padding + slack), Byte-prototype, GAMNBD, and a black-box attack method, namely Genetic Adversarial Machine learning Malware Attack (GAMMA).

- **Kolo-padding** [183] is a white-box method for producing Windows PE adversarial malware files. It considers MalConv [53], which is already described in Chapter 2 (section 2.3) a target model of the attack. In this method, the maximum input size of the target model is considered 1 MB, and up to 10,000 bytes are designated as modifiable. To preserve the file functionality, the bytes are added at the end of the malware file in the input space. Two strategies are used to select the adding bytes to add: random selection and gradient-based optimization. In the random selection strategy, random byte values are added directly at the end of the PE file in the input space, and the file is passed through the MalConv to classify it as malware or goodware. In the gradient-based optimization strategy, zero-valued bytes are added at the end of the PE file. The malware file is passed through the embedding layer of the MalConv and transformed into an 8-d matrix in embedding space. Each row in this matrix represents an embedding vector corresponding to a byte in the input space. The malware file in the embedding space is referred to as the perturbed malware file. The loss of the perturbed malware file as malware and goodware is computed using the loss function. The optimization goal is to reduce the MalConv confidence in classifying the file as malware, with a target confidence threshold <0.5. In the embedding space, the added bytes are optimized individually using the gradient descent algorithm. The gradient is computed one time for each individual added byte. The byte values are adjusted iteratively within the possible range [0-255], and the Euclidean distance is computed between the current adjusted byte value (perturbed vector) and the embedding vector that is already part of the matrix in embedding space in each iteration. The perturbed vector with minimum distance is selected. This selected perturbed vector is actually a new byte value. After optimizing the added bytes in the embedding space, these bytes are replaced with the previously added bytes in the input space. The itera-

tive process of modifying the malware file in the input space and passing it through the MalConv for classification continues until the model confidence in classifying the PE file as malware achieves a threshold or the number of iterations reaches its maximum limit of 20. Notably, in the embedding space the gradient is computed in one step, while the modification in input space is performed iteratively.

- **Full DOS** [1] method modifies the bytes inside the DOS Header of the Windows PE malware file to produce the adversarial file. It considers MalConv [53], DNN-Lin [13], DNN-ReLU [184], and Gradient Boosting Decision Tree (GBDT) as target models. It is based on the fact that the DOS header is still kept in Windows PE files to make these files still compatible with the older operating systems. In fact, the DOS header may be changed by keeping the functionality of the executable file except for the magic number "MZ" and the four-byte-long integer at offset "$0 \times 3c$". In particular, the magic number identifies the file uniquely, while the four-byte-long integer at offset "$0 \times 3c$" points to the starting point of the PE header in the binary code. Both cannot be changed in order to obtain an executable file. Hence, this attack method perturbs the bytes that are placed in the DOS header in the areas before the magic number and after the pointer to the PE header. The method begins by assigning the magic number location and the PE offset of the malware file to the binary mask to preserve the functionality of the file. The PE file in the input space is transformed into a perturbed PE file by passing through the embedding layer, and the loss is computed. The gradient is computed for each individual byte of the perturbed file DOS Header except the binary mask. The norm of the gradient value for each byte is computed and ranked. This is an indication of how likely modifying a particular byte is expected to affect the model output. In this way, only influential bytes are considered rather than attempting to adjust the value of every byte. The bytes with the first non-zero norms (largest norms) are selected. These selected bytes are the more influential for modification. The optimization process of each individual byte is similar to that described in [183]. After the optimization, the embedded space bytes are replaced with the corresponding input space bytes. The process continues until either the convergence or the maximum number of iterations are reached. Since each embedded byte computed in this way has a one-to-one mapping with the input space byte, this process returns an optimized adversarial malware file. The number of perturbed bytes in the DOS Header varies between 118 and 219 bytes.

- **Extend** [1] method creates a new area within the binary file by increasing the size of the DOS Header. It uses the new area in the DOS header to add the perturbed bytes. This byte injection is done by keeping the functionality of the executable file. Initially,

the number of bytes to be perturbed is set to 512. The Extend method operates in four steps. First, the current offset of the PE header in the file is retrieved. In the DOS header, a specific field typically located at offset "0 $\times$ 3c" contains the PE header offset. This offset indicates the beginning of the PE header within the file. In a typical PE file, the offset is set so that the PE header follows immediately after the DOS stub. However, since this value is just an offset, it is modified to point to a different location in the file. By increasing this value, a gap is inserted between the DOS stub and the actual start of the PE header. Second, the new offset is calculated by adding the initialized bytes to be perturbed to the current offset, and the PE header offset is updated with the new offset. According to the new offset, the PE header is physically relocated (pushed down) within the file. Adding initialized bytes in PE offset means adding a space between the DOS stub and the PE header. This space is filled with the null bytes equal to the number of initialized bytes to avoid changing the behavior of the file; later, these null byte spaces are replaced with optimized perturbed bytes. Third, after adding space, the relevant field is modified to preserve the structure of the PE file. The "SizeOfHeaders" field in the PE header indicates the total size of all headers (DOS header, PE header, and section headers) and informs the loader about the memory needed to load the executable file headers. This field is adjusted by increasing the header size for the added space. By adding the space between the DOS stub and the PE header, the bytes, that originally came after the PE Header, is pushed further down in the file. The sections following the PE header are also moved down due to the added space. To correctly reflect this change, each section offset in the Section Table of the PE header is increased by the same amount as the PE header offset. Fourth, after creating the new space inside the executable file, the modifications are made to this newly created space for producing adversarial malware files. The adversarial malware files producing method in Extendis like the Full DOS attack method; the only difference is a location where perturbed bytes are replaced with the corresponding bytes in the input space. The target models considered in the Extend method are the same models used in the Full DOS method.

- **Shift** [1] method applies the shift operation to the first section and creates a space to add the perturbed bytes in the newly created space for producing adversarial malware files. The functionality of the produced adversarial malware file is preserved by adding the perturbed bytes at the modifiable space and the amount of the perturbed bytes that align with the file alignment. The added perturbed bytes are 1024. The Shift method operates in three steps. First, the position of the first Section in the binary file is determined. Each section in a PE file has an entry in the Section Table, which specifies its offset within the

binary file. This offset is where the loader will find the section content in the file. Each section offset is a multiple of the file alignment value specified in the Optional Header. The size of the perturbed bytes should be a multiple of the file alignment to ensure that the subsequent sections' offsets remain valid multiples of the file alignment. If the perturbed bytes does not align with the file alignment value, then the method performs the padding of null bytes to reach the nearest multiple of the file alignment value. The file alignment determines the alignment of the raw data of sections on disk. Common file alignment values are 512, 1024, 2048, or 4096 bytes, corresponding to disk sector and memory page sizes. Second, null bytes equal to the amount of desired bytes to be perturbed are added at the position of the first section. The insertion of the null bytes pushes down the content of the first section inside the file and creates a new space before the start of the first section. This newly created space is used for injecting perturbed bytes. Sections are contiguous chunks of data in the PE file; pushing down the beginning of the first section pushes down the content of all the following sections within the file. The physical offset of each section in the section header is increased by adding the same number of null bytes that were added before in place of the content of the first section to reflect the new starting positions of the sections. Third, after creating the new space before the content of the first section inside the executable file in the input space, the modifications are made to this newly created space for producing adversarial malware files. The adversarial Windows PE malware files produced by the Shift attack differ from the adversarial PE files created by Full DOS in the location where perturbed bytes are injected. The target models considered in the Shift method are the same models used in the Full DOS method.

- **FGSM (padding+slack)** [185] is a white-box attack method for producing Windows PE adversarial files. The target model is MalConv [53]. The method begins by adding random null bytes at the end of the PE file. The added byte size is limited to 999 bytes. The PE file is transformed into a perturbed file in the embedding space of MalConv [53] that is the target model of this attack method. The model confidence is computed for the perturbed file. The goal is to decrease the model confidence for classifying a malware file as malware until a threshed ≤ 0.5. Then, the gradients of the perturbed file are computed, and the embedded vector values of the added bytes are adjusted using FGSM, which is described in Chapter 2 (section 2.5). The embedding of the new perturbed file is passed again to the model, and the confidence is computed. This is an iterative process that continues until the confidence of the model for the new perturbed malware file reaches a threshold. Next, the optimized embedding vectors of the added bytes are converted back

to the byte space by selecting the closest aligned byte values based on the Euclidean distance. In this way, the PE file is reconstructed in the input space. Notably, in this method, the file is reconstructed in the input space once. In addition, this method also produces adversarial malware files by adding bytes in slack spaces. The slack spaces are the spaces between the sections.

- **Byte-prototype** [186] is a gradient-based method that uses the byte prototype as guidance for producing the Windows PE adversarial malware files. It considers MalConv [53] as the target model. This method works in two stages. In the first stage, the byte prototype is produced, and in the second stage, the produced byte prototype is used as guidance to produce the Windows PE adversarial malware file that closely aligns with the byte prototype. In the first stage, the byte prototype is produced as follows: The Windows PE malware file is transformed into an embedding vector by passing through the embedding layer of the target model. This embedding vector is optimized using a gradient descent algorithm. The optimization process is guided by maximizing the activations of the output layer of the target model for the goodware class using the activation maximization [187] method. The activation maximization method visualizes the activations of each layer in a trained ConvNet model. However, the purpose of this method in producing the byte prototype is to adjust the embedding vector effectively so that the target model confidence in classifying the malware file as a goodware file increases. After optimizing the embedding vector, the vector is transformed back into a byte prototype in the byte space using a conversion method similar to that described in [53]. This iterative process continues until the byte prototype misclassifies as a goodware file or reaches a maximum of 15 iterations. Notably, for each iteration of converting the embedding vector to a byte prototype, the embedding vector is optimized 200 times.

  In the second stage, the Windows PE malware file is modified based on the byte prototype guidance into an adversarial Windows PE file by using two strategies: (1) the slack spaces between the sections and the padded bytes at the end of the Windows PE malware file are replaced by the corresponding bytes of the byte prototype; (2) the sections in the original Windows PE malware file and the corresponding byte prototypes are compared by computing the euclidean distance and the corresponding bytes in the byte prototype with significant modifications are selected and added as a new section at the end of the sections in the Windows PE malware file. After adding the new section, the structure of the adversarial PE file is preserved by performing the following modifications to the relevant fields: First, a new data structure is added to the Section Table at the end of the last entry, specifying attributes like Name, VirtualSize, SizeOfRawData, and PointerToRaw-

Data. Second, in the PE Header, the NumberOfSections field is updated to reflect the addition of the new section, and the SizeOfImage field is modified to compensate for the expanded file size.

- **GAMBD** [188] is a gradient-based method for producing adversarial Windows PE malware files. It considers MalConv [53] as the target model. This method consists of three stages: (a) perturbation initialization, (b) perturbation updation, and (c) conversion of embedding space perturbation to byte space. In the perturbation initialization stage, GAMBD uses two perturbation initialization approaches: without transfer and with transfer. In the without-transfer approach, multiple stochastic perturbations are produced. Initially, a stochastic perturbation is produced by considering the entire input range $[1, 255]$, ensuring broad exploration across all possible byte values. For the subsequent perturbations, the frequencies of byte values in the input sequence are counted, and the k least frequent bytes are selected as midpoints. A stochastic perturbation is then produced for each selected byte within the range $[seed - amount, seed + amount]$, where the seed is the least frequent byte value and the amount is the range of perturbation around the midpoint. The value of the amount is equal to the k least frequent bytes. The perturbation with the fewest zero gradients in embedding space is selected as the best initialization perturbation. In the transfer approach, the perturbation of the previously produced adversarial malware file is used in the current adversarial malware file. The stochastic perturbations with the input range $[1, 255]$ are used for the first adversarial malware file, where no previous perturbation exists.

  The perturbation updating stage consists of several steps. First, the initial perturbation is added to the original PE file to produce an adversarial file. This adversarial malware file is then fed into the classification model, and the confidence score is computed. The loss is computed between the confidence score and a target score (e.g., 0.5 or lower for goodware classification) rather than using the true label. Next, the perturbed byte input is passed through the embedding layer of the target model, and the embedding space perturbation is obtained. The embedding space perturbation is updated iteratively based on the gradients obtained using backpropagation. The iterative process is improved by using the Momentum Iterative Method (MIM) [189]. The MIM combines the gradients of the current and previous iterations, making the perturbation updating in the embedding space more effective in deceiving the decision model. The embedding space perturbation is updated during each iteration using larger step sizes and the gradient sign. In converting the embedding space to the byte space stage, each perturbed embedding vector is mapped back to byte space by finding the closest match in the fixed embedding matrix

using Mean Squared Error (MSE). The fixed embedding matrix contains embeddings for all possible byte values in the range $[1–255]$.

The iterative process of perturbation updating and embedding space-to-byte space conversion continues until the malicious score of the Windows PE adversarial malware drops below 0.5 or the maximum number of iterations is reached. The functionality of the Windows PE file is preserved by adding the perturbation at the end of the file. The produced Windows PE adversarial files are evaluated against the target model. The perturbation initialization using the transfer approach outperforms that without transfer. In addition, the adversarial files produced with the GAMBD method outperform baseline approaches in deceiving the detection model and reducing the time required for producing adversarial files.

- **GAMMA** [190] is a black-box attack method that produces the Windows PE adversarial malware files by injecting the benign content into the malware file. This benign content is extracted from a predefined set of k sections in goodware Windows PE files, ensuring the injected content is both valid and functionality-preserving. In the black-box setting, only querying is allowed for the target model to obtain the confidence score for the classification of a file. The target models used in [190] are: MalConv and Gradient Boosting Decision Tree (GBDT). The attack method begins by randomly initializing the population of N candidate manipulation vectors, $S' = s_1, s_2, ..., s_N$ where each manipulation vector $s = (s_1, s_2, ..., s_k)$, represents the fraction of the content to be extracted from the $k - th$ predefined sections. From the N population manipulation vectors S', using each manipulation vector s, the corresponding portions of benign content are extracted and injected into the malware files. Notably, each modified file corresponds to a manipulation vector s. These modified files are then queried one by one against the target model, and an objective function is computed. The objective function calculates the confidence score of the target model for the modified file and evaluates the size of the injected content. The goal is to optimize the population of the manipulation vectors to decrease the model confidence in classifying the malware file as malware, while keeping the size of the injected content as small as possible.

  The optimization of the manipulation vectors is performed iteratively. Each iteration includes three steps: selection, crossover, and mutation. In the selection step, the manipulation vectors obtained from the previous and the current iteration are combined, and the best N candidate manipulation vectors are selected based on the minimum objective function value. The selected manipulation vectors are the ones that are more influential for the modification. Notably, in the first iteration, the best candidate manipulation

vectors are selected from the initial population. In the crossover step, the selected manipulation vectors are paired randomly. The new manipulation vectors are generated by combining elements from each pair. For each pair, a random crossover point j is selected, and the resulting vector inherits the elements $s_1, ..., s_j$ from the one vector and the elements $s_{j+1}, ...s_k$ from the other vector. This crossover step creates a diversity in the extracted content. In the mutation step, a small probability of random modification is applied to the selected manipulation vectors obtained after the crossover step. This step introduces variability into the population, ensuring that the optimizer can escape local minima and explore new regions of the solution space. The mutation step allows the optimizer to explore the solution space further. After performing these steps, the newly generated manipulation vectors are used to inject corresponding portions of benign content into the malware files. These malware files are then queried against the target model. The iterative process continues until the maximum query budget is reached or the convergence is observed. At the end of the process, the best-performing manipulation vector s from the final population is selected and injected into the original malware file. The binary content is injected at the end of the file or by creating a new section in the file.

3.4.1 Benchmark datasets

There are several online systems that are publicly available to check and, in fewer cases, download Windows PE files. For example, VirusTotal [14] provides a free service to analyse and recognise the malicious behaviour of files submitted for a check. However, it provides for-pay services to obtain recorded files. On the other hand, VirusShare [15], and MalShare [16] are among the most popular, continuously updated repositories of live malicious code. Both repositories provide a free service that can be used to download Windows PE malware for research scope but no free service to obtain goodware files. By leveraging the online services described above and some private collections, several datasets have been created recently to conduct Windows PE malware detection studies. The most popular dataset used in the literature for Windows PE malware detection is EMBER [191] with around 1.1 million samples collected from VirusTotal between 2017 and 2018.

More recently, researchers have also started using the SoReL dataset [192] with 20 million samples collected between 2017 and 2019. The authors of SoReL do not describe the source of these samples in [192]; however, the repository documentation [17] reports that a large amount of Windows PE malware recorded in the SoReL repository also appears to be available via VirusTotal.

BODMAS [193] is another Windows PE dataset with around one hundred and thirty-four thousand samples collected between 2019 and 2020 from a security company's internal database.

Finally, DasMalwerk [18], is a Windows PE repository that contains a collection of Windows PE malware retrieved in late 2018. The total amount of samples collected in DasMalwerk is 104, and 37 of them are contained inside the EMBER dataset.

For EMBER, SoReL and BODMAS, pre-extracted features were obtained through the static analyser LIEF. However, the EMBER repository does not provide any binary files (for either malware or goodware examples). The SoReL repository provides the disarmed binaries of a few Windows PE examples only. The BODMAS repository provides the binaries of the Windows PE malware files, but it does not provide any binary file for goodware examples. Finally, the DasMalwerk repository, which contains only Windows PE malware files, provides the binaries of the collected files. Although EMBER is one of the oldest repositories, it is still used more than SoReL, BODMAS, and DasMalwerk in research studies. The PEMML [19] is an older Windows PE file repository that was made available by Practical Security Analytics LLC. It contains 201,549 Windows PE files collected before 2018 from VirusShare and MalShare. In this repository, the binary files are publicly available for all samples (both malware and goodware). The Mal-API-2019 [194] is a dynamic dataset that records API calls of around 7000 samples. The AVAST-CTU [195] is a recently released dataset that contains static and behavioural data collected from around 50 malicious examples. Behavioural features were observed in a sandbox over an extensive period of time.

A summary of the characteristics of the discussed datasets is reported in Table 3.1.

Table 3.1: A short description of characteristics of the EMBER, SoReL, BODMAS and DasMalWerk datasets

Dataset	**Malware**	**Goodware**	**Collection Time**	**Binary File Availability**
EMBER	400,000	400,000	2017–2018	No
SoReL	9,919,251	9,470,626	2017–2019	Malware
BODMAS	57,293	77,142	2019–2020	Malware
DasMalwerk	104	0	2018	Malware
PEMML	114,737	86,812	Before 2018	Goodware and malware
MAL-API-2019	7107	0	2019	No
AVAST-CTU	48,976	0	2017–2019	No

Chapter 4

Adversarial Windows PE analysis from offensive to defensive

This chapter is organized as follows: Section 4.1 outlines the methodology experimented for evaluating Full DOS, Extend, Shift, FGSM (padding+slack) and GAMMA Windows PE adversarial attacks, computing Euclidean distance in engineered feature-based space and raw byte-based space, Computing SHAP in engineered feature-based space and adversarial training strategy. Section 4.2 describes the preparation of the Windows PE dataset and evaluation metrics. Section 4.3 presents a discussion of the results, and Section 4.5 highlights the lessons learned and suggests future research directions in malware detection.

4.1 Motivation

Recent research studies have shown that AI-based cybersecurity systems, particularly developed for Windows PE malware detection are vulnerable to adversarial examples. The attackers exploit these vulnerabilities by producing malicious functinality preserving adversarial examples, resulting in misclassification by AI-based Windows PE malware detectors. The methods used to produce adversarial examples with their practical implications are evolving continuously, requiring through investigation to understand how attackers evades AI-based Windows PE malware detectors and how the robustness of these detectors can be improve against such attacks. The majority of the existing AI-based malware detection studies explores the accuracy of the malware detectors without paying attention to explore their vulnerability and robustness to adversarial examples. Therefore, in this chapter, we evaluate the evasion ability of the realistic Windows PE adversarial examples produced with five state-of-the-art attack method and assess the defensive strategy aiming to improve the robustness of AI-based Windows PE malware detectors against such adversarial attacks. This methdology contributes to assess how existing methods can be effective in the construction of a benchmark of adversarial Windows

PE to be used for studying robustness of AI-based malware detection methods, as well as the effectiveness of defensive strategies.

4.2 Adopted Methodology

This section describes the methodology used to analyse both the accuracy of the two pre-trained models, namely MalConv and LGBM, selected based on the literature for Windows PE malware detection, the evasion ability of Windows PE malware produced with Extend, Full DOS, Shift, FGSM (padding+slack), and GAMMA by attacking the MalConv, and the integrity of the models MalConv and LGBM against produced adversarial files, the Euclidean distance computed between the Windows PE malware files and Windows PE adversarial malware files, the SHAP analysis of original Windows PE malware and adversarial counterparts, and the performance of adversarial training done using LGBM and the adversarial samples produced in this study.
Both MalConv and LGBM were trained produced for Windows PE malware detection using the labelled PE files recorded in the EMBER dataset [191]. The MalConv model is a publicly available pre-trained model. The architecture of the MalConv model is described in detail in Chapter 3 (Section 2.3). The LGBM model is trained from the engineered features extracted with the LIEF static analysis of the binary PE files. LIEF is described in Chapter 3 (Section 3.2)

The evaluation methodology adopted to conduct the empirical study described in this study is divided into five steps, illustrated in the following.

Step 1: We evaluated the accuracy of the pre-trained models, MalConv and LGBM, on the prepared Windows PE dataset, namely WinPE dataset. The pre-trained MalConv model is evaluated on the raw bytes of the WinPE dataset, while the pre-trained LGBM model is evaluated on the LIEF engineered features extracted using the LIEF library. The results of the accuracy analysis are illustrated in (Section 4.4.1).

Step 2: In this step, we performed the following tasks: (1) We identifies the Windows PE malware files of the WinPE dataset that were correctly classified as malware by the MalConv model. (2) We produced the realistic Windows PE adversarial malware files from the identified malware files by using Extend, Full DOS, Shift, FGSM (padding+slack), and GAMMA. These attack methods were described in Chapter 3 (Section 3.4). (3) We created a new modified version of the WinPE dataset, by replacing Windows PE malware files with the corresponding adversarial files. The integrity of the MalConv and LGBM is evaluated on the modified dataset. The integrity analysis and the results are discussed in Section 4.3.2.

Step 3: We computed the Euclidean distance between the Windows PE adversarial malware files produced with the Full DOS, Extend, Shift, FGSM (padding+slack), and GAMMA attack methods and their corresponding Windows PE malware files. The Euclidean distance was computed in two spaces: raw byte-based space of MalConv and engineered feature-based space of LGBM. The distance analysis and the results are discussed in Section 4.3.3.

Step 4: We performed the SHAP analysis of the classification of the original Windows PE malware and the corresponding adversarial samples generated with Full DOS, Extend, Shift, FGSM (padding+slack), and GAMMA attack methods. The results of the SHAP analysis are described in Section 4.3.4.

Step 5: We evaluated the performance of the adversarial training as a defensive strategy in two configurations, named O and O + A, considered to analyze the performance of the LGBM model trained with adversarial training, named model $LGBM^{AT}$, and the baseline model trained with the original samples, name $LGBM^{O}$. In configuration O + A, $LGBM^{O}$ was trained on the original training set. is trained with the training folds of the WinPE dataset and tested on the testing set, namely augmented testing, which contains data of the testing fold from the WinPE dataset and the corresponding adversarial files. $LGBM^{AT}$ was trained on the training set augmented with the adversarial malware generated for the training malware. The evaluation conducted in the configuration O considered the original samples as the testing set. The evaluation conducted in the configuration $O + A$ considered the original samples of each testing set augmented with the adversarial malware generated for the considered testing malware. The adversarial strategy and results are described in Section 4.3.5.

4.2.1 Implementation Details

All experiments were conducted using Python 3.6 and Keras 2.6. The Python implementation of five Windows PE adversarial attacks methods is free available in Secml_malware library [196]. The Secml-malware implements state-of-the-art white-box and black-box attacks on Windows malware classifiers by leveraging a set of feasible manipulations that can be applied to Windows programs while preserving their functionality. The library consists of three main modules: attack, model, and util. The "attack" module contains several attacking strategies divided into white-box and black-box modules. The "model" module integrates two state-of-the-art classifiers: MalConv and the Gradient Boost Decision Tree (GBDT). The "Util" module contains support code to deal with the technicalities of the practical manipulations contained in the library. We trained the LGBM model on 400K goodware examples of engineered features and 400K malware examples of engineered features with default parame-

ters: 100 trees, 31 leaves per tree. In adversarial training, the evaluation was done using the 3-fold cross-validation (CV) of the PEwin dataset.

4.3 Dataset and Evaluation metrics

This section describes the data collection method of the Windows PE files and evaluation metrics, including accuracy, precision, recall, FScore, false negative rate, and false positive rate.

The evaluation of this empirical study was conducted by considering a collection of 27,035 Windows PE files: 13,494 goodware and 13,541 malware. Amonmg the malware, 55% belong to Trojan family, 26% to General malware, 15% to Viruses, 2.1% to Adware, 0.9% to Worms, and 0.7% to Ransomware. Goodware PE files were selected from the public PEMML [20] repository, which is an online repository that provides public binaries of goodware. Specifically, we selected 980 goodware files recorded in 2017 and 12514 goodware files recorded in 2018. Malware files were selected from VirusShare. In particular, we randomly selected 6459 malware files recorded in 2021, 83 in 2022, and 6999 in 2023. The following three queries are performed on VirusShare to retrieve the Windows PE malware files:

1. "filetype: "PE32 executable" extension:exe after 1 January 2021 ",
2. "filetype: "PE32 executable" extension:exe after 1 January 2022",
3. "filetype: "PE32 executable" extension:exe after 1 January 2023".

The queries were performed between March and June 2023. Malware files are downloaded in that period one by one.

We measured the performance of both MalConv and LGBM models by computing standard metrics commonly used to evaluate the predictive ability of binary classification models. Specifically, let us consider tm —the amount of Windows PE malware that is correctly predicted as malware; fm—the amount of Windows PE goodware that is wrongly predicted as malware;tg—the amount of Windows PE goodware that is correctly predicted as goodware; and fg—the amount of Windows PE malware that is wrongly predicted as goodware. We computed the following metrics:

- Overall accuracy (oa) that measures the proportion of correctly classified Windows PE files, regardless of the class, out of all the predicted files, i.e., $\mathsf{oa} = \frac{\mathsf{tm}+\mathsf{tg}}{\mathsf{tm}+\mathsf{tg}+\mathsf{fm}+\mathsf{fg}}$. This metric estimates the overall ability of a decision model to correctly classify a sample in its proper class, regardless of the class value.

- Precision (prec) that measures how many Windows PE malware files are correctly classified as malware, given all predictions of the malware class, i.e., $\mathsf{prec} = \frac{\mathsf{tm}}{\mathsf{tm}+\mathsf{fm}}$. This metric estimates how often the decision model is correct when predicting the target class "malware".

- Recall (recall) that measures how many Windows PE malware files are correctly classified as malware, given all occurrences of class malware, i.e., $\mathsf{recall} = \frac{\mathsf{tm}}{\mathsf{tm}+\mathsf{fg}}$. This metric estimates whether the decision model can find all samples of the target class "malware".

- Fscore (F) that measures the harmonic mean of precision and recall, i.e., $\mathsf{F} = 2\frac{\mathsf{prec}\cdot\mathsf{recall}}{\mathsf{prec}+\mathsf{recall}}$. As precision and recall are equally important, the Fscore is measured to estimate the trade-off between precision and recall. In particular, the higher the Fscore, the better the balance between precision and recall achieved by the evaluated approach.

- False negative rate (fnr) that measures the probability that Windows PE malware is wrongly classified as goodware, i.e., $\mathsf{fnr} = \frac{\mathsf{fg}}{\mathsf{tm}+\mathsf{fg}}$. The lower the false negative rate, the lower the number of malware files that are undetected.

- False positive rate (fpr) that measures the probability that Windows PE goodware is wrongly classified as malware, i.e., $\mathsf{fpr} = \frac{\mathsf{fm}}{\mathsf{tg}+\mathsf{fm}}$. The lower the false positive rate, the lower the number of goodware files that are wrongly detected as malware.

The higher the values of oa, prec, recall and F, the better the decision model. The lower the values of fnr and fpr, the better the decision model.

As an additional metric to measure the evasion ability of an attack method against a malware detection model, we considered evasion, which is the number of Windows PE malware files that are correctly classified with the model but whose adversarial counterparts, generated via the attack method, are wrongly classified as goodware according to the model. Formally, let tm be the number of true malware files recovered on the set of Windows PE malware files and tm^{A} be the number of true malware files recovered on the set of original Windows PE malware files where original malware files are replaced with adversaries whenever adversaries exist. Hence, $\mathsf{evasion} = \mathsf{tm} - \mathsf{tm}^{\mathsf{A}}$. The higher the evasion value, the higher the number of adversarial Windows PE malware files that evade the decision model, and consequently, the lower the integrity of the model versus adversarial attacks.

4.4 Results and discussion

4.4.1 Accuracy analysis of Pre-trained MalConv and lightGBM

Table 4.1 reports oa, prec, recall, F, fnr and fpr computed by measuring the accuracy performance of the pre-trained MalConv and LGBM models on WinPE dataset prepared to conduct this evaluation study.

Table 4.1: Accuracy performance analysis of the pre-trained MalConv and LGBM models. The accuracy metrics (oa—overall accuracy, prec—precision, recall—recall, F—Fscore, fnr— the false negative rate and fpr—the false positive rate) were measured on the WinPE dataset prepared for this evaluation study. The best results are underlined.

Model	**oa**	**prec**	**recall**	**F**	**fnr**	**fpr**
MalConv	0.8034	0.9766	0.6224	0.7603	0.3776	0.0150
LGBM	0.9294	0.9857	0.8717	0.9252	0.1283	0.0127

The results confirm that the conclusions drawn in [191] are still valid, even when considering the newest Windows PE malware files we collected for this evaluation study. In particular, despite the increased model size of MalConv (an input space composed of 1 million raw byte-based features against 2381 engineered features extracted through the static analysis of PE files), the feature engineering step still allowed LGBM to account for PE file characteristics that contribute to better disentangling malware from goodware.

4.4.2 Integrity Analysis of Pre-Trained MalConv and lightGBM

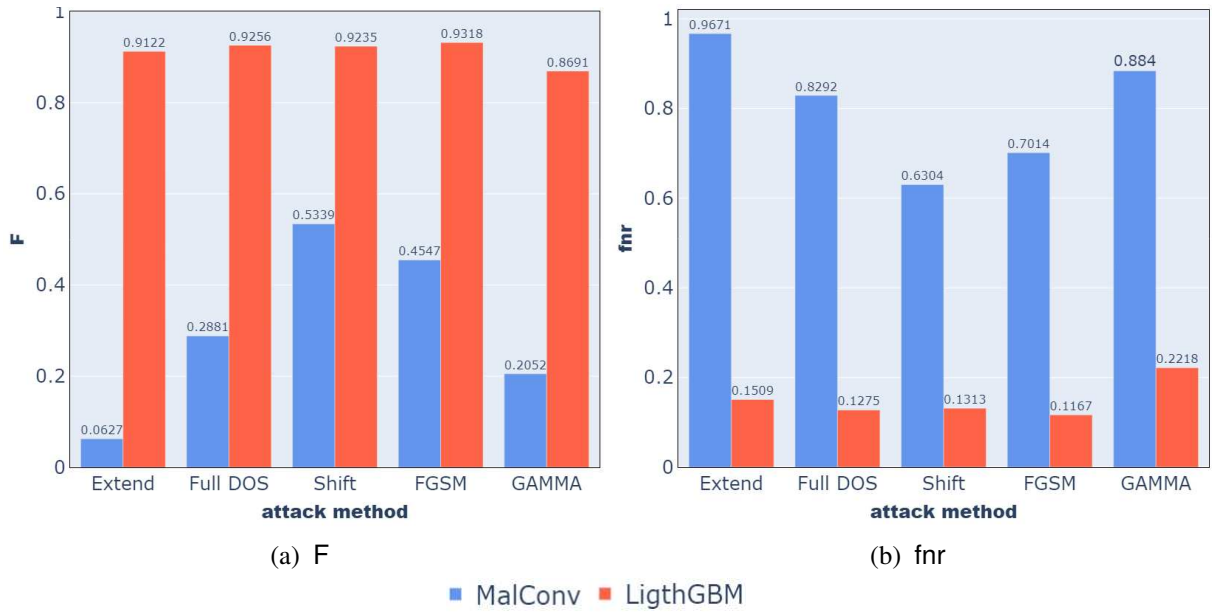
Table 4.2 reports the evasion metric measured for the pre-trained MalConv and LGBM models with respect to the adversarial Windows PE malware produced with the evaluated attack methods. As the attack methods were with MalConv as the target model, the evasion metric measured on the LGBM model allows us to verify the transferability of the evasion ability of the adversarial Windows PE malware files produced by fooling the pre-trained MalConv model versus the pre-trained LGBM model. In fact, the higher the evasion measured on the LGBM model, the higher the number of adversarial Windows PE malware files that were produced to fool the MalConv model but also evaded the LGBM model. In addition, we note that a negative value of evasion means that the Windows PE malware files produced with the considered attack method are all correctly detected in the "malware" class using the LGBM

model, even when the original malware counterparts from which the adversarial malware files are produced are wrongly classified as goodware according to the same model.

Table 4.2: The integrity performance (measured through the evasion metric) of the pre-trained MalConv and LGBM models and computed with respect to the following attack methods: Extend, Full DOS, Shift, FGSM padding + slack and GAMMA. The attack methods were used with the Windows PE malware of the study dataset to attack the pre-trained MalConv model.

Attack Method	**MalConv**	**LGBM**
Extend	7951	306
Full DOS	6115	− 10
Shift	3423	41
FGSM	4384	−157
GAMMA	6857	1266

So, based upon the considerations reported above, the results of the evasion metric achieved using the pre-trained MalConv model show that Extend is the attack method that is able to produce the higher number of Windows PE malware files that evade the MalConv model (i.e., they are wrongly classified as goodware), while their original counterparts (i.e., the malware files used for adversarial file generation) are correctly classified as malware according to the MalConv model. GAMMA and Full DOS are the runner-up attack methods for the pre-trained MalConv model. On the other hand, the analysis of the transferability of the realistic adversarial malware files, which fooled both the MalConv and LGBM models, shows that GAMMA achieved the highest transferability using the produced adversarial Windows PE malware files with Extend as the runner-up. In fact, GAMMA measured the highest value of evasion using the LGBM model, as it produces the highest number of adversarial files fooling the LGBM model (in addition to the MalConv model). Notably, Full DOS, which produced 6115 adversarial malware files to fool the MalConv model, achieved a negative evasion value with LGBM. This means that 10 malware samples of the WinPE dataset are wrongly classified according to the LGBM model as goodware, while their adversarial counterparts generated to fool the MalConv model with the Full DOS method are correctly detected as malware according to the LGBM model. A similar behaviour is observed for the Shift method.



(a) F (b) fnr

Figure 4.1: Evasion analysis of the pre-trained MalConv and LGBM models: F score (**a**) and fnr (**b**), measured on adversarial PE malware produced from the attack methods: Extend, Full DOS, Shift, FGSM padding + slack and GAMMA.

To complete this analysis, Figure 4.1a,b shows the F and fnr metrics measured for the pre-trained MalConv and LGBM models on the modified version of the WinPE dataset prepared for this study. Specifically, to this aim, for each attack method, we produce a modified version of the WinPE dataset by replacing each original PE malware for which the attack method is able to produce realistic adversarial malware that fools the MalConv with the produced adversarial malware. We perform this analysis considering F and fnr, as both metrics may have been affected by the evading ability of adversarial samples. In this analysis, the lower the value of F and the higher the value of fnr, the higher the evasion ability of the attack method with respect to the decision model, and consequently, the lower the integrity of the study decision model. Notably, both metrics confirm that GAMMA is the attack method for this evaluation study; it is able to evade LGBM with a higher number of realistic adversarial samples generated by fooling MalConv.

4.4.3 Distance Analysis of Adversarial Windows PE Malware

In this section, we analyse the distribution of the distance values that were computed between the original PE malware and the adversarial PE malware produced from each attack method considered in this evaluation study. This distance is measured through the Euclidean distance. The Euclidean distance values were computed after scaling the input dimensions of

all files between 0 and 1. Each distance value is divided by the number of dimensions in the input space in which it is calculated. This distance analysis is conducted to verify the existence of a possible relationship between the overall amount of changes introduced in the binary code of the adversarial PE files with each attack method and the corresponding amount of changes observed in the engineered features extracted for the same samples in the input space of LGBM. In addition, we intend to explore whether this relationship may be somehow related to the transferability of the attack method. The existence of such a relationship can be considered as a mechanism to foresee and explain the transferability of an attack method. Figure 4.2 shows the box plots of the Euclidean distance values computed between the original PE malware and the adversarial PE counterpart malware produced from each attack method considered in this evaluation study.

The results collected in the raw byte-based input space show that both Full DOS and FGSM (padding + slack) produced the adversarial malware files of this evaluation study closest to the binary files of the original counterpart malware. We recall that no adversarial malware that is produced from both of these methods by attacking MalConv is able to also evade LGBM. Even some adversarial PE files produced from both of these attack methods are correctly classified as malicious. In contrast, the original PE files are wrongly classified as goodware according to LGBM (see the negative evasion results reported in Table 4.2). Hence, this analysis suggests that a low raw byte distance between the adversarial malware and its original counterpart negatively affects the transferability of the adversarial malware (i.e., the possibility of evading a model different from the one for which it was generated). This conclusion is also supported by the fact that the highest raw byte distances are commonly observed in GAMMA, which is the attack method that produced the higher amount of adversarial malware able to evade both MalConv and LGBM. On the other hand, distances measured in the engineered feature-based input space show that GAMMA is the attack method that produces adversarial malware files that are generally furthest from the original counterpart malware also in the engineered feature-based space.

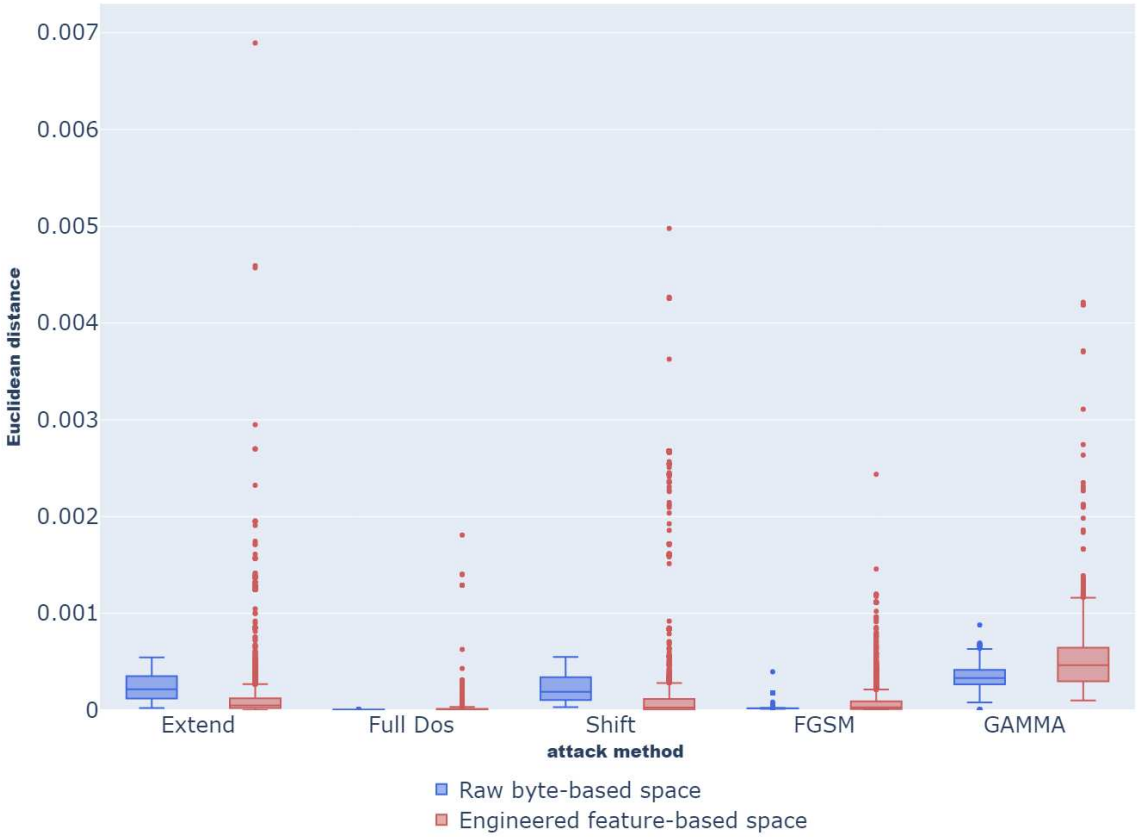


Figure 4.2: Box plots of Euclidean distance values computed in both the raw byte-based input space of MalConv (blue boxes) and the engineered feature-based input space of LGBM (red boxes) between original PE malware and its realistic adversarial malware counterparts produced with Extend, Full DOS, Shift, FGSM padding + slack and GAMMA.

To complete this study, we examined in-depth distances measured on the adversarial samples produced using GAMMA. Figure 4.3 shows the box plots of the Euclidean distances computed for the adversarial malware produced with GAMMA. To perform this analysis, we grouped GAMMA adversarial malware into two groups with respect to their ability to evade LGBM (in addition to MalConv). We note that distances measured in the engineered feature-based space show that the adversarial malware that evaded LGBM are generally furthest from the original counterpart malware files than the adversarial malware that non-evade LGBM. This supports the conclusion that an attack method that modifies a PE file in the raw byte space should be able to induce a remarkable change in the engineered features subsequently generated through the static analysis of the file. This allows the produced PE files to evade an accurate decision model like LGBM learned in a sophisticated feature-engineered space.

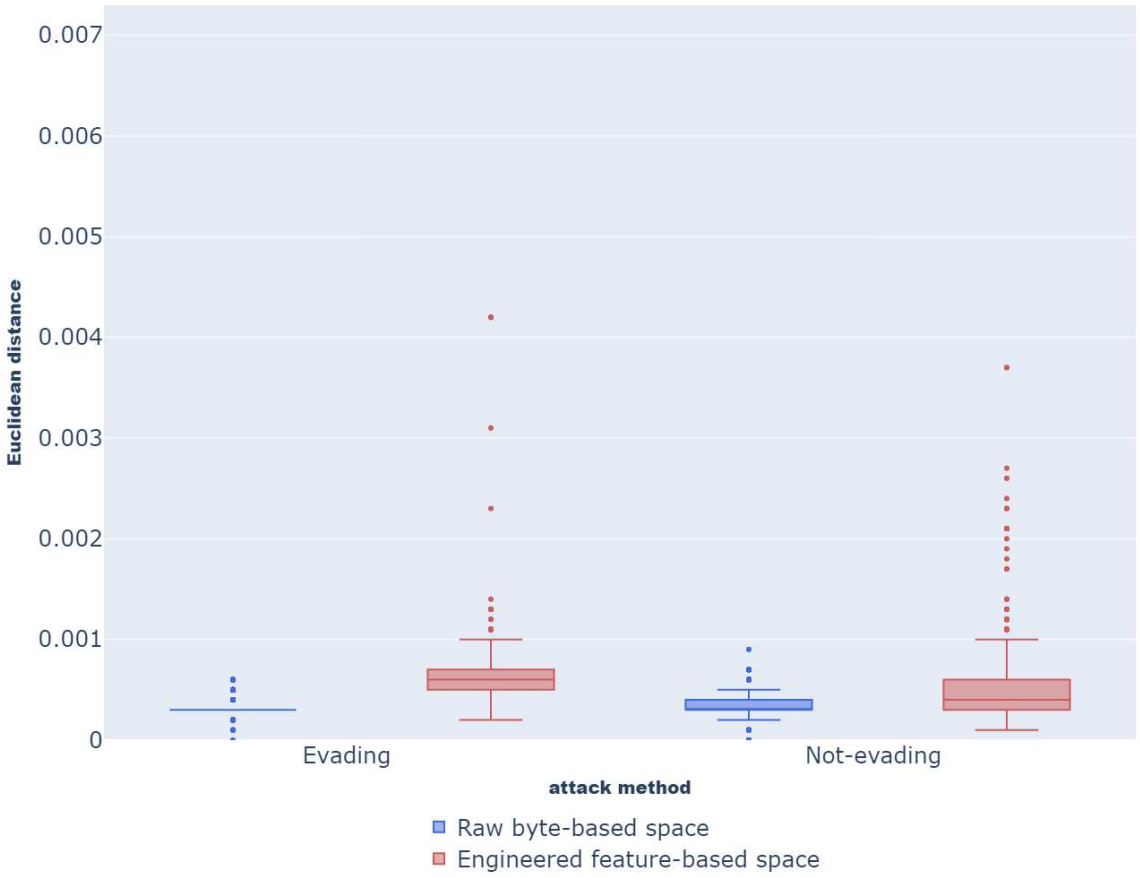


Figure 4.3: Box plots of Euclidean distance values (axis Y) computed in both the raw byte-based input space of MalConv (blue boxes) and the engineered feature-based input space of LGBM (red boxes) between original PE malware and their realistic adversarial malware counterparts produced with GAMMA and grouped with respect to their ability to evade or not-evade the pre-trained LGBM model (axis X).

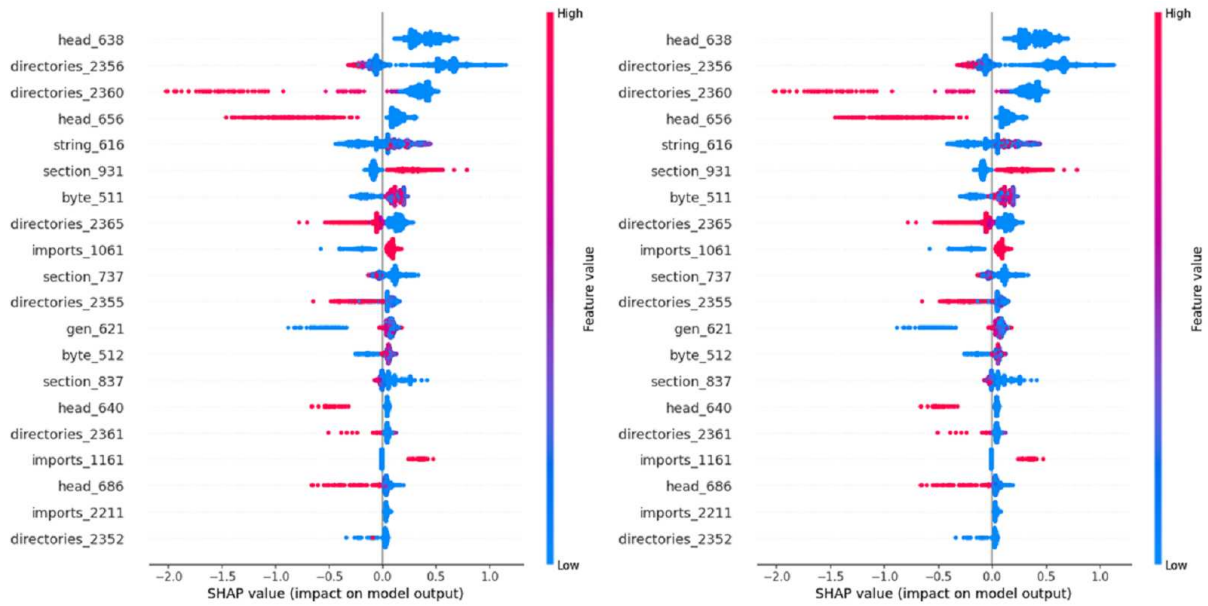
4.4.4 XAI-Based Analysis of the Effect of Adversarial Windows PE Malware on Engineered Features

In this section, we report the results of the analysis of the Shapley values computed for the decisions yielded with the pre-trained LGBM model for the adversarial malware files produced with the evaluated attack methods. We considered the adversarial Windows PE malware files that fool the LGBM model. Figures 4.4–4.8 show the distribution of the average Shapley values computed with respect to the "malware" class for the top 20 engineered features of the input feature space of LGBM. In particular, the left-side charts (Figures 4.4a–4.8a) show the Shapley values computed to explain the decisions concerning the original Windows PE malware, while the right-side charts (Figures 4.4b–4.8b) show the Shapley values computed to

explain the decisions concerning its adversarial malware counterparts. In these two charts, the colour scale expresses how the values observed for each feature are distributed in the explained samples; "new entry" means that a feature appeared in the top 20 ranking of the input features that mainly affected the decisions yielded via the LGBM model for the study's adversarial Windows PE malware files, but the same feature does not appear in the top 20 ranking of the features that affected the decisions yielded using the LGBM model for the original Windows PE malware files (used to produce the study's adversarial samples). Additionally, "up" (or "down") means that a feature gained (or lost) positions in the top 20 ranking of the input features that mainly affected the decisions yielded via the LGBM model for the study's adversarial malware files compared to the top 20 ranking of the features that affected the decisions yielded for the study's original malware files. Finally, "changed feature value" means that the distribution of the values observed for the feature changed from the plot produced for the study's original samples to the plot produced for the study's adversarial samples.

Notably, the plots do not show any change in the top 20 features ranked according to the Shapley values for Full DOS, which are shown in Figure 4.4. This means that LGBM works quite similarly to decide about both Windows PE malware files and their adversarial counterparts produced through Full DOS. This is unsurprising since no adversarial sample was produced via Full DOS that evades LGBM. Even 10 Windows PE malware files that were originally misclassified according to LGBM were correctly detected as malware when processed as they were modified via Full DOS (see Table 4.2). On the other hand, only a few changes appeared in the top 20 features ranked according to the Shapley values for FGSM padding + slack, which are shown in Figure 4.7. This is the other attack method of this evaluation study that produced adversarial PE malware capable of evading MalConv but not LGBM (see Table 4.2). On the contrary, numerous changes appeared in the top 20 Shapley-based feature rankings of Extend, shown in Figure 4.5, and Shift, shown in Figure 4.6. Both methods were able to produce some realistic adversarial malware capable of evading LGBM in addition to MalConv (see Table 4.2). The higher number of changes is seen in the explanation of decisions produced for adversarial malware files produced using Extend, which is the runner-up for GAMMA as the most effective attack method in this study. For GAMMA, the plots show that numerous changes appear in the top 20 Shapley-based feature rankings. Our feeling is that the high number of changes seen in the explanation of decisions yielded for the adversarial Windows PE malware files produced from GAMMA highlights which input dimensions of the LGBM model are mainly fooled by these adversarial malware files. This analysis shows that transferable attacks produced with GAMMA introduce some relevant changes in decisions yielded via LGBM with transferred adversarial samples.

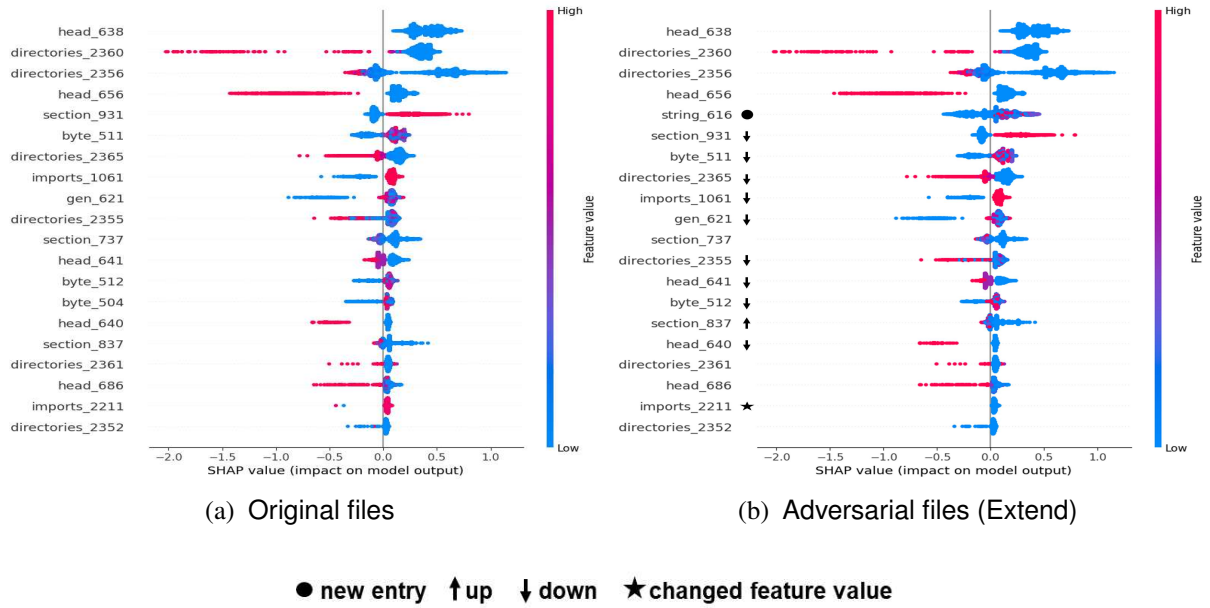
Hence, this additional exploratory analysis, which was performed to explain how the considered attack methods change the effect of features on decisions yielded via the LGBM model, supports the conclusion that there is a relationship between the effectiveness of the transferability of a Windows PE attack from MalConv to LGBM and the ability of this attack to introduce some relevant changes in decisions produced via LGBM with transferred adversarial samples.



(a) Original files

(b) Adversarial files(Full DOS)

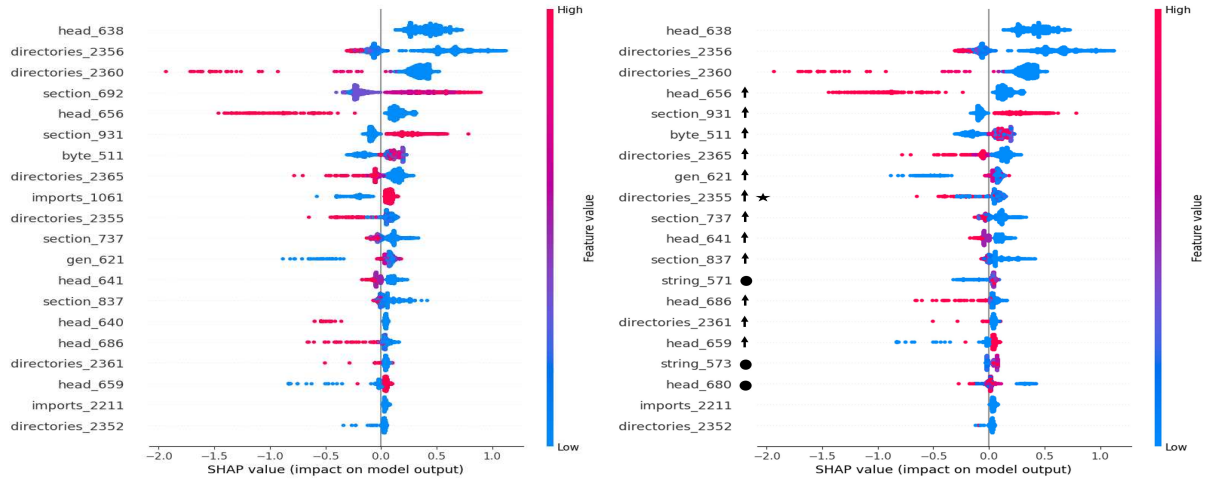
Figure 4.4: Shapley values measured for the top 20 input features (axis Y) of the LGBM input space and plotted with respect to the feature value (axis X) for the original 6115 PE malware files (**a**) and the adversarial counterparts (**b**) produced using Ful DOS. The two charts show that the same feature ranking is obtained in the two groups of files.



(a) Original files (b) Adversarial files (Extend)

● new entry ↑ up ↓ down ★ changed feature value

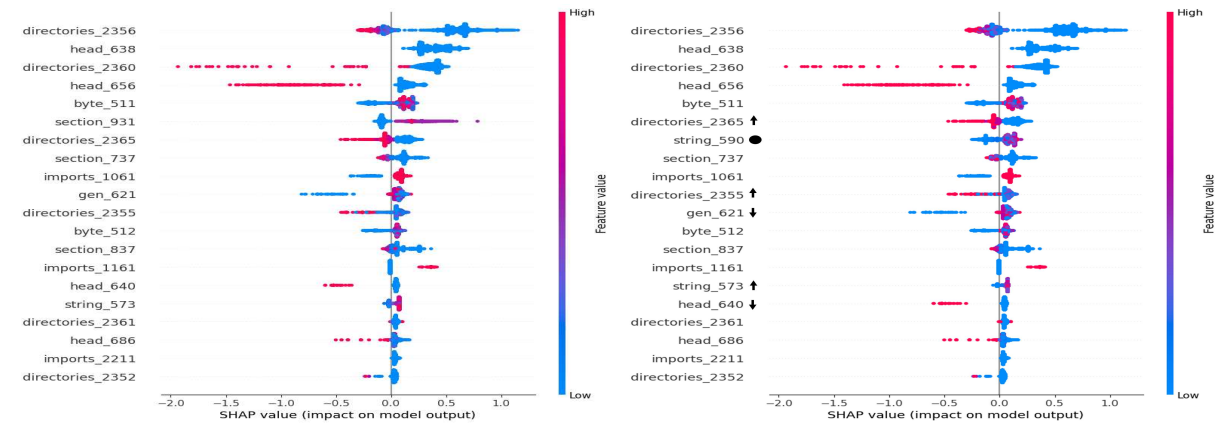
Figure 4.5: Global Shapley values measured for the top 20 input features (axis *Y*) of the LGBM input space and plotted with respect to the feature value (axis *X*) for the original 7951 PE malware files (**a**) and the adversarial counterparts (**b**) produced using Extend. Changes in the Shapley value-based feature ranking are marked in (**b**).



(a) Original files (b) Adversarial files (Shift)

● new entry ↑ up ↓ down ★ changed feature value

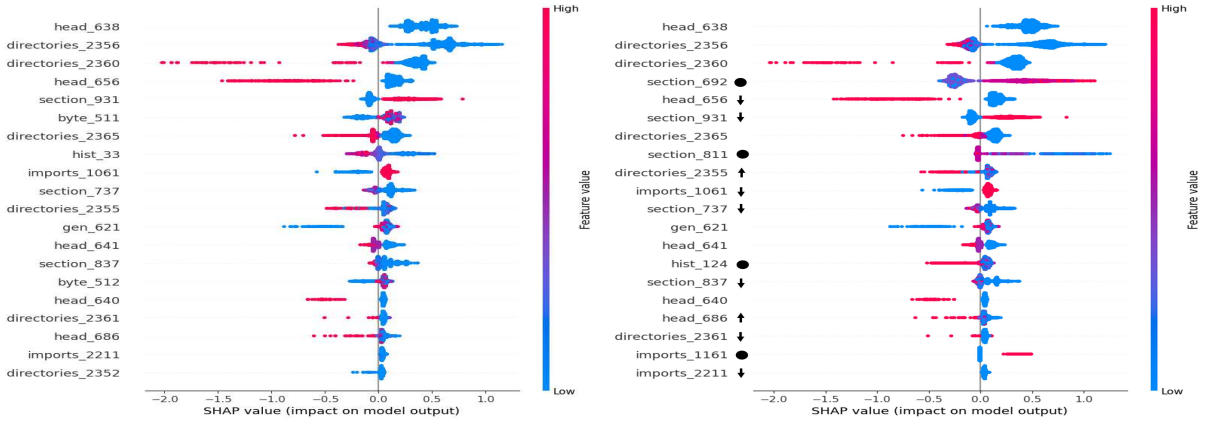
Figure 4.6: Shapley values measured for the top 20 input features (axis *Y*) of the LGBM input space and plotted with respect to the feature value (axis *X*) for the original 3423 PE malware files (**a**) and the adversarial counterparts (**b**) produced using Shift. Changes in the Shapley value-based feature ranking are marked in (**b**).



(a) Original files (b) Adversarial files (FGSM padding+slack)

● new entry ↑ up ↓ down

Figure 4.7: Shapley values measured for the top 20 input features (axis Y) of the LGBM input space and plotted with respect to the feature value (axis X) for the original 4384 PE malware files (**a**) and the adversarial counterparts (**b**) produced using FGSM padding+slack. Changes in the Shapley value-based feature ranking are marked in (**b**).



(a) Original files (b) Adversarial files (GAMMA)

● new entry ↑ up ↓ down

Figure 4.8: Shapley values measured for the top 20 input features (axis Y) of the LGBM input space and plotted with respect to the values measured for the features in the explained samples (axis X) for the original 6857 PE malware files (**a**) and the adversarial malware counterparts produced from GAMMA (**b**). Changes in the Shapley value-based feature ranking are marked in (**b**).

4.4.5 Accuracy Analysis of Adversarial Training with LGBM and Realistic Windows PE Attack Methods

In this section, we analyse the accuracy performance of the adversarial training strategy by exploring the effect of this strategy on the accuracy performance of the machine learning model trained with LGBM when this strategy was evaluated using a WinPE dataset (denoted as O), as well as when it evaluated using WinPE extended with the adversarial Windows PE malware (denoted as O + A). The study was conducted considering the adversarial Windows PE malware files generated with Extend, Full DOS, Shift, FGSM padding+slack and GAMMA. The accuracy metrics were computed using the decisions produced with the 3-fold CV methodology for all the Windows PE files (and eventually their adversarial counterparts) of the WinPE dataset.

We started the analysis of the results by examining the performance achieved in configuration O to evaluate the accuracy of predictions produced for the collection of the original Windows PE files. Table 4.3 reports the values collected for both the detailed accuracy metrics (tg—the amount of true goodware, fm—the amount of false malware, fg—the amount of false goodware and tm—the amount of true malware) and the summary accuracy metrics (oa—overall accuracy, F—Fscore, fnr— the false negative rate and fpr—false positive rate). These metrics were measured using the collection of predictions produced from $LGBM^O$ and $LGBM^{AT}$ in the evaluation configuration O. These results show that the use of the adversarial training strategy worsened the performance of LGBM in terms of tg, fm, oa, F and fpr regardless of the attack method used to perform the adversarial training strategy. On the other hand, the use of the adversarial training strategy allows $LGBM^{AT}$ to perform better than (or equally to) $LGBM^O$ in terms of fg, tm and fnr when adversarial files considered in the adversarial training stage of $LGBM^{AT}$ are produced with Extend, Full DOS and GAMMA. Specifically, when considering these attack methods, the injection of realistic adversarial samples in the training stage allows us to learn about an LGBM model that increases the number of malware decisions while also considering the original PE files only for evaluation. Notably, this had the consequence of increasing the amount of true malware detected. However, this also had the consequence of reducing the amount of true goodware detected by increasing the amount of false malware. Although we are aware that handling false malware is an inefficient use of time and resources, which may prevent a cybersecurity team from handling actual malware, we also note that false goodware may raise a serious cyber risk affecting the capacity to promptly mitigate cyber threats' consequences. From this point of view, any reduction in fnr may be considered a desirable behaviour, as long as it does not come at the expense of an excessive increase in fpr.

Table 4.3: Detailed accuracy metrics (tg—amount of true goodware, fm—amount of false malware, fg—amount of false goodware and tm—amount of true malware), and summary accuracy metrics (oa—overall accuracy, F—Fscore, fnr—the false negative rate and fpr—the false positive rate) of $LGBM^{O}$ and $LGBM^{AT}$ measured in the evaluation configuration O [1]. The evaluation was conducted through the 3-fold CV. Adversarial Windows PE malware files used to perform the adversarial training strategy in the training stages of $LGBM^{AT}$ were produced with Extend, Full DOS, Shift, FGSM padding + slack and GAMMA. Metrics for which $LGBM^{AT}$ outperformed (or performed equally to) $LGBM^{O}$ are underlined.

Model	**Test Set** (O)							
	tg	**fm**	**fg**	**tm**	**oa**	**F**	**fnr**	**fpr**
$LGBM^{O}$	13,432	62	60	13,481	0.9955	0.9955	0.0044	0.0046
$LGBM^{AT}$ (Extend)	13,423	71	54	13,487	0.9954	0.9954	0.0040	0.0053
$LGBM^{AT}$ (Full DOS)	13,422	72	55	13,486	0.9953	0.9953	0.0041	0.0053
$LGBM^{AT}$ (Shift)	13,423	71	60	13,481	0.9952	0.9952	0.0044	0.0053
$LGBM^{AT}$ (FGSM)	13,430	64	62	13,479	0.9953	0.9953	0.0046	0.0047
$LGBM^{AT}$ (GAMMA)	13,425	69	61	13,480	0.9952	0.9952	0.0045	0.0051

[1] O denotes the collection of decisions produced for the study's collection of original Windows PE files.

We continued the analysis by examining the performance achieved in the configuration O + A to evaluate the accuracy of the predictions produced for the collection of the original Windows PE files augmented with the realistic Windows PE malware produced with one of the attack methods considered in the evaluation stage. Table 4.4 reports the values collected for both the detailed accuracy metrics and the summary accuracy metrics measured using the collection of predictions produced from $LGBM^{O}$ and $LGBM^{AT}$ in the evaluation configuration O + A. In this evaluation setting, the LGBM model learned with the adversarial training strategy gained accuracy with respect to all the summary accuracy metrics of this study when the realistic adversarial Windows PE malware considered in both the training and evaluation stages

is produced with GAMMA. In fact, in examining the detailed accuracy metrics, we noted that the adversarial training strategy performed with GAMMA decreased the amount of true goodware (tg) by increasing the amount of false malware (fm) detected. However, it also increased the amount of true malware (tm) by decreasing the amount of false goodware (fg) detected. As the gain in the ability to correctly detect malware is greater than the reduction in the ability to correctly detect goodware, the use of the adversarial training strategy may be considered beneficial with GAMMA. In addition, the use of the adversarial training strategy gained accuracy with respect to fg, tm, oa, F and fnr when the realistic adversarial PE malware is produced with Extend. We recall that GAMMA is identified in this evaluation study as the most effective attack method, with Extend as the runner-up. So, this analysis highlights that adversarial training with GAMMA-produced samples can be a strategy to strengthen the LGBM model against this attack type. Notice that this analysis is performed on LGBM, which is, in the end, the most effective decision model for the Windows PE malware detection task in this study.

These considerations are also supported by the in-depth analysis of the accuracy performance of both $LGBM^O$ and $LGBM^{AT}$ measured with the subset of the adversarial Windows PE malware files that were used in the evaluation stage of the $A + O$ setting of each attack method. Table 4.5 reports the number of adversarial Windows PE malware files correctly classified in the "malware" class (true malware—tm) and the remaining number of adversarial Windows PE malware files wrongly classified in the "goodware" class (false goodware—fg) from both decision models. These results show that the adversarial training strategy is ineffective with attacks generated via Full DOS, Shift and FGSM padding + slack, while it allows us to reduce the number of misclassified adversarial Windows PE malware files (fg) when Extend and GAMMA are used as attack methods. In any case, the highest performance improvement was achieved using the adversarial training strategy with GAMMA. In fact, in this case, $LGBM^{AT}$ achieved the greatest reduction in the number of adversarial Windows PE malware files that were wrongly classified as goodware compared to $LGBM^O$ by passing from 75 to 2 false goodware decisions (fg). This result better supports our findings about the effectiveness of the adversarial training strategy with adversarial Windows PE malware files produced with GAMMA.

To complete the analysis of the performance of adversarial training with GAMMA, we examined explanations computed with SHAP for decisions yielded via $LGBM^O$ and $LGBM^{AT}$, respectively. We conducted this explanation analysis using the adversarial PE malware produced with GAMMA, which changed from being misclassified as goodware with $LGBM^O$ to being correctly classified as malware with $LGBM^{AT}$. Figure 4.9 shows the distribution of the top 20 Shapely values averaged on the decisions yielded via $LGBM^O$ and $LGBM^{AT}$ with the selected samples. The plots show that the use of adversarial training caused several changes in

Table 4.4: Detailed accuracy metrics (tg—amount of true goodware, fm—amount of false malware, fg—amount of false goodware and tm—amount of true malware), and summary accuracy metrics (oa—overall accuracy, F—Fscore, fnr—the false negative rate and fpr—the false positive rate) of $LGBM^O$ and $LGBM^{AT}$ measured in the evaluation configuration O + A [1]. The evaluation is conducted through the 3-fold CV. Adversarial Windows PE malware files are produced with
Extend, Full DOS, Shift, FGSM padding + slack and GAMMA. Metrics for which $LGBM^{AT}$ outperformed (or performed equally to) $LGBM^O$ are underlined.

Model	**Test Set (O + A)**							
	tg	**fm**	**fg**	**tm**	**oa**	**F**	**fnr**	**fpr**
$LGBM^O$	13,432	62	95	21,398	0.9955	0.9963	0.0044	0.0046
$LGBM^{AT}$ (Extend)	13,423	71	69	21,424	0.9960	0.9967	0.0032	0.0053
$LGBM^O$	13,432	62	84	19,571	0.9956	0.9963	0.0043	0.0046
$LGBM^{AT}$ (Full DOS)	13,422	72	77	19,578	0.9955	0.9962	0.0039	0.0053
$LGBM^O$	13,432	62	77	16,887	0.9954	0.9959	0.0045	0.0046
$LGBM^{AT}$ (Shift)	13,423	71	79	16,885	0.9951	0.9956	0.0047	0.0053
$LGBM^O$	13,432	62	81	17,844	0.9954	0.9960	0.0045	0.0046
$LGBM^{AT}$ (FGSM)	13,430	64	83	17,842	0.9953	0.9959	0.0046	0.0047
$LGBM^O$	13,432	62	135	20,265	0.9942	0.9952	0.0066	0.0046
$LGBM^{AT}$ (GAMMA)	13,425	69	63	20,337	0.9961	0.9968	0.0031	0.0051

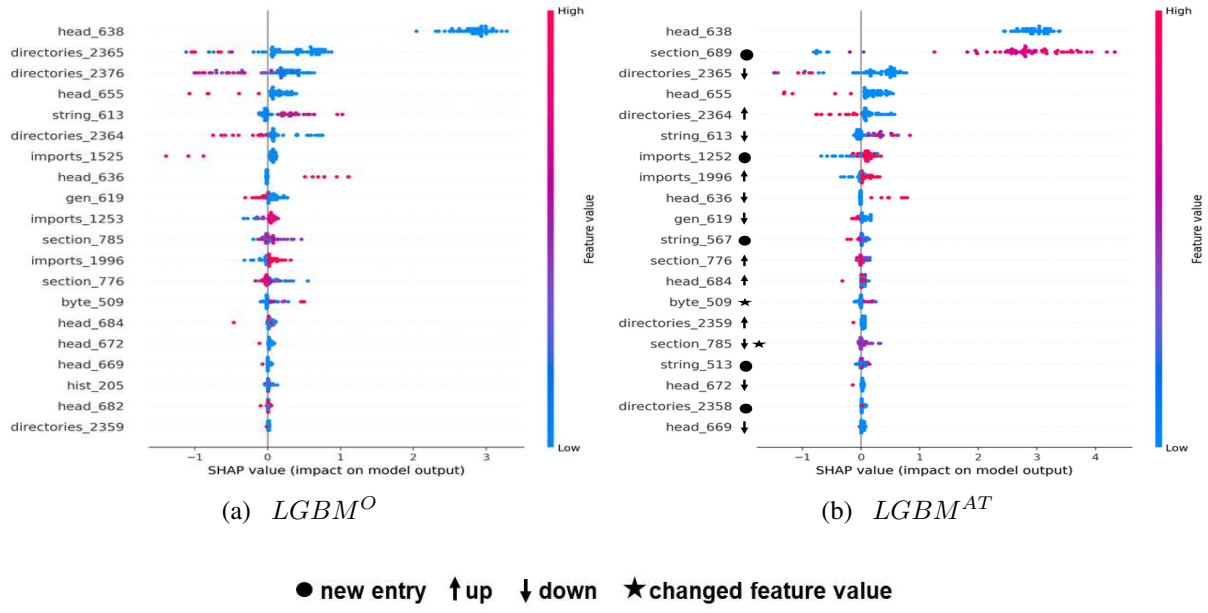
[1] O denotes the collection of decisions produced for the study's collection of original Windows PE files, while A denotes the collection of decisions produced for the collection of realistic adversarial malware produced from the study's original PE malware recorded in O.

Table 4.5: The number of "true malware" decisions (tm) and the number of "false goodware" decisions (fg) yielded via $LGBM^O$ and $LGBM^{AT}$ for the adversarial Windows PE malware files produced with the Extend, Full DOS, Shift, FGSM padding+slack and GAMMA attack methods and used in the evaluation stage of the $O + A$ setting considered in Table 4.4.

Attack Method	$LGBM^O$		$LGBM^{AT}$	
	tm	fg	tm	fg
Extend	7916	35	7936	15
Full DOS	6091	24	6093	22
Shift	3406	17	3404	19
FGSM	4363	21	4363	21
GAMMA	6782	75	6855	2

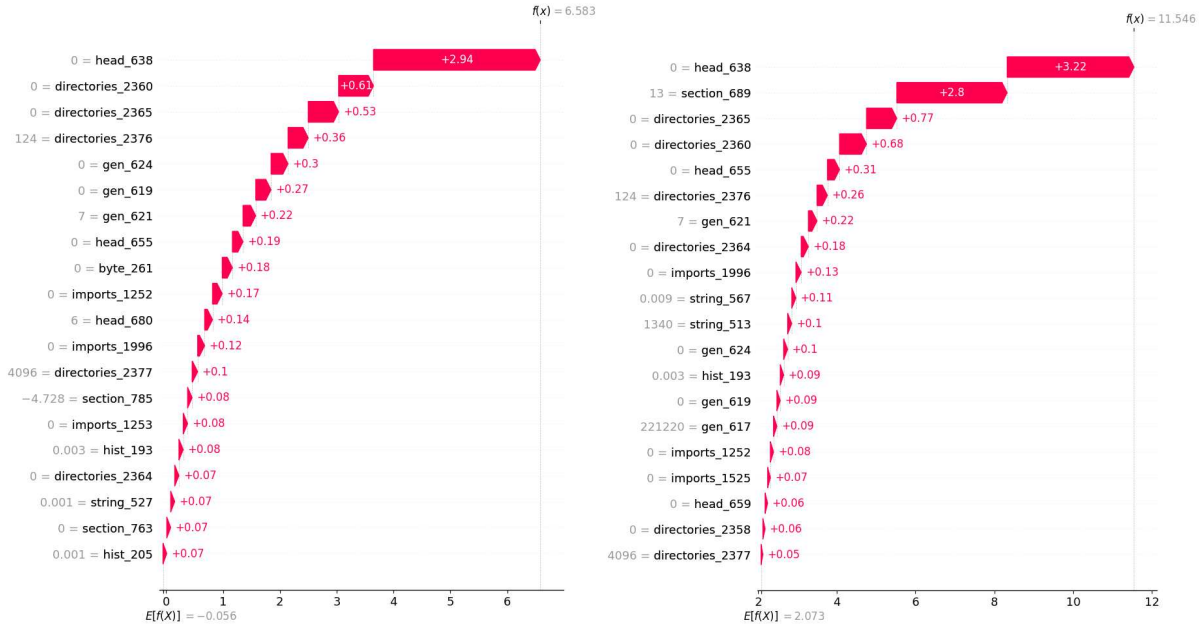
the top 20 Shapley-based feature ranking of $LGBM^O$ and $LGBM^{AT}$. The changes observed in the effect of features on decisions explain how adversarial training modified the decision-

making process of LGBM to allow the decision model to recognise some of the adversarial PE malware produced with GAMMA and originally misclassified with $LGBM^O$. For example, we note that both high values of "section_689" and high values of "import_1252" gained relevance from $LGBM^O$ to $LGBM^{AT}$, in order to recognise the malicious behaviour of the selected adversarial PE malware. Instead, we note a change in the relationship between the Shapley value distribution and the feature value distribution of "byte_509". In fact, the higher, positive Shapley values of "byte_509" that were measured in correspondence with high values of "byte_509" for $LGBM^O$ decisions were measured in correspondence with the low values of "byte_509" for $LGBM^{AT}$ decisions.



(a) $LGBM^O$ (b) $LGBM^{AT}$

Figure 4.9: Global Shapley values measured for the top 20 input features (axis *Y*) of $LGBM^O$ (**a**) and $LGBM^{AT}$ using GAMMA (**b**). The global Shapley values are plotted with respect to the feature value (axis *X*) for the 72 PE adversarial malware generated with GAMMA, which were wrongly classified in the "goodware" class according to $LGBM^O$ and correctly classified in the "malware" class according to $LGBM^{AT}$. Changes in the top ranking of the importance of input features for the model's decisions are marked in (**b**).



(a) $LGBM^{O}$ (b) $LGBM^{AT}$

feature	range	feature	range	feature	range
byte_261	[0, 0.00022]	directories_2358	[0, 48]	directories_2360	[0, 11704]
directories_2364	[0, 56]	directories_2365	[0, 37654]	directories_2376	[0, 22520]
directories_2377	[0, 37642]	gen_617	[4608, 4141139]	gen_619	[0, 1]
gen_621	[1, 636]	gen_624	[0, 1]	head_638	[0, 0]
head_655	[0, 1]	head_659	[−2, 0]	head_680	[2, 12]
hist_193	[0.00077, 0.01101]	hist_205	[0.00026, 0.01829]	imports_1252	[−2, 0]
imports_1253	[−2, 2]	imports_1525	[0, 1]	imports_1996	[−2, 0]
section_689	[3, 40]	section_763	[−213.6437, 5.14889]	section_785	[−16.38742, 4.461381]
string_513	[52, 18190]	string_527	[0, 0.00534]	string_567	[0.00564, 0.04212]

(c) Feature range

Figure 4.10: Local Shapley values (axis X) measured for the top 20 input features (axis y) of the decisions yielded via $LGBM^{\mathsf{O}}$ (**a**) and $LGBM^{\mathsf{AT}}$ (**b**), respectively, for a a Windows PE malware file generated with GAMMA. This is one of the files in the file set explained in Figure 4.9. It was wrongly classified as goodware according to $LGBM^{\mathsf{O}}$, while it was correctly classified as malicious according to $LGBM^{\mathsf{AT}}$. For each input feature, the value assumed by the feature in the study file is shown in the feature name reported on the axis Y in the FeatureName = FeatureValue format. The range of values that the ranked input features assume in the 72 PE malware files explained in Figure 4.9 is shown in (**c**). The Shapely value measured for each feature is reported in the corresponding feature bar. The higher the Shapley value, the more important the effect of the input feature on the decision using the LGBM model for the considered sample.

To complete the analysis of explanations reported in Figure 4.9, we selected a single Windows PE malware file from the 72 PE malware files explained in Figure 4.9. Figure 4.10 shows the top 20 local Shapley values of the input features that have a higher effect on the decisions yielded for the selected sample using $LGBM^{O}$ and $LGBM^{AT}$, respectively. This figure also shows values measured for the study sample with the top-ranked input features, as well as the range of values that each top-ranked feature assumes in the set of selected Windows PE malware files explained in Figure 4.9. The local explanation values produced for the two decisions yielded via both $LGBM^{O}$ and $LGBM^{AT}$ for this sample confirm that the correct classification achieved with $LGBM^{AT}$ can be partially explained by the fact that feature section_689 (which assumes a value equal to 13, which is in the middle part of the feature range [3, 40]), string_567 (which assumes the value equal to 0.009, which is in the lower part of the feature range [0.0056486, 0.0421279), string_513 (which assumes the value 1340, which is in the lower part of the feature range [52,18190]) and directories_2358 (which assumes the feature value 0, which is in the lower part of the feature range [0, 48]) gain importance in explaining the model's decision when the correct decision is yielded via the decision model $LGBM^{AT}$, which was trained using adversarial training.

4.5 Lessons learned and future research direction

Over the past decade, the proliferation and application of machine learning methods in several Windows PE malware detection studies have spurred the evolution of a new generation of Windows PE malware detection systems that have been integrated into several cyber defence platforms. On the other hand, recent advancements in adversarial learning have shown that, alongside traditional cyber attacks, machine learning methods have created an additional attack vector. In fact, machine learning models may be subject to cyber attacks called adversarial samples, like any other software system. Such attacks may have severe consequences on cyber defence systems, as adversaries could potentially bypass the machine learning-based Windows PE anti-malware systems. Therefore, AI models for Windows PE malware detection must be extensively evaluated in the context of, and possibly secured against, realistic adversarial machine learning attacks.

In the performed empirical study, we have illustrated the results of an extensive evaluation study on the performance of five state-of-the-art attack methods used to produce realistic adversarial Windows PE malware files. The study aimed to critically understand how an adversarial method could actually fool a machine learning model trained for Windows PE malware detection and explore how the adversarial training strategy could be used as a means of improving the security of the machine learning model against adversarial attacks. Notably, in this study,

we considered realistic Windows PE attacker methods that use the machine learning model feedback (i.e., gradient information in white-box attack methods and model decisions in black-box attack methods) to modify unused locations of Windows PE malware files by preserving the executable structure of Windows PE binary files, and guaranteeing that the functionality of the binary files is not compromised [190]. Although attacks are produced in the raw byte-based space, we analysed their transferability to the LIEF feature engineered-based space, where a more accurate decision model can be learned to address the Windows PE malware detection task. We performed a new exploratory study to explain how the attack can also fool this model. More importantly, we explored the effectiveness of adversarial training as a defensive strategy to make the decision model learned in the feature-engineered-based space more robust to the adversarial malware considered in this study. Our work is founded on the idea that AI methods for Windows PE malware detection must demonstrate their robustness to adversarial actions to transition from academia to become a crucial component of the cyber defence security line of public and private companies. Exploring the potential vulnerabilities of AI models is the first step to performing the adversarial defences for AI-based Windows PE malware detection against adversarial attacks.

In future work, we plan to extend this investigation to further Windows PE attack methods comprising poisoning methods [197], which are commonly studied to subvert learning with injected poisoned samples. In addition, we note that a work conducted under the umbrella of adversarial learning must be orthogonal to a work concerning common adversary tactics that are not specifically tailored to evading machine learning models. Modern malware uses common obfuscation techniques such as encryption, oligomorphic, polymorphic, metamorphic, stealth and packing methods to make the detection process more difficult [130]. In addition, armouring techniques can be used in malware to delay or stop the analysis of an executable through behavioural observation and/or reverse engineering [198]. Based on these considerations, our future investigations will include an exploration of the effect of adversarial training on files generated using re-coding/armouring adversary tactics. Finally, we intend to conduct a similar study in the field of Android malware detection [199], taking into account the fact that the Android operating system (OS) has been the leading platform for mobile devices since 2012. In this domain, [200] recently formalised the problem of realistic adversarial Android attacks, while attack methods against Android malware decision models have been studied in [182, 201].

Chapter 5

Tabular Synthetic Data Generation

The chapter is organized as follows: Section 5.1 introduces the request and the challenges associated with producing TSD. Section 5.2 illustrates an overview of GAN methods for generating TSD, with a focus on data quality differential privacy (DP) using GAN for TSD. Section 5.3 outlines the methodology experimented for producing TSD, with a focus on downstream classification tasks. Section 5.4 describes the real datasets, including benchmark cybersecurity datasets, considered for the experimentation on the considered methodology for TSD and the metrics employed to evaluate synthetic data in downstream classification tasks. Section 5.5 illustrates the experimental setup and presents a discussion of the results. Finally, Section 5.6 highlights the lessons learned and suggests future research directions in TSD. This work is done during the stage completed at LUTECH SpA Bari.

5.1 Introduction

We are in an era of rising data generation, leading to a paradigm shift from manual processes to AI-based applications. AI models are becoming essential tools across various domains. However, several challenges affect the AI model development, such as data privacy, high collection costs, lack of labels, inherent biases in data, and limited data availability. Synthetic data [202] is artificial data that can be produced with AI systems to handle these challenges. The AI-based model to produce synthetic data can learn the statistical distribution properties of the original data, such as variance, structure, and feature correlation [203]. Synthetic data may be used for various objectives, e.g., to anonymize sensitive information [204], to perform data augmentation [205, 206], or to balance data distributions [207].

Tabular data is structured in columns (features) and rows (examples). As tabular data are commonly available in various domains, TSD generation is a crucial task in several real sce-

narios. TSD generation is not only focused on producing synthetic data, but also requires evaluating the produced synthetic data considering fidelity, utility, and privacy dimensions.

In the following, we show some quality parameters to evaluate the TSD process, leveraging downstream classification and regression tasks relevant to the cybersecurity domain.

- Fidelity evaluates the degree of statistical resemblance of the TSD to the RTD.

- Utility refers to how well TSD performs in place of RTD in downstream AI tasks. The following process is performed to evaluate the utility: first, a classification model is trained on a real training set and evaluated on a real testing set; then, the same model is trained on a synthetic training set and evaluated on a real testing set; finally the accuracy performance metrics of the two models are compared.

- Privacy evaluates that the tabular synthetic training set is not an exact copy of the original training set, to guarantee that the risk of re-identifying the confidential information from the synthetic training set is minimal.

  In TSD, achieving an optimal balance among fidelity, utility, and privacy is challenging, as an improvement in a dimension often affects the remaining dimensions. In addition, tabular data comprises categorical and numerical features increasing the complexity of the data structure. A significant challenge of several problems requiring TSD for classification problems is the data imbalance, where some classes are more frequent than other classes. This imbalance impacts the fidelity and utility of the TSD. Addressing these challenges is an active area of research in designing TSD methods. In this thesis, we focused the attention on utility in TSD performed with the data quality-focused methodology evaluated several classification tasks comprising problems of network intrusion detection and malware classification. The motivation for focusing on the utility dimension is to ensure that AI-based models trained on synthetic training set can achieve comparable performance in downstream classification tasks to those trained on real training set, maintaining reliable accuracy in scenarios where access to real training data is limited or restricted. The Fidelity and privacy are critical concerns in TSD generation but are not explicitly assessed in this study. Fidelity evaluation requires statistical comparison between synthetic and real data distributions, and privacy evaluation involves measuring the risk of data leakage or re-identification. We plan to extend this work by adding fidelity and privacy evaluations in the future.

5.2 Literature overview

The notations used in this section are reported in the following.

- $\mathcal{D} = \{(\mathbf{x_i}, \mathbf{y_i})\}_{i=1}^{N}$: The training set, consisting of N training examples.
- $\boldsymbol{\mu_{\mathbf{x}}}$: mean of the feature vectors extracted from the real examples in the batch. A batch is a subset of data examples that are processed together during the training or testing stage of the decision model.
- $\boldsymbol{\mu_{\mathbf{G(u)}}}$: mean of the feature vectors extracted from the synthetic examples in the batch.
- $\boldsymbol{\sigma_{\mathbf{x}}}$: standard deviation of the feature vectors extracted from the real examples in the batch.
- $\boldsymbol{\sigma_{\mathbf{G(u)}}}$: standard deviation of the feature vectors extracted from the synthetic examples in the batch.

This section illustrates a literature review of TSD methods using GAN architectures. The motivation to focus on GAN-based methods for TSD is their distinct advantage over conventional TSD methods. Conventional TSD methods induce noise into the real data to ensure privacy, by creating a one-to-one mapping between real and synthetic data. This mapping increases the risk of re-identification of private information from synthetic data. In contrast, a GAN produces synthetic data from the noise sampled from a Gaussian distribution. This approach removes the one-to-one mapping between real and synthetic data and mitigates the risk of re-identification of private information from synthetic data. In addition, the adversarial sample generation realized with a GAN accommodates the synthetic data to preserve the utility of real data, making generated suitable for downstream tasks. The following literature review is categorized into data-quality-focused and differential privacy-focused methods for TSD. The selected data-quality-focused methods for TSD include: MedGAN [208], Table-GAN [209], IT-GAN [210], CTGAN [3], and CWGAN [211]. The selected differntial privacy-focused methods for TSD include: PATE-GAN [212], DP-GAN [213], RDP-CGAN [214], and CTAB-GAN+ [4].

5.2.1 Data quality-focused TSD methods

- **medGAN** [208] is a generative method for handling the TSD task for Electronic Health Record (EHR). The EHR data consists of categorical features, including binary features such as the presence of a disease and integer features such as count features such

as the number of visits. The core components of the medGAN architecture include: a pre-trained AE and a GAN. The training of the GAN relies on gradient-based optimization, which is difficult when the training set is categorical because gradients cannot flow through non-differentiable operations. This limitation is addressed by integrating a pre-trained AE that maps categorical data into a continuous latent space for smooth optimization. The pre-trained AE is trained separately on the $\mathcal{D}_{EHR}$ before the GAN training. The AE consists of an encoder Enc and a decoder Dec. Consider a single real example $\mathbf{x}$. The Enc compresses $\mathbf{x}$ into a continuous latent space representation, while the Dec reconstructs $\mathbf{x}$ from this continuous latent space. The AE uses the ReLU activation function for count features in both Enc and Dec. The Enc uses the Tanh activation function for binary features, and Dec uses the Sigmoid activation function. The reconstruction error is minimized using the MSE for count features and cross-entropy loss for binary features.

The GAN in medGAN consists of a generator $\mathbf{G}$ and a discriminator $\mathbf{D}_{gen}$. The generator $\mathbf{G}$ receives a random noise vector $\mathbf{u}$ sampled from a Gaussian distribution and produces a continuous latent representation **G(u)** that resembles the latent representation $Enc(\mathbf{x})$ produced by the Enc from a real example $\mathbf{x}$. The continuous latent representation **G(u)** is passed through the decoder Dec, which reconstructs the representation into tabular example Dec(**G(u)**)) before feeding it to $\mathbf{D}_{gen}$. $\mathbf{G}$ uses the ReLU activation function in all layers except for the output layer, which uses the Tanh activation function. $\mathbf{D}_{gen}$ receives real example $\mathbf{x}$ and reconstruct example Dec**(G(u))** as input. It produces a probability that represents the likelihood that the input example is real or fake. $\mathbf{D}_{gen}$ uses the ReLU activation in all layers except for the output layer, which uses the Sigmoid activation function. $\mathbf{D}_{gen}$ provides feedback to both Dec and $\mathbf{G}$: it fine-tunes Dec to reconstruct a more realistic example and guides $\mathbf{G}$ to produce representations that are closer to $Enc(\mathbf{x})$.

The mode collapse problem occurs during the GAN training when $\mathbf{G}$ starts producing similar examples for different random input noises, focusing only on fooling the $\mathbf{D}_{gen}$ rather than producing diverse examples. To mitigate the mode collapse issue, the medGAN method employs the minibatch averaging technique. In minibatch averaging, the average of all examples in a minibatch is concatenated with each example, for both real and fake examples, before feeding to the $\mathbf{D}_{gen}$. This ensures that $\mathbf{D}_{gen}$ access to the diversity within the minibatch while classifying individual examples. The mathematical formulation of the medGAN is defined in equation 5.1.

$$\min_{\mathbf{G}} \max_{\mathbf{D}_{gen}} \frac{1}{mini} \sum_{i=1}^{mini} \left[\log \mathbf{D}_{gen}([\mathbf{x}, \bar{\mathbf{x}}]) + \log(1 - \mathbf{D}_{gen}([Dec(\mathbf{G}(\mathbf{u})), \bar{\mathbf{x}}_{\mathbf{u}}])) \right] \tag{5.1}$$

where $mini$ is the minibatch size, $\bar{\mathbf{x}}$ is average of all real examples in the minibatch, $\bar{\mathbf{x}}_{\mathbf{u}}$ is average of all synthetic examples in the minibatch.

- **Table-GAN** [209] is a generative method that produces tabular synthetic data for binary classification and regression downstream tasks by handling categorical and numeric features. The table-GAN architecture comprises a generator $\mathbf{G}$, a discriminator $\mathbf{D}_{gen}$, and a classifier. $\mathbf{G}$ consists of multiple deconvolutional layers, which transform the input noise vector $\mathbf{u}$ into a 2-D matrix. $\mathbf{D}_{gen}$ is a CNN with multiple layers, with a sigmoid activation function in the output layer. The classifier has the same architecture as the discriminator $\mathbf{D}_{gen}$. The purpose of classifier integration in the table-GAN architecture is to ensure semantic integrity between labels and features in the original training set. During the training of a table-GAN model, three losses are employed: adversarial loss, information loss, and classification loss. The adversarial loss is used to train $\mathbf{G}$ to produce examples that can deceive $\mathbf{D}_{gen}$ while training $\mathbf{D}_{gen}$ to classify examples correctly.

  The information loss measures the statistical difference between the features of the real and synthetic examples. These features are extracted from the $\mathbf{D}_{gen}$ output layer just before the sigmoid activation function. To measure the information loss, the mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$ of the real and synthetic examples feature vectors are compared using the L2-norm. The mathematical formulation of the mean and standard deviation is defined in equation 5.2.

$$\mathcal{L}_{\text{mean}} = \|\boldsymbol{\mu}_{\mathbf{x}} - \boldsymbol{\mu}_{\mathbf{G(u)}}\|^2, \quad \mathcal{L}_{\text{sd}} = \|\boldsymbol{\sigma}_{\mathbf{x}} - \boldsymbol{\sigma}_{\mathbf{G(u)}}\|^2 \tag{5.2}$$

  where $\mathcal{L}_{\text{mean}}$ is the mean loss, and $\mathcal{L}_{\text{sd}}$ is the standard deviation loss. $\mathcal{L}_{\text{mean}} = 0$ and $\mathcal{L}_{\text{sd}} = 0$ indicate that the features of the real and synthetic examples are statistically similar from the perspective of $\mathbf{D}_{gen}$. The synthetic examples that are more similar to the real examples have higher risks of privacy leakage compared to the synthetic examples that are less similar. The hinge-loss is applied to the $\mathbf{G}$ to balance the quality and privacy of the produced examples, as defined in equation 5.3.

$$\mathcal{L}_{\mathbf{G}_{\text{info}}} = \max(0, \mathcal{L}_{\text{mean}} - \boldsymbol{\delta}_{\text{mean}}) + \max(0, \mathcal{L}_{\text{sd}} - \boldsymbol{\delta}_{\text{sd}}) \tag{5.3}$$

where $\boldsymbol{\delta}_{\text{mean}}$, and $\boldsymbol{\delta}_{\text{sd}}$ are predefined thresholds that define the acceptable level of similarity between the features of real and synthetic examples in terms of mean loss and standard deviation loss. The small value of $\boldsymbol{\delta}_{\text{mean}}$ and $\boldsymbol{\delta}_{\text{sd}}$ indicate high data fidelity but low privacy, while the large value indicates low fidelity but high privacy. The classification loss measures the discrepancy between the labels of the synthetic examples and the labels predicted by the classifier for those synthetic examples.

- **Conditional Tabular GAN (CTGAN)** [3] is a conditional generative method that produces tabular synthetic data for binary classification, multiclass classification, and regression downstream tasks by addressing the class imbalance problem and handling categorical and numeric features. In the preprocessing stage, the categorical features are transformed into a one-hot vector, while the numeric features are transformed into a one-hot vector using a mode-specific normalization technique. The mode-specific normalization technique works in three steps: (1) it estimates the number of modes with their parameters mean $\boldsymbol{\mu_m}$, standard deviation $\boldsymbol{\sigma_m}$, and $weig_m$ of each numeric feature O_i using variational Gaussian mixture model (VGM) [215] and fit the Gaussian mixture. (2) It computes the probability density ρ_k for each value $o_{i,j}$ in O_i to determine its likelihood of belonging to the k^{th} mode, as defined in equation 5.4.

$$\rho_k = weig_{m,k} N(o_{i,j}; \boldsymbol{\mu_{m,k}}, \boldsymbol{\sigma_{m,k}}) \tag{5.4}$$

where $N(.)$ is the probability density function. (3) It normalizes the numeric values as defined in equation 5.5.

$$\hat{o}_{i,j} = \frac{o_{i,j} - \boldsymbol{\mu_{m,k}}}{4\boldsymbol{\sigma_{m,k}}} \tag{5.5}$$

where 4 is a scaling factor. The one-hot vector representation of each numeric value is a combination of its normalized form $\hat{o}_{i,j}$ and the one-hot encoded vector of modes, where the value 1 indicates the k^{th} position the selected mode. The CTGAN integrates the conditional vector, **cond**, to apply constraints on specific categorical features for producing the TSD. The **cond** vector is constructed by transforming the categorical feature values into one-hot vectors, and the corresponding mask vectors are created to identify the active category. For example, let us consider a training set that contains two categorical features, $Cat_1 = \{1, 2, 3\}$ and $Cat_2 = \{1, 2\}$, and the constraint is $Cat_2 = 1$, the mask vectors are $m1 = \{0, 0, 0\}$ and $m2 = \{1, 0\}$, respectively. Then, **cond** $= \{0, 0, 0, 1, 0\}$ is the constructing vector. This **cond** vector ensures that $\mathbf{G}$ produces the data that satisfies

the specified conditions. In addition, during the training stage, in case of deviation from the conditions, $\mathbf{G}$ is penalized by the cross-entropy loss. In addition to **cond**, CTGAN employs a training-by-sampling strategy to address the class imbalance problem. This strategy ensures the equal exploration of all categories of categorical features during the training stage. The sampling step is performed as follows. First, for each categorical feature Cat_i, the corresponding zero-filled mask vectors $m_i = [m_i^{(k)}]_{k=1...|Cat_i|}$ are created, where each component of the mask vector corresponds to a category in the feature. Then, a categorical feature Cat_{i^*} is randomly selected from all categorical features with equal probability. For the selected feature, a probability mass function (PMF) is constructed across its range of values, where the probability mass for each value is the logarithm of its frequency in the feature. A value k^* is sampled from this PMF, and the corresponding component in the mask vector for Cat_{i^*} is set to 1, i-e., $m_{i^*}^{(k^*)} = 1$. Finally, the $cond$ is calculated as the concatenation of all mask vectors, $cond = m_1 \oplus m_2 \oplus \cdots \oplus m_n$.

In CTGAN architecture, $\mathbf{G}$ and $\mathbf{D}_{gen}$ are fully connected neural networks. $\mathbf{G}$ consists of two hidden layers with batch normalization and ReLU activation, followed by a synthetic row representation produced using mixed activation functions. The Tanh activation function is used for producing scaler values, while the Gumbel-Softmax activation function is used for producing mode-indicator and categorical values. The Gumbel-Softmax activation function is a differentiable approximation of the argmax operation. It introduces Gumbel noise into the softmax function to allow categorical sampling during backpropagation. $\mathbf{D}_{gen}$ consists of two hidden layers with Leaky ReLU activation and dropout on each layer. To mitigate the mode collapse issue, CTGAN uses the PacGAN [216] framework, where 10 samples in each pac are passed to $\mathbf{D}_{gen}$ for classification. The CTGAN is trained using Adam optimizer with a learning rate 2×10^{-4} and with the WGAN-GP loss.

- **Invertible Tabular GAN (IT-GAN)** [210] is a generative method that produces tabular synthetic data for binary classification, multiclass classification, and regression downstream tasks by handling numeric and categorical features. In the preprocessing stage, the features with numeric values are transformed into a one-hot vector using a mode-specific normalization [3], while the features with categorical values are transformed into a one-hot vector directly. The IT-GAN architecture comprises an AE and a GAN. Both Enc and Dec apply the ReLU activation function at each layer. $\mathbf{G}$ is a Neural Ordinary Differential Equations (NODE)-based [217] invertible neural network. It transforms the noise vector $\mathbf{u}$, sampled from a standard Gaussian distribution, into a synthetic latent representation by applying a series of NODE-based layers. $\mathbf{G}$ aims to produce a latent repre-

sentation that closely resembles the representation produced by the Enc. $\mathbf{D}_{gen}$ is a neural network consisting of multiple fully connected layers, with Leaky ReLU activation applied in the intermediate layers and dropout for regularization. It receives the latent representation produced by the Enc from the real training set and the synthetic latent representation produced by $\mathbf{G}$ to perform the learning task. $\mathbf{G}$ and $\mathbf{D}_{gen}$ are trained using a Wasserstein Generative Adversarial Network with Gradient Penalty (WGAN-GP) [218] loss, which ensures stable training and data fidelity. The mathematical formulation of the WGAN-GP is defined in equation 5.6.

$$\mathcal{L}_{\text{WGAN-GP}} = \mathbf{D}_{\text{gen}}(\mathbf{h}_{\text{syn}}) - \mathbf{D}_{\text{gen}}(\mathbf{h}_{\text{real}}) + \boldsymbol{\lambda}\left(\|\nabla \mathbf{D}_{\text{gen}}(\mathbf{h}_{\text{interp}})\|_2 - 1\right)^2 \tag{5.6}$$

where $\mathbf{h}_{\text{real}}$ is the latent representation of the real examples and $\mathbf{h}_{\text{syn}}$ is the latent representation produced by $\mathbf{G}$. $\boldsymbol{\lambda}$ is a gradient penality coefficient. $\mathbf{h}_{\text{interp}}$ is the interpolated representation between real and synthetic samples.

Additionally, $\mathbf{G}$ is periodically trained with a log-density regularizer to avoid mode collapse. The Enc is trained with WGAN-GP loss and AE loss, applied independently, while Dec is trained using AE loss. The AE loss is defined in equation 5.7.

$$\mathcal{L}_{\text{AE}} = \mathcal{L}_{\text{Reconstruct}} + \frac{1}{2}\|\mathbf{h}_{\text{real}}\|^2 + \|\mathbf{h}_{\text{syn}} - \hat{\mathbf{h}}_{\text{syn}}\|^2 \tag{5.7}$$

where $\mathcal{L}_{\text{Reconstruct}}$ is a loss of reconstructing the real input from the latent representation produced by the Enc. $\hat{\mathbf{h}}_{\text{syn}}$ is the latent representation produced by Enc.

- **Conditional Wasserstein GAN (CWGAN)** [211] addresses the class imbalance problem in synthetic data generation for classification problems by producing minority class examples. This method uses conditional GAN for producing minority class examples. The CWGAN architecture comprises a GAN and an auxiliary classifier. The GAN consists of $\mathbf{G}$ and $\mathbf{D}_{gen}$. $\mathbf{G}$ takes a noise vector $\mathbf{u}$ and class label y as an input and processes it through parallel hidden layers $H_1 \ldots H_n$ and cross layers $Cross_1 \ldots Cross_n$. The output results of the final hidden layer and the final cross layer are concatenated and passed to the output layers. An individual output layer is for each categorical feature, producing an output vector with a length equal to the number of categories in the respective feature. A single output layer is for all numerical features, producing an output vector with a length equal to the number of numerical features. Each categorical feature output is embedded using an embedding layer E and processed through a hidden layer, which reduces the embedding dimensions of all feature outputs to a single 16-D dense vector.

This dense vector is concatenated with the input to the numerical output layer, allowing $\mathbf{G}$ to produce numerical outputs that are explicitly conditioned on the categorical outputs.

$\mathbf{D}_{gen}$ receives input data with labels y. The one-hot encoded vectors of the categorical features are passed through their respective embedding layers $E_1 \dots E_n$, which reduce the dimensionality of these features. In numerical features, the Gaussian noise with a standard deviation of 0.01, sampled from the normal distribution, is added. The processed categorical and numerical features, combined with the labels, are passed in parallel through the hidden layers $H_1 \dots H_n$ and cross layers $Cross_1 \dots Cross_n$. the layer normalization is applied after each hidden layer to stabilize training and improve convergence. The outputs from the hidden layers and cross-layers are concatenated and passed to the final layer, which produces a single scalar output without any activation function. The auxiliary classifier has the same architecture as $\mathbf{D}_{gen}$ but with three modifications: No Gaussian noise is added in numerical features; the input is unlabeled; and the sigmoid activation function is applied in the output layer. To mitigate the model collapse issue, CWGAN uses the WGAN-GP loss. The auxiliary classifier evaluates the alignment of synthetic data with the conditioned class by computing the binary cross-entropy loss for real and synthetic examples. This loss is fed to the generator, encouraging it to produce class-consistent synthetic data.

5.2.2 Differential privacy-focused TSD methods

Differential privacy increases the difficulty of an attacker to re-identify the individual data example from the data produced by a differential private algorithm $\mathcal{M}$. This privacy guarantees that including or excluding a single data point from input data does not significantly alter $\mathcal{M}$ output probability. An algorithm $\mathcal{M}$ is differential private if, for any subset of possible outputs $\mathcal{S}$ from the output space $\mathcal{O}$, and for any two neighbouring datasets $\mathcal{D}_1$ and $\mathcal{D}_2$ differing in exactly one example, the probability of producing an output in the subset $\mathcal{S}$ does not differ significantly between these neighbouring datasets. The mathematical formula of this condition is defined in equation 5.8.

$$P_r(\mathcal{M}(\mathcal{D}_1) \in \mathcal{S}) \leq e^{\boldsymbol{\epsilon}_p} \cdot P_r(\mathcal{M}(\mathcal{D}_2) \in \mathcal{S}) + \zeta \tag{5.8}$$

where P_r is the probability that the produced output belongs to $\mathcal{S}$. $\mathcal{O}$ is output space. $\boldsymbol{\epsilon}_p$ is the privacy budget controlling the level of privacy guarantee. ζ is a small quantity accounting for the probability of failure of privacy guarantee.

- **Private Aggregation of Teacher Ensembles-Generative Adversarial Network (PATE-GAN)** [212] describes a method produces synthetic data ensuring differential privacy. The PATE-GAN architecture comprises a generator $\mathbf{G}$ and a discriminator $\mathbf{D}_{dp}$. $\mathbf{G}$ takes noise vector $\mathbf{u}$ and produces the synthetic data. $\mathbf{D}_{dp}$ consists of k teacher-discriminators, $T_1, \ldots, T_k$ and a student discriminator S_d. The real dataset is initially partitioned into k disjoint subsets $d_1, \ldots d_k$. Each teacher discriminator T_i can handle the corresponding subset d_i of the real dataset. During the training stage, each teacher discriminator is trained to distinguish between real and synthetic examples, using its assigned subset of the data and the examples produced by the $\mathbf{G}$. To classify an example, each teacher-discriminator assigns a vote, and the noise is added after aggregating these votes to produce a differentially private label for S_d. This noisy voting mechanism ensures the differential privacy by limiting the influence of any single teacher discriminator decision. In this step, the teacher-discriminators trained themselves only, not $\mathbf{G}$. S_d is trained on $\mathbf{G}$-produced data, not on real data directly, with the noisy label as ground truth. The S_d training is defined in equation 5.9.

$$L_{S_d}(S_d) = \sum_{j=1}^{bs} \left[l_j \log(S_d(\mathbf{u}_j)) + (1 - l_j) \log(1 - S_d(\mathbf{u}_j)) \right] \tag{5.9}$$

  where l_j is the noisy label and bs is batch size. $\mathbf{G}$ is optimized with respect to the S_d, allowing it to learn from a privacy-preserving representation of the real data.

- **Differentially Private Generative Adversarial Network (DP-GAN)** [213] defines a method that produces synthetic data ensuring differential privacy. DP-GAN is built on the Wasserstein GAN architecture, that employs the Wasserstein distance as the loss function between real and synthetic data distributions. The core contribution of the DP-GAN method is the noise addition into the gradients associated with Wasserstein distance during $\mathbf{D}_{gen}$ training, which ensures differential privacy. The DP-GAN architecture comprise of G and $\mathbf{D}_{gen}$. $\mathbf{G}$ takes noise vector $\mathbf{u}$ as input and produces the synthetic data. $\mathbf{D}_{gen}$ receives the real examples and the examples produced by $\mathbf{G}$ and computes the Wasserstein distance by assigning higher scores to real examples and lower scores to synthetic examples. It then computes the gradients of the Wasserstein loss with respect to its parameters for both real and synthetic examples, as defined in equation 5.10.

$$\nabla_w = \frac{1}{bs} \sum_{i=1}^{bs} \left[\nabla_w \mathbf{D}_{gen}(\mathbf{x}_i) - \nabla_w \mathbf{D}_{gen}(\mathbf{G}(\mathbf{u}_i) \right] \tag{5.10}$$

where bs is batch size. $\mathbf{D}_{gen}$ clips the gradients norm to a fixed threshold, preventing them from becoming excessively large, which could destabilize training and add Gaussian noise to these clipped gradients. $\mathbf{D}_{gen}$ parameters are then updated with these perturbed gradients using the root mean square propagation (RMSProp) optimizing algorithm.

- **Rényi Differential Privacy and Convolutional Generative Adversarial Networks (RDP-CGAN)** [214] describes a method that produces synthetic data considering the Renyi Differential Privacy (RDP) [219] concept. The RDP is a general form of differential privacy, which measures the difference between two probability distributions using the Renyi divergence RD. The difference is controlled by a parameter $r\alpha$. The mathematical formulation of the RD between two probability distributions $Dist_1$ and $Dist_2$ is defined in equation 5.11.

$$RD_{r\alpha}(Dist1 \| Dist2) = \frac{1}{r\alpha - 1} \log \mathbb{E}_{Dist2(\mathbf{X})} \left[\left(\frac{Dist1(\mathbf{x})}{Dist2(\mathbf{x})} \right)^{r\alpha} \right] \tag{5.11}$$

$\mathcal{M}$ satisfies the RDP conditions, if for any pair of neighbouring datasets $\mathcal{D}_1$ and $\mathcal{D}_2$, the Renyi divergence RD of order $r\alpha$, computed between the outputs of $\mathcal{M}(\mathcal{D}_1)$ and $\mathcal{M}(\mathcal{D}_2)$, is bounded by $\boldsymbol{\epsilon}_{rd}$ as defined in equation 5.12.

$$RD_{r\alpha}(\mathcal{M}(\mathcal{D}_1) \parallel \mathcal{M}(\mathcal{D}_2)) \leq \boldsymbol{\epsilon}_{rd} \tag{5.12}$$

where $\boldsymbol{\epsilon}_{rd}$ is a privacy budget. The RDP-CGAN architecture comprises a one-dimensional convolutional autoencoder (1D-CAE) and a GAN. The 1D-CAE goal is: (a) to capture correlations between neighboring features in the input data, (b) to map the input data to a new, compact feature space, reducing dimensionality, (c) to transform categorical features into a continuous space, (d) to enforce privacy by clipping gradients and adding noise during training. The 1D-CAE consists of Enc and Dec, and it is pre-trained separately on the real training set before training the GAN. The Enc consists of five 1-D convolutional layers, each using a Parametric ReLU (PReLU) activation function, except the final layer, which uses the Tanh activation function. The Dec consists of five 1-D transposed convolutional layers arranged in reverse order to the Enc, each using the PReLU activation functions, except the final layer, which uses the Sigmoid activation function. During the pre-training step, each mini-batch of data is partitioned into micro-batches. The Enc transforms each micro-batch into a latent representation, while the Dec reconstructs it. For each micro-batch, the reconstruction loss is computed using

the binary cross-entropy, the gradients of the loss are calculated, and these gradients are then clipped to a fixed norm bound. Privacy is ensured by adding Gaussian noise independently to clipped gradients of each micro-batch and aggregating the noisy gradients. Finally, the 1D-CAE weights are updated using the SGD optimizer.

$\mathbf{G}$ consists of five 1-D transposed convolutional layers, which take the input noise vector $\mathbf{u}$ of size 100 and produce an output of size 128. The $\mathbf{G}$-produced output is then passed through the pre-trained Dec and reconstructed before feeding to $\mathbf{D}_{gen}$. The $\mathbf{D}_{gen}$ consists of five 1-D convolutional layers, each using PReLU activation functions except the final layer, which does not use any activation function. The final layer is replaced by a dense layer with an output size of 1 for making classification. The GAN is trained using the Wasserstein distance loss.

- **CTAB-GAN+** [4] describes a method to produce synthetic data by focusing on both data quality and differential privacy. The CTAB-GAN+ architecture comprises $\mathbf{G}$, $\mathbf{D}_{gen}$ and an auxiliary classifier/regressor Aux. The input tabular data is preprocessed before being passed to $\mathbf{D}_{gen}$ and Aux. The preprocessing technique depends on the type of the features. A feature containing both categorical and numeric values is called a mixed-data feature. The numeric values are normalized using mode-specific normalization that we already discussed in [3], where each value is associated with the most probable mode and normalized accordingly. The mode is then represented using one-hot encoding. For categorical values, one-hot encoding is applied directly. The final transformed value is the concatenation of the normalized numeric value and the one-hot encoded categorical vector. The categorical features with many categories are transformed into the range $(-1, 1)$ using shifted and scaled min-max normalization. The logarithmic transformation is applied to numeric values with long tail distributions. It reduces the gap between tail values and the main data bulk, enabling more efficient mode-specific normalization.

  $\mathbf{G}$ takes random noise vector $\mathbf{u}$ and conditional vector $cond$ as input and produces TSD. The construction of the conditional vector and the training-by-sampling strategy to address the class imbalance problem adopted from [3] that we already discussed; however, in CTAB-GAN+, these are extended to handle numeric and mixed features. $\mathbf{G}$ is trained using the three losses: adversarial loss, information loss, and classification loss adopted from [209] described in Chapter 5 (Section 5.2). $\mathbf{D}_{gen}$ is trained using the adversarial loss, and the Aux is trained using the classification loss. The CNN architecture of $\mathbf{G}$ and $\mathbf{D}_{gen}$ in CTAB-GAN+ is adopted from [209]. Aux is an MLP with four 256-neuron hidden layers. $\mathbf{D}_{gen}$ receives the data produced by the $\mathbf{G}$ and classifies it as real or synthetic. Aux receives the data produced by the $\mathbf{G}$, computes the information loss, and then com-

putes the classification loss by comparing the predicted and ground truth labels. CTAB-GAN+ employs the Differentially Private Stochastic Gradient Descent (DP-SGD) [220] to ensure privacy during training. DP-SGD ensures differential privacy by clipping the gradients to a fixed norm and adding Gaussian noise to the gradients. The privacy budget is tracked using the RDP subsampling moments accountant, which enhances privacy further. In $\mathbf{D}_{gen}$, the DP-SGD is applied only to the gradients of the real data by adding Gaussian noise, while in $\mathbf{G}$, it is applied to the gradients associated with the information loss.

5.3 Motivation

The existing GAN-based TSD producing methods focus on improving data utility or privacy without paying attention to class imbalance problem that is commonly present in real-world datasets. Moreover, some existing methdologoies integrates auxiliary classifier trained jointly with the GAN, but they do not investigate the potential benefit of using a pre-trained auxiliary classifier. In addition, the existing approaches are generally evaluated on limited datasets and do not include statistical tests to validate the significance of the observed performance differences. In this chapter, we address these limitations by proposing a methodology named Focal-AuxCTGAN, which integrates focal loss in both the generator and an auxiliary classifier to encourage the generator to particlaurly focus on minority classes during syntehtic data generation. The auxiliary classifier is evaluated under two training settings: joint-training and pre-training. The evaluation is performed on a wide range of benchmark and cybersecurity datasets. The observed performance differences between settings are validated using a Friedman statistical test. The motivation for adopting this specific approach is the need of producing high-utility TSD to develop more relaiable AI models by enhacning training with syntehic data, particularly for cybersecurity tasks such as malware classification and network intrusion detection.

5.4 Adopted Methodology

We adopt a Focal-AuxCTGAN methodology, that is an extension of the methodology defined in the existing study [3]. It addresses the TSD problem for binary and multiclass classification downstream tasks by improving the quality of the produced data in the utility dimension. The adopted methodology handles categorical and numeric features.

In the preprocessing stage, we transformed the values of the categorical and numerical features into one-hot vectors. The values of the categorical features are directly transformed into one-hot vectors. To transform numeric features, we adopted mode-specific normalization inspired by [3], following four steps. First, the modes for each numeric feature are estimated using a VGM [215], and a Gaussian mixture is fitted to the modes. Second, for each value of the numeric feature, the probability density of how the numeric feature may belong to the estimated modes is computed, and the mode with the highest probability is selected. Third, each numeric value is normalized according to its assigned mode. Finally, the numeric feature values are transformed into one-hot vectors to indicate the mode of the features, along with continuous values representing the value within the mode.

The Focal-AuxCTGAN architecture comprises three components: the generator $(\mathbf{G})$, discriminator $(\mathbf{D}_{\text{gen}})$, and an MLP as auxiliary classifier Aux (see Figure 5.1).$\mathbf{G}$ is a fully connected neural network, consisting of two hidden layers with batch normalization and ReLU activation, followed by a synthetic row representation produced using mixed Tanh activation function and Gumbal-Softmax activation function. $\mathbf{D}_{gen}$ is a fully connected neural network consisting of two hidden layers with Leaky ReLU activation and dropout on each layer. $\mathbf{G}$ and $\mathbf{D}_{\text{gen}}$ compete with each other. The $\mathbf{G}$ produces the TSD and $\mathbf{D}_{\text{gen}}$ differentiate between the TSD and the real data. The goal of the discriminator is to maximize the probability of correctly classifying both real and the synthetic examples, while the goal of the generator is to minimize the discriminator ability to differentiate between real and synthetic examples. The Aux architecture consists of four 256-neuron hidden layers. It computes the Focal loss between the predicted labels on the synthetic data and real data and encourages $\mathbf{G}$ to produce synthetic data that closely resembles the real data in terms of class balance.

The generator $\mathbf{G}$ of the GAN architecture is trained with the focal loss [221] and the WGAN-GP loss. Both these losses are applied independently. The focal loss addresses the class imbalance problem by weighting examples based on their predicted class confidence score corresponding to the true class. The parameter $\boldsymbol{\gamma}$ reduces the loss contribution of the examples with high confidence scores (easy-to-classify) while having a proportionally smaller impact on examples with low confidence scores (hard-to-classify). The focal loss allows the decision model to focus more on examples from minority classes. The mathematical formulation of the focal loss is defined in equation 5.13.

$$\mathcal{F}_{\text{loss}} = -\boldsymbol{\alpha}_t(1-p_t)^{\gamma}\log(p_t) \tag{5.13}$$

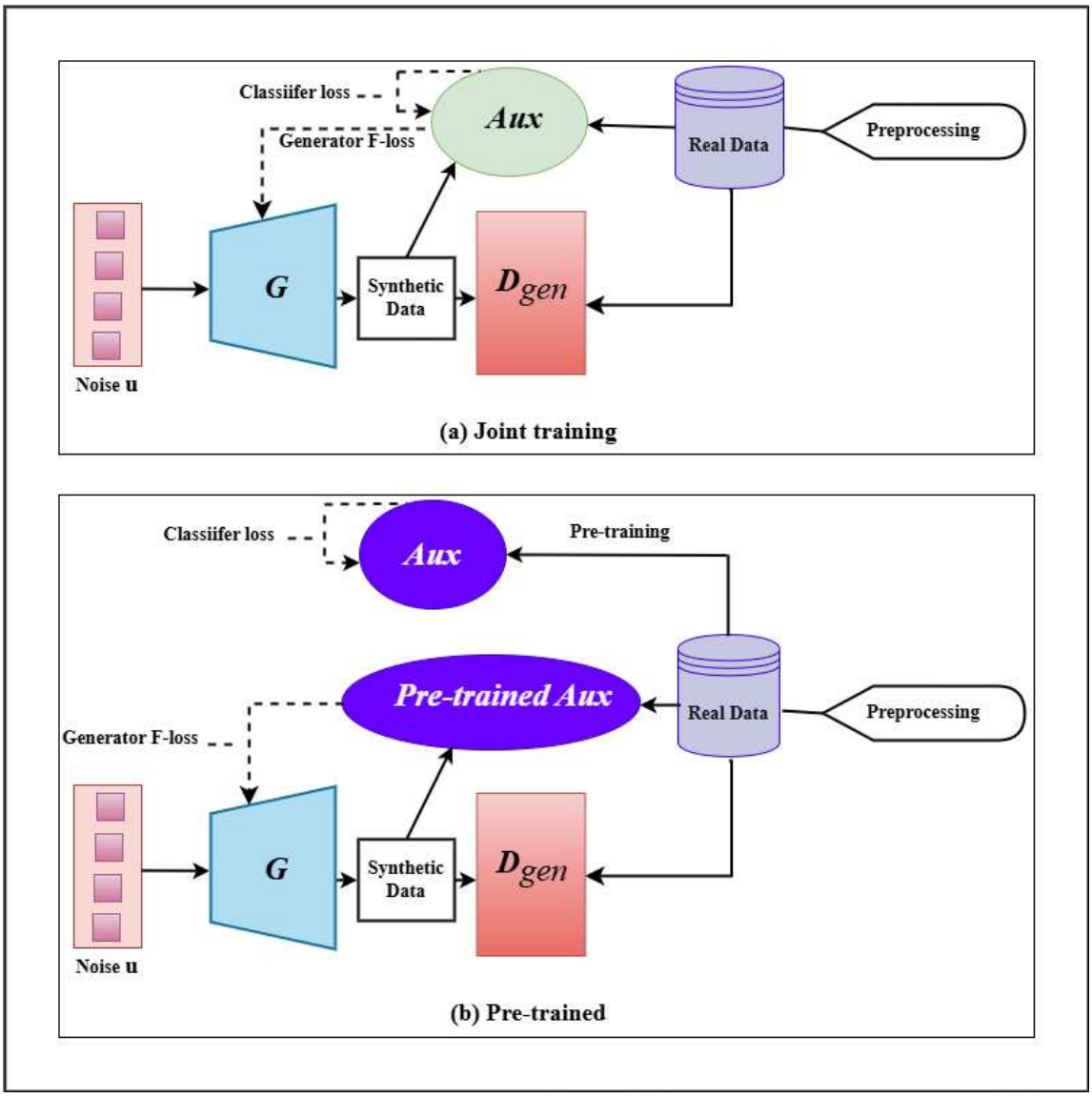


Figure 5.1: Focal-AuxCTGAN architecture. In 5.1 (a), the Aux classifier is trained in parallel with $\mathbf{G}$ and $\mathbf{D}_{gen}$. In 5.1 (b), Aux is pre-trained.

where p_t is the predicted confidence score for a class corresponding to the true class. $\boldsymbol{\alpha}_t$ is a class rebalancing factor. The WGAN-GP loss stabilizes the adversarial training process [218]. It ensures the smooth gradient flow through a gradient penalty and enables the generator to model complex relationships between numeric and categorical features in tabular data.

In addition, $\mathbf{G}$ takes random noise $\mathbf{u}$ sampled from a normal Gaussian distribution concatenated with a conditional vector $cond$ to produce tabular synthetic data. $\mathbf{D}_{gen}$ classifies the produced data as real or synthetic. Aux is trained for classification using the focal loss also. In the following we refer to the focal loss of Aux as the generator focal loss, between the predicted labels on the synthetic data and the real labels. It also computes the focal loss between the predicted labels on the real data and the real labels, which we refer to as the classifier focal loss. The Aux is used in two scenarios: (a) joint training and (b) pre-trained. In particular, Aux is trained in parallel with the GAN in the joint training scenario. During the GAN training, it updates $\mathbf{G}$ according the generator focal loss, encouraging $\mathbf{G}$ to produce synthetic data that closely resembles the real data in terms of class balance. Additionally, Aux is trained with the classifier focal loss to gain accuracy in label prediction, particularly for minority classes. In the pre-trained scenario, the training of Aux is performed separately before the training of the GAN architecture. During the training of the GAN, the pre-trained Aux is considered to compute the generator focal loss and updates $\mathbf{G}$ with this computed loss.

5.5 Evaluation

This section illustrates the dataset description and evaluation metrics used in this study.

5.5.1 Datasets

Both the baseline CTGAN and the focal-AuxCTGAN model obtained with both the joint training setting and the pre-training setting are evaluated using ten datasets from various domains (comprising cybersecuirty domians). This ensures the generality of focal-AuxCTGAN for TSD tasks. The datasets are described in detail in Table 5.1.

All datasets are publicly available and are commonly used for classification tasks. Six datasets belong to binary classification, while the remaining five belong to multi-class classification. For the datasets: Mushroom [21], Credit [22], Census [23], Cover-Type [24], Covid [25], Stellar [26], NB-15 [27], and MalMem2022 [222], 50K examples were randomly

Table 5.1: Description of the datasets used in this study. The "Dataset Name" column lists the name of each dataset, the "Train/Test Split" column specifies the number of examples used for training and testing, the "Dataset Type" column indicates the domain to which each dataset belongs, the "Categorical features" column shows the number of categorical features in each dataset, the "Numerical features" column lists the number of numerical features, the "No. of Classes" column reports the number of classes in each dataset, and the "Classification type" column indicates whether the dataset is binary or multi-class.

Dataset Name	**Dataset type**	**Train/Test split**	**Categorical features**	**Numeric-features**	**No. of classes**	**Classification-Type**
Mushroom	Mushroom classification	40K/10K	18	3	2	Binary
Bank-Loan-Modeling	Loan risk assessment	4K/1K	1	12	2	Binary
Adult	Census Income	39073/9769	9	7	2	Binary
Credit	Credit card fraud detection	40K/10K	1	30	2	Binary
Census	Census Income	40K/10K	29	13	2	Binary
Cover-type	Forest cover type	40K/10K	3	10	7	Multi-class
Covid	Covid-19	40K/10K	14	6	7	Multi-class
Stellar	Astronomy	40K/10K	1	15	3	Multi-class
MalMem 2022	Malware memory analysis	40K/10K	1	55	4	Multi-class
NB-15	Network intrusion detection	40K/10K	4	39	10	Multi-class
WinPE	Windows malware detection	9012/9012	1	2381	2	Binary

selected using a stratified sampling approach. Datasets were then split into training set (80%) and testing set (20%). For the dataset WinPE 18024 examples were randomly selected using stratified sampling. The datasets Bank-loan-modeling [28], and Adult [29] datasets were used without any sampling step. The features 'Wilderness-Area' and 'Soil-type' of the Cover-Type dataset were original provided in the one-hot encoding format. We transformed these features back into the categorical form. The distribution of classes in the considered datasets is reported in Table 5.2. The datasets were selected from different domains to ensure diversity and to evaluate the proposed methdology across a broad range of classification tasks. In particular, NB-15, MalMem2022, WinPE, and Credit are from cybersecurity tasks, including network intrusion detection, malware memory analysis, Windows malware detection, and fraud detection. The datasets Bank-Loan-Modeling, Adult, and Cover-Type are widely used in existing studies for TSD generation. The datasets Mushroom, Covid, Stellar, and Census, were included for further domain diversity. All datasets were in structured format. In the preprocessing step, the values of the categroical and numerical features were transformed into one-hot vectors. The class distribution of all datasets, shown in Table 5.2, shows the presence of

both balanced and highly imbalanced datasets, which is important for evaluating the effectiveness of the proposed TSD producing methdology under different data conditions.

5.5.2 Implementation Details

All experiments are performed using Python 3.10. In the training setup, both GAN and Aux are trained using the Adam optimizer with learning rate 0.0002, neurons in each layer 256, batch size 500, weight decay 0.000001 and epochs 300. To optimize the focal loss specifically, we use Optuna [223], an open-source library, to find the optimal values of the focal loss parameters $\boldsymbol{\alpha}_t$ and $\boldsymbol{\gamma}$, to minimize the focal loss. The parameter ranges explored are $\boldsymbol{\gamma} \in [0.0, 10.0]$ and $\boldsymbol{\alpha}_t \in [0.0, 1.0]$. We train the MLP classifier, which has the same architecture, optimizer, and parameters as the Aux component of the Focal-AuxCTGAN model. The Optuna search is performed on 50 trials with 300 epochs per trial. Optuna employs the TPE algorithm [49] to efficiently explore the hyper-parameter space during optimization. The working of TPE algorithm is already described in chapter 2 (section 2.3). In each trial, Optuna trains the MLP classifier with a unique combination of $\boldsymbol{\gamma}$ and $\boldsymbol{\alpha}_t$ values and evaluates the resulting focal loss. To improve efficiency, Optuna uses the median pruner algorithm to terminate underperforming trials early. The Median Pruner algorithm calculates the median performance of all completed trials at a specific stage, such as epoch or iteration, and compares this value with the performance of the current trial at the same stage. If the current trial underperforms the median, then the process is pruned (terminated) to save computational resources. Through this optimization process, Optuna explores different combinations of values of $\boldsymbol{\gamma}$ and $\boldsymbol{\alpha}_t$ within the defined ranges and identifies the values that minimize the focal loss.

The predictive performance of the produced TSD is evaluated using an MLP classifier MLP_{eval}. The MLP_{eval} architecture consists of two hidden layers, each comprising a fully connected layer, a batch normalization layer, and a dropout layer. To ensure the optimal performance of the classifier, hyperparameter tuning is performed using the Optuna library, focusing on maximizing the F1-Score. The hyperparameter search space includes neurons per layer with values ranging from [32, 64, 128, 256, 512, 1024], dropout rates ranging from 0 to 1, and a learning rate varying between 0.0001 and 0.001. The Optuna search is performed on 50 trials with 300 epochs per trial.

To validate the performance differences among CTGAN and focal-AuxCTGAN under joint-training and pre-training settings, the Friedman test followed by the Nemenyi post-hoc test is implemented using the Orange Data Mining Library (ODML)[30].

5.5.3 Evaluation metrics

We evaluate the predictive performance of the MLP classification model referred as MLP_{eval} for the downstream binary or multi-class classification task, according to the considered dataset.

MLP_{eval} is trained on both the tabular real training set and the synthetic training dataset produced by both CTGAN and Focal-AuxCTGAN. MLP_{eval} was evaluated on the corresponding real testing set. The accuracy performance was measured using Area Under the Receiver Operating Characteristic Curve (AUC-ROC) and F1-Score (macro, micro, weighted) metrics. In binary classification tasks, the positive class refers to the target class of interest (commonly denoted as "1", while the other class is considered as the negative class. In multi-class classification tasks, the One-vs-Rest (OvR) strategy was used to handle each class as the positive class one at a time, with all other classes grouped together were handled as the negative class. In Chapter 4, the F1-Score was introduced as a metric considered to evaluate the classification performance for Windows PE malware detection models. In this chapter, F1-Score evaluates the classification performance for positive and negative examples. Specifically,

- AUC-ROC (AUC) measures the model ability to separate positive examples from negative examples across a range of probability thresholds, which vary from 0 to 1. The ROC is a resulting curve to represent a trade-off between recall and fpr (false positive rate) at each threshold. The AUC is the total area under the ROC curve. The final score is obtained using a macro-average approach, ensuring equal contribution from each class. The closer the value of AUC to 1, the better the performance of the classification model.

- $\text{Fscore}_{\text{macro}}$ ($\mathsf{F}_{\text{macro}}$) measures the Fscore for each class and then computes the average Fscore across all classes considering the equal contribution of each class, i.e., $\mathsf{F}_{\text{macro}} = \frac{1}{C}\sum_{i=1}^{C}\mathsf{F}_i$, where F_i is the Fscore of $i-th$ class.

- $\text{Fscore}_{\text{micro}}$ ($\mathsf{F}_{\text{micro}}$) sums the tp, fp, and fn across all classes and then computes the overall F1-Score, i.e., $\mathsf{F}_{\text{micro}} = 2\frac{\text{prec}_{\text{micro}}.\text{recall}_{\text{micro}}}{\text{prec}_{\text{micro}}+\text{recall}_{\text{micro}}}$, where $\text{prec}_{\text{micro}} = \frac{\sum_{i=1}^{C}\text{tp}_i}{\sum_{i=1}^{C}\text{tp}_i+\sum_{i=1}^{C}\text{fp}_i}$, and $\text{recall}_{\text{micro}} = \frac{\sum_{i=1}^{C}\text{tp}_i}{\sum_{i=1}^{C}\text{tp}_i+\sum_{i=1}^{C}\text{fn}_i}$.

- $\text{Fscore}_{\text{weighted}}$ ($\mathsf{F}_{\text{weighted}}$) measures the Fscore for each class and then computes the weighted average considering the percentage of examples per class as class weights, i.e., $\mathsf{F}_{\text{weighted}} =$

$\sum_{i=1}^{C} p_i \mathsf{F}_i$, where p_i is the number of examples for class C_i divided by the number of examples.

5.6 Results and discussion

The AUC, $\mathsf{F}_{\text{macro}}$, $\mathsf{F}_{\text{micro}}$, $\mathsf{F}_{\text{weighted}}$ of MLP_{eval} reported in Table 5.3. In our analysis, we observe that no single scenario consistently outperforms the others across all datasets for a specific metric in downstream classification tasks. We assume that this may depend on the different characteristics of the considered datasets. Hence, we group the datasets into six groups: binary datasets, multi-class datasets, balanced datasets, imbalanced datasets, numeric datasets, and mixed datasets.

In each dataset scenario, we perform the Friedman test [224, 225] followed by the Nemenyi test [226] to compare CTGAN, AuxCTGAN in Joint-training setting and AuxCTGAN in Pre-training setting across each dataset group, considering all metrics independently. The Friedman test is a non-parametric statistical test for comparing multiple methods across multiple datasets. The Friedman test measures the statistical differences between the compared methods by ranking them based on their metric scores for each dataset within a given group. The method with the higher metric score receives a higher rank, while the method with a lower metric score receives a lower rank. After ranking all methods for each dataset, the average rank of each method across all datasets within the group is computed. Next, the average rank difference between methods is computed using the Friedman statistics, which provides a p-value. The mathematical formulation of the Friedman statistics is defined in equation 5.14.

$$\mathcal{Q} = \frac{12\mathcal{D}_{\mathbf{n}}}{v(v+1)} \left[\sum_{j=1}^{v} \bar{R_j}^2 - \frac{v(v+1)^2}{4} \right] \tag{5.14}$$

where $\mathcal{D}_{\mathbf{n}}$ is the number of datasets, v is the number of methods, $\bar{R_j}$ is an average rank of method $j-th$ and constants 12 and 4 are normalization terms. A p-value below a threshold (e.g., 0.05) shows a statistically significant difference between the scenarios, while a p-value above this threshold means no significant difference. All methods are evaluated independently for each dataset group and across all evaluation metrics.

The Friedman test provides an overall assessment of the statistical difference; however, it does not determine which method is significantly different from another. The Nemenyi

test, a post-hoc analysis method, determines the pairwise differences between the compared methods. The Nemenyi test computes the Critical Difference (CD) and produces diagrams based on the computed CD. The mathematical formulation of the CD is defined in equation 5.15.

$$CD = q_\alpha \cdot \sqrt{\frac{v(v+1)}{12\mathcal{D}_\mathbf{n}}} \tag{5.15}$$

where q_α is the predefined parameter with a value 0.05. The CD diagram suggests which method is performing better in the pairwise comparison for the considered metric within the processed datasets. In the diagram, the scenarios with no statistically significant differences are connected by the same line, while the methods with significant differences are not connected by any line.

5.6.1 Binary dataset group

In a binary dataset group, we considered binary classification datasets, i.e., Mushroom, Bank-loan-modelling, Adult, Credit, WinPE, and Census. Figure 5.2 shows the critical difference diagrams obtained for AUC, $\mathsf{F}_{\text{macro}}$, $\mathsf{F}_{\text{micro}}$, and $\mathsf{F}_{\text{weighted}}$.

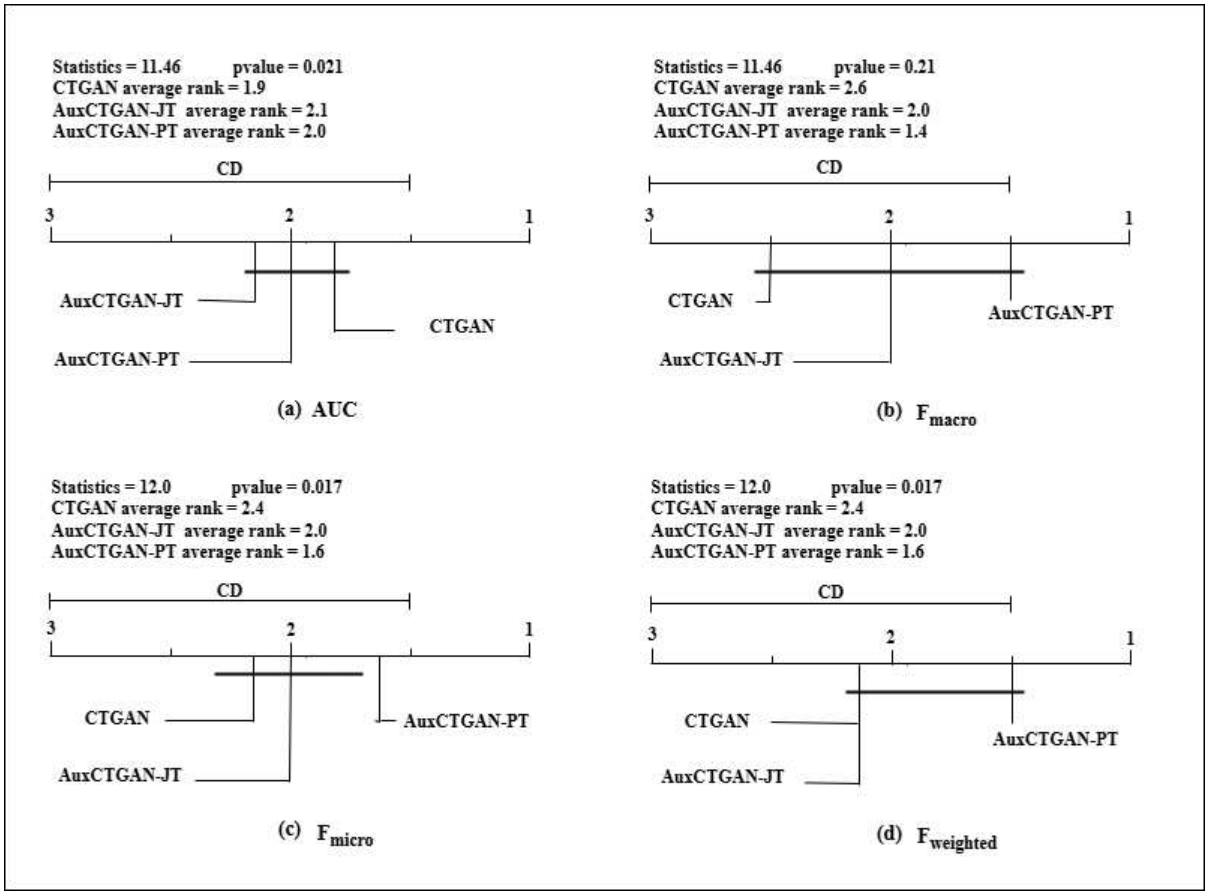


Figure 5.2: Critical difference diagram of Binary datasets group

The statistical test shows that AuxCTGAN with Pre-training outperforms remaining methods with respect to all metrics except for F_{micro}. On the other hand, the p-value is not statistically significant in the test performed with F_{macro}. In general, the statistical analysis suggests that the AuxCTGAN in the Pre-training setting is a reasonable choice with binary datasets.

5.6.2 Multi-class dataset group

The multi-class classification datasets include: Cover-Type, Covid, Stellar, MalMem2022, and NB-15. Figure 5.3 shows the critical difference diagrams obtained for AUC, F_{macro}, F_{micro}, and $F_{weighted}$.

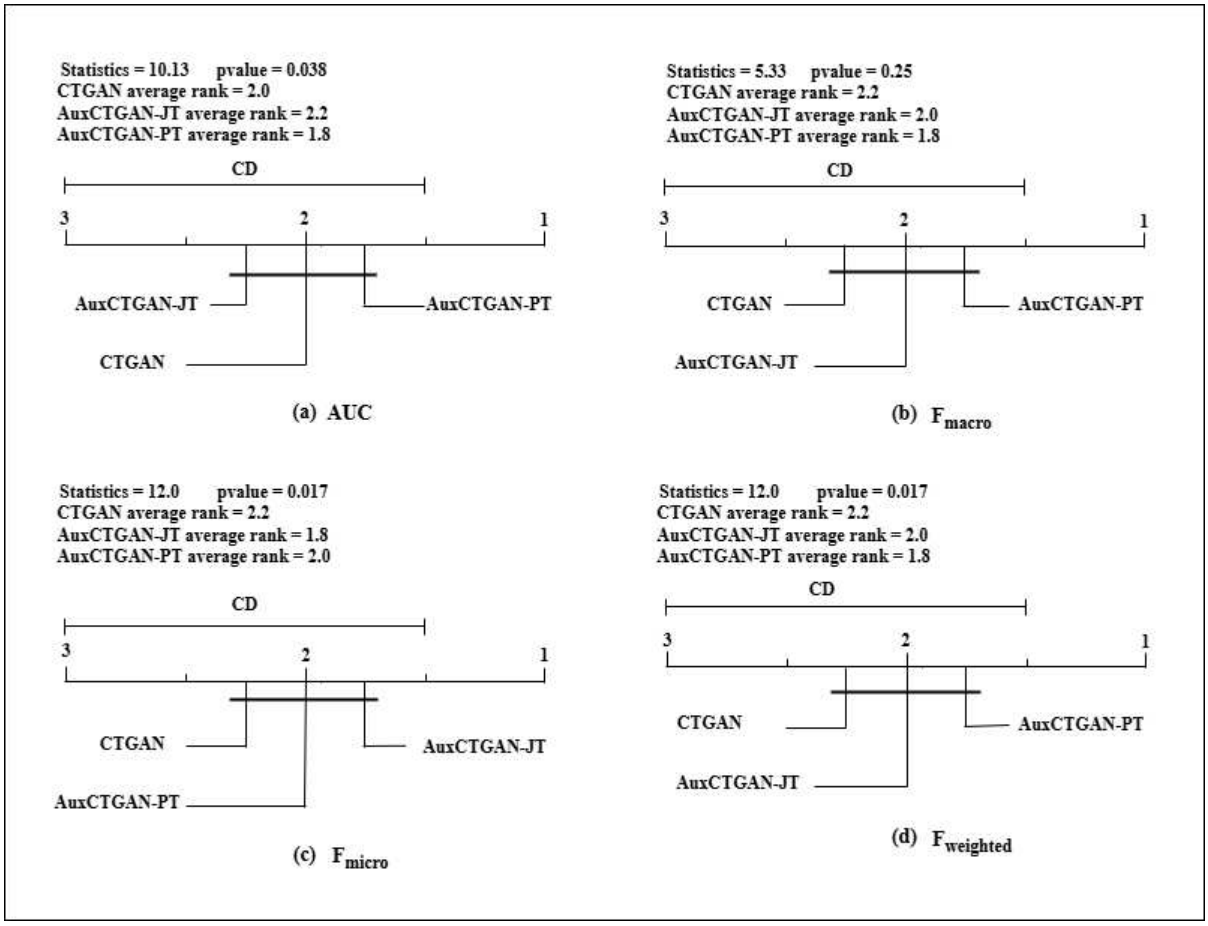


Figure 5.3: Critical difference diagram of Multi-class datasets group

The diagrams show that AuxCTGAN with Pre-training setting outperforms remaining methods with respect to AUC, and $\mathsf{F}_{\text{weighted}}$, while AuxCTGAN with Join-training setting outperforms remaining methods with respect to $\mathsf{F}_{\text{macro}}$, and $\mathsf{F}_{\text{micro}}$. So, at least, a configuration of AuxCTGAN outperforms CTGAN independently of the considered metric.

5.6.3 Balanced dataset group

We consider a dataset balanced if each class contains more than 10% of the total examples. The balanced datasets of this study are: Mushroom, Adult, Stellar, WinPE, and Malmem2022. Figure 5.4 shows the critical difference diagrams obtained for AUC, $\mathsf{F}_{\text{macro}}$, $\mathsf{F}_{\text{micro}}$, and $\mathsf{F}_{\text{weighted}}$. The diagrams show that AuxCTGAN with Pre-training setting outperforms remaining methods with respect to AUC only, while AuxCTGAN with Joint-training setting outperforms remaining methods with respect to $\mathsf{F}_{\text{macro}}$, $\mathsf{F}_{\text{micro}}$, and

$F_{weighted}$. In general, we consider that AuxCTGAN with Joint-training setting should be used in the balanced data scenario.

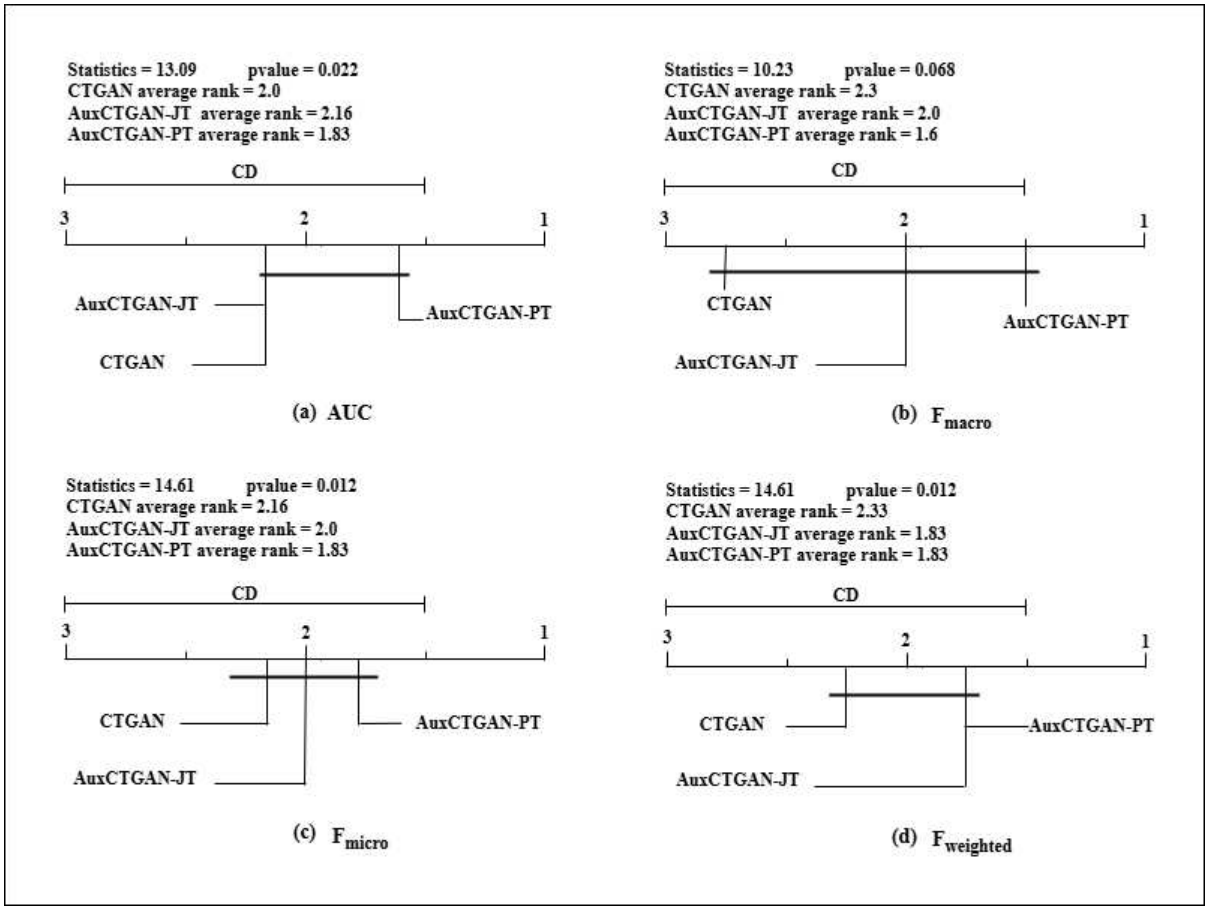


Figure 5.4: Critical difference diagram of Balanced datasets group

5.6.4 Imbalanced dataset group

We consider a dataset imbalanced if it contains at least a class that contains less than or equal to 10% of the total examples. Imbalanced datasets, of this study are: Bank-loan modeling, Credit, Census, Cover-Type, Covid, and NB-15. Figure 5.5 shows the critical difference diagrams obtained for AUC, F_{macro}, F_{micro}, and $F_{weighted}$. The diagrams show that AuxCTGAN with Pre-training setting outperforms remaining methods with respect to all metrics supporting the conclusion that AuxCTGAN with Pre-training setting should be used in the imbalanced data scenario.

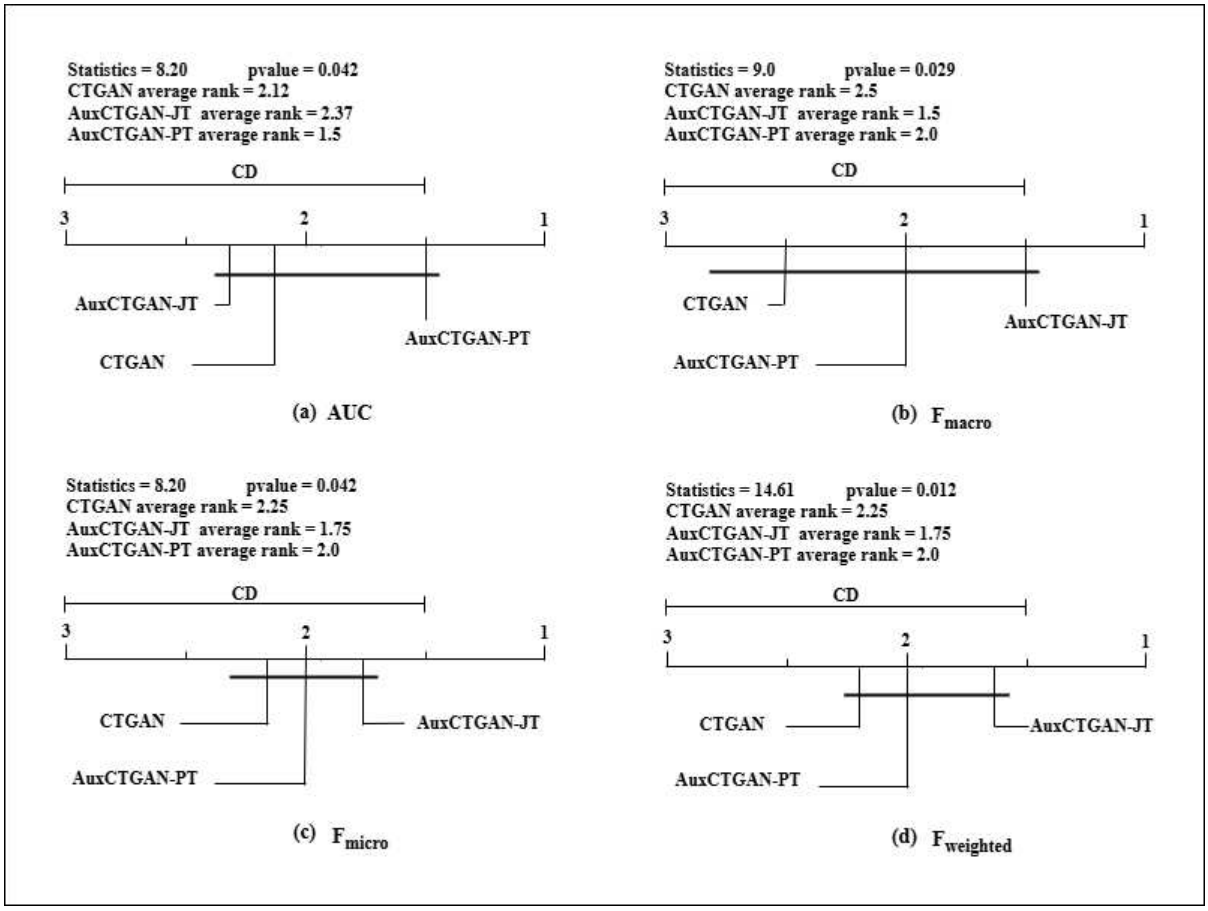


Figure 5.5: Critical difference diagram of imbalanced datasets group

5.6.5 Numeric dataset group

We perform a two-step analysis to classify datasets into the numeric group. First, we use the Pandas [31] library to check the data type of each input feature in the dataset. Features with numeric types, e.g., int, float are initially considered as numeric. Then we manually verify the meaning and value distribution of each input feature to ensure consistency with the numeric classification. The datasets in the numeric group are:Stellar, Credit, bank-loan Modeling, WinPE, and Malem2022. Figure 5.6 shows the critical difference diagrams obtained for AUC, F_{macro}, F_{micro}, and $F_{weighted}$. The diagrams show that AuxCTGAN with Pre-training setting outperforms remaining methods with respect to all metrics except for AUC (where AuxCTGAN with Pre-training is the runner-up method). In general, we consider, AuxCTGAN with Pre-training setting a reasonable choice with numeric datasets.

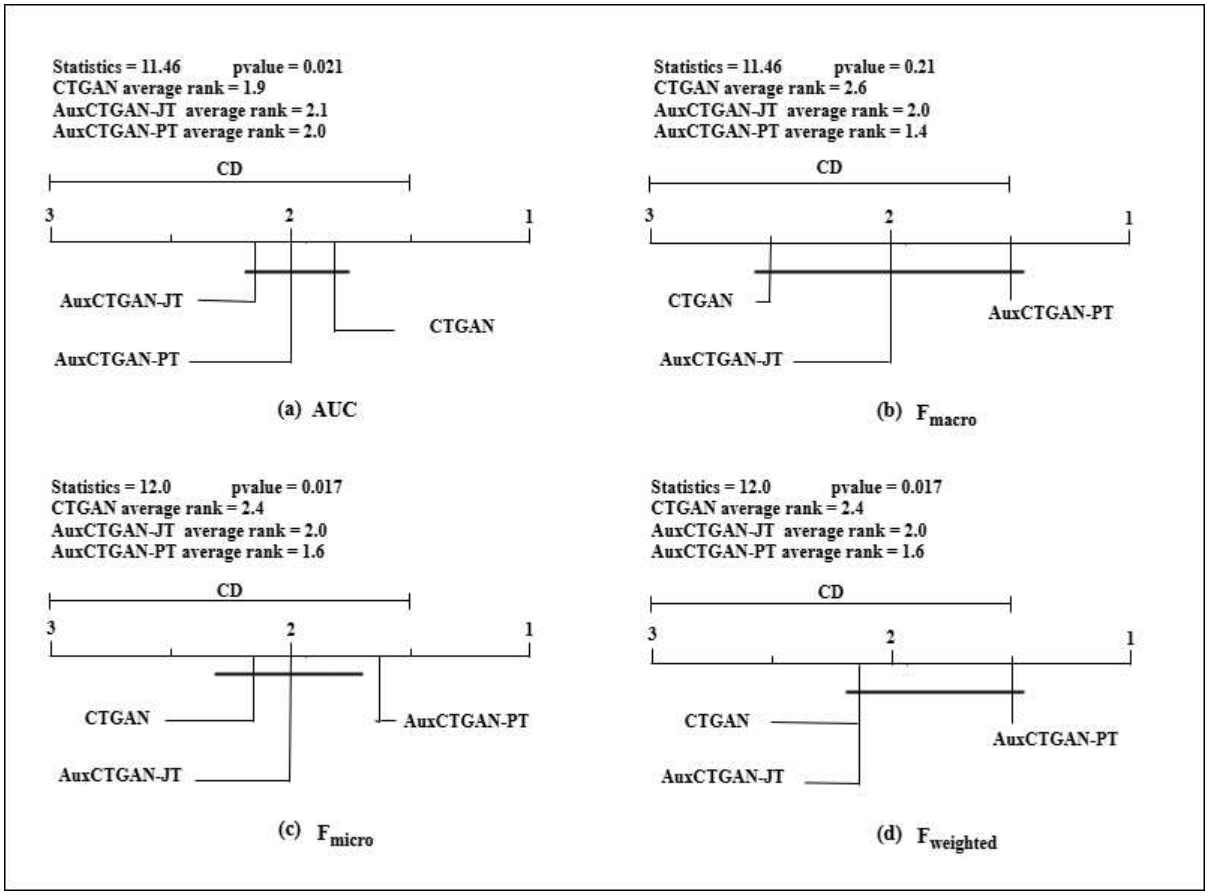


Figure 5.6: Critical difference diagram of numeric datasets group

5.6.6 Mixed dataset group

A dataset that contains both numeric and categorical features is considered as a mixed dataset. The mixed datasets of this study are: Mushroom, Covid, Census, Cover-Type, NB-15, and Adult. Figure 5.7 shows the critical difference diagrams obtained for AUC, F_{macro}, F_{micro}, and $F_{weighted}$. The diagrams show that AuxCTGAN with Joint-training setting outperforms remaining methods with respect to all metrics except for AUC (where AuxCTGAN with Pre-training is the best method). In general, we consider, AuxCTGAN with Joint-training setting a reasonable choice with mixed datasets.

5.7 Lessons learned and future research direction

In this chapter, we compare the accuracy performance (utility) of the generative adversarial method CTGAN and the proposed methodology Focal-AuxCTGAN. The Focal-

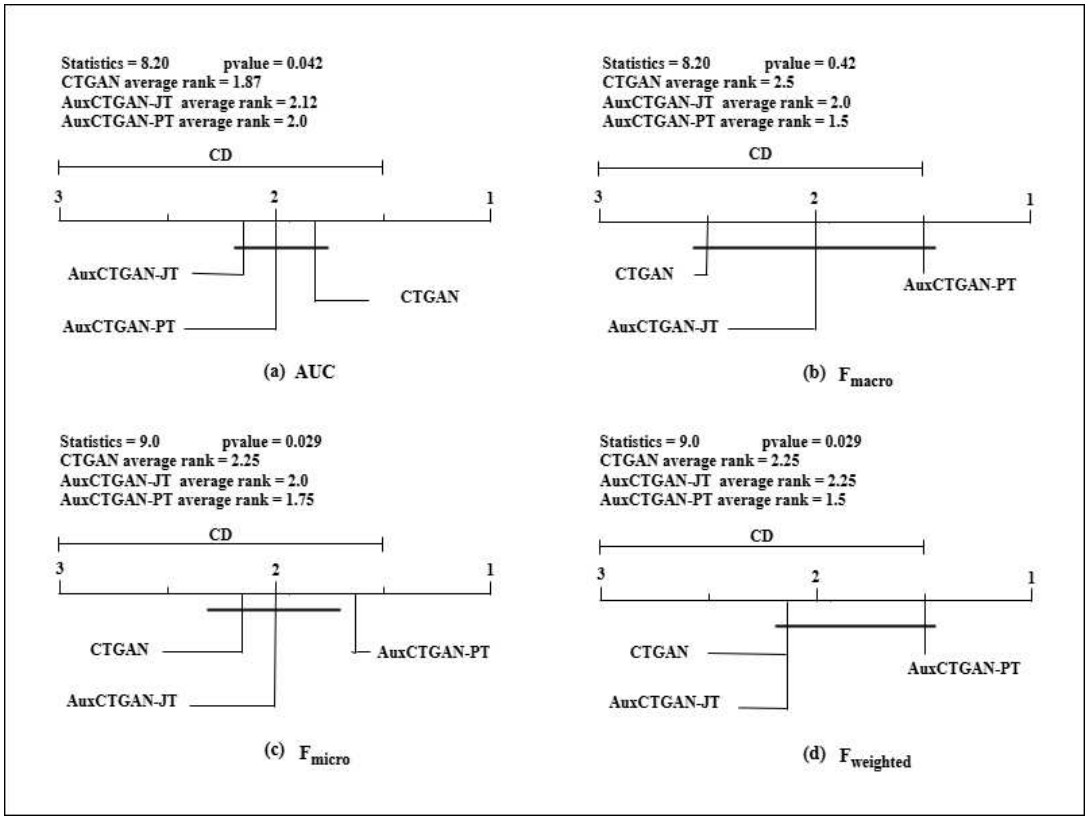


Figure 5.7: Critical difference diagram of mixed datasets group

AuxCTGAN model was trained in two settings: Joint-training and Pre-training. The utility of the produced TSD for downstream classification task was evaluated using an MLP_{eval} classifier according to four metrics: AUC, $\mathsf{F}_{\mathsf{macro}}$, $\mathsf{F}_{\mathsf{micro}}$, and $\mathsf{F}_{\mathsf{weighted}}$. These evaluation metrics measure different aspects of the MLP_{eval} classifier performance, such as class separation ability and the trade-off between precision and recall depending on the class distribution or independently of the class distribution.

Our evaluation shows that no method (CTGAN, AuxCTGAN in the Joint-training setting and AuxCTGAN in the Pre-training setting) consistently outperforms the remaining methods across all datasets for any specific metric. Hence, we analysed the methods' performance in subgroup of datasets by using a statistical process. The following conclusion is drawn.

Data-specific characteristics such as class distribution and features type affect the performance of the TSD method. The effectiveness of each method varies based on the dataset type and the chosen evaluation metric.

In future work, we plan to extend our work by adding synthetic data fidelity and privacy dimensions. Our current work focuses on downstream classification tasks; we aim to expand it to include regression tasks, increasing the adoption of TSD across diverse real-world scenarios. Additionally, the data utility can be further enhanced by integrating XAI techniques during the TSD generation. This integration will ensure that feature importance and correlations observed in the real data are preserved in the synthetic data, resulting in the reliability of synthetic data. Furthermore, we plan to explore DP-SGD techniques to ensure data protection against potential attackers without compromising utility.

Although Chapters 4 and 5 investigate different research directions, they are both motivated by the need to improve the reliability of AI-based models in cybersecurity. In Chapter 4, the focus is on evaluating the vulnerability of Windows PE malware detectors to adversarial attacks and enhancing robustness through adversarial training. In Chapter 5, the focus is on producing high-utility TSD to address the challenges such as data imbalance and limited availability of datasets for training AI-based models, particularly for cybersecurity tasks like malware and intrusion detection. Both studies adopt data-centric approaches to strengthen AI-based security solutions. Reliable and resilient models require both robustness against adversarial manipulation and access high-quality training data.

Table 5.2: Description of class distribution in the considered datasets. The Dataset Name column specifies the name of each dataset, the Class column lists the classes name in the target feature of each dataset, the Training Set Percentage Value column indicates the percentage distribution of each class in the training set, and the Testing Set Percentage Value column represents the percentage distribution of each class in the testing set.

Dataset Name	Class	Training Set Percentage Value	Testing Set Percentage Value
Mushroom	p	55.49	55.49
	e	44.51	44.51
Bank Loan-Modeling	0	90.4	90.4
	1	9.6	9.6
Adult	<=50K	76.07	76.07
	>50K	23.93	23.93
Credit	0	99.02	99.02
	1	0.99	0.99
Census	<50K	93.77	93.77
	>=50K	6.23	6.11
Cover-Type	1	36.46	36.46
	2	48.76	48.76
	3	6.15	6.16
	4	0.47	0.47
	5	1.64	1.63
	6	2.99	2.99
	7	3.53	3.53
Covid	1	0.82	0.82
	2	0.18	0.18
	3	36.39	36.38
	4	0.30	0.30
	5	2.49	2.49
	6	12.22	12.22
	7	47.61	47.61
Stellar	GALAXY	59.44	59.45
	STAR	21.60	21.59
	QSO	18.96	18.96
MalMem2022	0	50	50
	1	16.71	16.71
	2	17.10	17.10
	3	16.19	16.19
NB-15	Normal	31.94	44.94
	Generic	22.81	22.92
	Exploits	19.05	13.52
	Fuzzers	10.37	7.36
	Dos	6.99	4.97
	Recon-naissance	5.98	4.25
	Analysis	1.14	0.82
	Backdoor	1.00	0.71
	Shellcode	0.65	0.46
	Worms	0.08	0.05
WinPE	0	49.9	49.9
	1	50	50

Table 5.3: Accuracy performance (AUC, $\mathsf{F}_{\text{macro}}$, $\mathsf{F}_{\text{micro}}$, and $\mathsf{F}_{\text{weighted}}$) of MLP_{eval} trained on the TSD training set and evaluated on the real testing set, across the study datasets. Each TSD training set was produced using both the CTGAN and AuxCTGAN in the Joint-training (JT) and Pre-training (PT) settings, respectively. For each dataset, for each metric, the best results are in bold.

Dataset Name	**AUC**			**F macro**			**F micro**			**F weighted**		
	CTGAN	AuxCTGAN-JT	AuxCTGAN-PT	CTGAN	AuxCTGAN-JT	AuxCTGAN-PT	CTGAN	AuxCTGAN-JT	AuxCTGAN-PT	CTGAN	AuxCTGAN-JT	AuxCTGAN-PT
Mushroom	0.73	**0.84**	**0.84**	0.68	**0.74**	**0.74**	0.69	**0.74**	**0.74**	0.69	0.74	**0.75**
Bank-loan-Modeling	**0.93**	**0.93**	0.92	0.72	0.73	**0.74**	0.85	**0.86**	**0.86**	0.87	**0.88**	**0.88**
Adult	**0.88**	0.87	**0.88**	**0.77**	**0.77**	0.76	**0.82**	**0.82**	0.81	0.82	**0.83**	0.82
Credit	0.93	0.92	**0.96**	0.75	0.62	**0.78**	0.98	0.95	**0.99**	**0.99**	0.97	**0.99**
Census	**0.90**	0.84	0.82	**0.72**	0.69	0.66	**0.94**	0.93	0.88	**0.94**	0.93	0.90
Cover-Type	**0.91**	**0.91**	**0.91**	0.51	0.53	**0.54**	0.63	0.64	**0.65**	0.62	0.64	**0.65**
Covid	0.59	0.59	**0.60**	0.16	**0.19**	0.17	**0.50**	0.48	0.44	0.46	**0.47**	0.43
Stellar	**0.98**	**0.98**	**0.98**	0.93	**0.94**	0.92	0.93	**0.94**	0.93	0.93	**0.94**	0.93
MalMem-2022	**0.88**	0.87	**0.88**	0.58	0.58	**0.59**	0.72	0.72	**0.73**	**0.72**	**0.72**	**0.72**
NB-15	**0.81**	**0.81**	**0.81**	0.27	**0.32**	**0.32**	0.56	**0.61**	0.59	0.61	**0.65**	0.64
WinPE	**0.76**	**0.76**	**0.76**	0.41	0.43	**0.45**	0.44	0.49	**0.50**	0.60	**0.63**	0.61

Chapter 6

Conclusions

This chapter summarizes the contributions of this thesis, focusing on conclusions drawn from the research performed regarding the adversarial attacks and defense in Windows PE malware detection and TSD generation in cybersecurity and future research directions of this thesis.

6.1 Conclusions

In this thesis, we explored two topics referred to the cybersecurity domain: evaluating the performance of the adversarial attack methods and defensive strategies in Windows PE malware detection and exploring the problem of the TSD generation in network intrusion detection and malware detection problems. The results of the conducted evaluation are reported in Chapters 4 and 5, respectively.

In Chapter 4, we conducted an evaluation study of five state-of-the-art attack methods: Extend, Full DOS, Shift, FGSM (padding + slack), and GAMMA. These methods produced realistic adversarial Windows PE malware files while preserving their functionality. Our study confirmed conclusions drawn in [1] that AI models for Windows PE malware detection, including MalConv and LGBM, are vulnerable to adversarial attacks. Notably, adversarial malware produced with Extend, Shift and GAMMA in the raw byte-based space exhibited transferability to the feature-engineered space, impacting the robustness of models trained on engineered features. To enhance the resilience of AI models, we evaluated the performance of adversarial training. The evaluation results show that the use of the adversarial training strategy with GAMMA-produced samples can be an effective strategy to strengthen the LGBM model trained with LIEF

features against this attack type. Using SHAP, we explained how adversarial attacks changed LIEF feature importance and influence model decisions. This analysis showed that adversarial attacks reduce the detection accuracy and shift the model's reliance on important features.

In Chapter 5, we evaluated the utility of two GAN-based methods for TSD generation: CTGAN and Focal-AuxCTGAN. Our evaluation of the utility of these methods was conducted in multiple classification problems comprising network intrusion detection and malware detection problems. The results showed that no method consistently outperforms the other method across all datasets and evaluation metrics. In particular, the utility of each method depends on the characteristics of the dataset.

6.2 Limitations and future works

The study performed on Windows PE malware detection focused on evasion attacks and neither explored malware obfuscation methods such as encryption, metamorphism, or packing, nor poisoning methods. In addition, this study focused on the Windows operating system and PE files, it can be extend to other file types and operating system such as Android. The empirical study was conducted on a selection of attack methods (Extend, Full DOS, Shift, FGSM (padding+slack), and GAMMA) and a single defensive strategy (adversarial training). Other attack methods and defensive strategies can be used in an extension of the conducted evaluation study.

The TSD generation study was limited to the consideration of GANs, with a focus on utility of TSD for classification tasks. Regression tasks and privacy-preserving techniques, such as differential privacy, were not explored, potentially limiting the adoption of TSD in sensitive domains. This study was conducted by integrating an MLP auxiliary classifier; integration of an explainability technique can be an extension of this study.

Bibliography

[1] L. Demetrio, S. E. Coull, B. Biggio, G. Lagorio, A. Armando, and F. Roli, "Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection," *ACM Transactions on Privacy and Security (TOPS)*, vol. 24, no. 4, pp. 1–31, 2021.

[2] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[3] L. Xu, M. Skoularidou, A. Cuesta-Infante, and K. Veeramachaneni, "Modeling tabular data using conditional gan," *Advances in neural information processing systems*, vol. 32, 2019.

[4] Z. Zhao, A. Kunar, R. Birke, H. Van der Scheer, and L. Y. Chen, "Ctab-gan+: Enhancing tabular data synthesis," *Frontiers in big Data*, vol. 6, p. 1296508, 2024.

[5] I. H. Sarker, "Machine learning: Algorithms, real-world applications and research directions," *SN computer science*, vol. 2, no. 3, p. 160, 2021.

[6] M. Alloghani, D. Al-Jumeily, J. Mustafina, A. Hussain, and A. J. Aljaaf, "A systematic review on supervised and unsupervised machine learning algorithms for data science," *Supervised and unsupervised learning for data science*, pp. 3–21, 2020.

[7] J. E. Van Engelen and H. H. Hoos, "A survey on semi-supervised learning," *Machine learning*, vol. 109, no. 2, pp. 373–440, 2020.

[8] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[9] B. Charbuty and A. Abdulazeez, "Classification based on decision tree algorithm for machine learning," *Journal of Applied Science and Technology Trends*, vol. 2, no. 01, pp. 20–28, 2021.

[10] Y. Lu, T. Ye, and J. Zheng, "Decision tree algorithm in machine learning," in *2022 IEEE International Conference on Advances in Electrical Engineering and Computer Applications (AEECA)*. IEEE, 2022, pp. 1014–1017.

[11] L. Breiman, *Classification and regression trees*. Routledge, 2017.

[12] M. M. Ghiasi, S. Zendehboudi, and A. A. Mohsenipour, "Decision tree-based diagnosis of coronary artery disease: Cart model," *Computer methods and programs in biomedicine*, vol. 192, p. 105400, 2020.

[13] T. Daniya, M. Geetha, and K. S. Kumar, "Classification and regression trees with gini index," *Advances in Mathematics: Scientific Journal*, vol. 9, no. 10, pp. 8237–8247, 2020.

[14] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, pp. 81–106, 1986.

[15] B. Hssina, A. Merbouha, H. Ezzikouri, and M. Erritali, "A comparative study of decision tree id3 and c4. 5," *International Journal of Advanced Computer Science and Applications*, vol. 4, no. 2, pp. 13–19, 2014.

[16] A. Cherfi, K. Nouira, and A. Ferchichi, "Very fast c4. 5 decision tree algorithm," *Applied Artificial Intelligence*, vol. 32, no. 2, pp. 119–137, 2018.

[17] A. Navada, A. N. Ansari, S. Patil, and B. A. Sonkamble, "Overview of use of decision tree algorithms in machine learning," in *2011 IEEE control and system graduate research colloquium*. IEEE, 2011, pp. 37–42.

[18] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[19] Y. Manzali and M. Elfar, "Random forest pruning techniques: a recent review," in *Operations research forum*, vol. 4, no. 2. Springer, 2023, p. 43.

[20] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, pp. 123–140, 1996.

[21] Y. Amit and D. Geman, "Shape quantization and recognition with randomized trees," *Neural computation*, vol. 9, no. 7, pp. 1545–1588, 1997.

[22] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to platt's smo algorithm for svm classifier design," *Neural computation*, vol. 13, no. 3, pp. 637–649, 2001.

[23] J. Cervantes, F. Garcia-Lamont, L. Rodríguez-Mazahua, and A. Lopez, "A comprehensive survey on support vector machine classification: Applications, challenges and trends," *Neurocomputing*, vol. 408, pp. 189–215, 2020.

[24] C. Cortes, "Support-vector networks," *Machine Learning*, 1995.

[25] L. Breiman, "Arcing classifier (with discussion and a rejoinder by the author)," *The annals of statistics*, vol. 26, no. 3, pp. 801–849, 1998.

[26] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[27] I. D. Mienye and Y. Sun, "A survey of ensemble learning: Concepts, algorithms, applications, and prospects," *IEEE Access*, vol. 10, pp. 99 129–99 149, 2022.

[28] M. Al-Kasassbeh, M. A. Abbadi, and A. M. Al-Bustanji, "Lightgbm algorithm for malware detection," in *Intelligent Computing: Proceedings of the 2020 Computing Conference, Volume 3*. Springer, 2020, pp. 391–403.

[29] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Machine learning*, vol. 6, pp. 37–66, 1991.

[30] N. Bhatia *et al.*, "Survey of nearest neighbor techniques," *arXiv preprint arXiv:1007.0085*, 2010.

[31] Z. Zhang, "Introduction to machine learning: k-nearest neighbors," *Annals of translational medicine*, vol. 4, no. 11, 2016.

[32] H. Peng, Z. Xu, W. Mo, Y. Wang, and Q. Huang, "Survey on knn," in *CAIBDA 2022; 2nd International Conference on Artificial Intelligence, Big Data and Algorithms*. VDE, 2022, pp. 1–7.

[33] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," *arXiv preprint arXiv:1302.4964*, 2013.

[34] P. B. Pajila, B. G. Sheena, A. Gayathri, J. Aswini, M. Nalini *et al.*, "A comprehensive survey on naive bayes algorithm: Advantages, limitations and applications," in *2023 4th International Conference on Smart Electronics and Communication (ICOSEC)*. IEEE, 2023, pp. 1228–1234.

[35] S. Chen, G. I. Webb, L. Liu, and X. Ma, "A novel selective naïve bayes algorithm," *Knowledge-Based Systems*, vol. 192, p. 105361, 2020.

[36] J. Han, J. Pei, and H. Tong, *Data mining: concepts and techniques*. Morgan kaufmann, 2022.

[37] T. M. Hope, "Linear regression," in *Machine learning*. Elsevier, 2020, pp. 67–81.

[38] D. Maulud and A. M. Abdulazeez, "A review on linear regression comprehensive in machine learning," *Journal of Applied Science and Technology Trends*, vol. 1, no. 2, pp. 140–147, 2020.

[39] M. Tranmer and M. Elliot, "Multiple linear regression," *The Cathie Marsh Centre for Census and Survey Research (CCSR)*, vol. 5, no. 5, pp. 1–5, 2008.

[40] S. l. Cessie and J. V. Houwelingen, "Ridge estimators in logistic regression," *Journal of the Royal Statistical Society Series C: Applied Statistics*, vol. 41, no. 1, pp. 191–201, 1992.

[41] X. Zou, Y. Hu, Z. Tian, and K. Shen, "Logistic regression model optimization and case analysis," in *2019 IEEE 7th international conference on computer science and network technology (ICCSNT)*. IEEE, 2019, pp. 135–139.

[42] J. C. Stoltzfus, "Logistic regression: a brief primer," *Academic emergency medicine*, vol. 18, no. 10, pp. 1099–1104, 2011.

[43] T. G. Nick and K. M. Campbell, "Logistic regression," *Topics in biostatistics*, pp. 273–301, 2007.

[44] S. Sperandei, "Understanding logistic regression analysis," *Biochemia medica*, vol. 24, no. 1, pp. 12–18, 2014.

[45] I. H. Sarker, M. H. Furhad, and R. Nowrozy, "Ai-driven cybersecurity: an overview, security intelligence modeling and research directions," *SN Computer Science*, vol. 2, no. 3, p. 173, 2021.

[46] H. Taud and J.-F. Mas, "Multilayer perceptron (mlp)," *Geomatic approaches for modeling land change scenarios*, pp. 451–455, 2018.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[48] I. H. Sarker, "Deep cybersecurity: a comprehensive overview from neural network and deep learning perspective," *SN Computer Science*, vol. 2, no. 3, p. 154, 2021.

[49] S. Watanabe, "Tree-structured parzen estimator: Understanding its algorithm components and their roles for better empirical performance," *arXiv preprint arXiv:2304.11127*, 2023.

[50] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[51] S. Cong and Y. Zhou, "A review of convolutional neural network architectures and their optimizations," *Artificial Intelligence Review*, vol. 56, no. 3, pp. 1905–1969, 2023.

[52] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[53] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole exe," in *Workshops at the thirty-second AAAI conference on artificial intelligence*, 2018.

[54] D. P. Mandic and J. Chambers, *Recurrent neural networks for prediction: learning algorithms, architectures and stability*. John Wiley & Sons, Inc., 2001.

[55] Z. C. Lipton, "A critical review of recurrent neural networks for sequence learning," *arXiv Preprint, CoRR, abs/1506.00019*, 2015.

[56] S. Hochreiter, "Long short-term memory," *Neural Computation MIT-Press*, 1997.

[57] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[58] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *IEEE signal processing magazine*, vol. 35, no. 1, pp. 53–65, 2018.

[59] Y. Lu, M. Shen, H. Wang, X. Wang, C. van Rechem, T. Fu, and W. Wei, "Machine learning for synthetic data generation: a review," *arXiv preprint arXiv:2302.04062*, 2023.

[60] I. Goodfellow, "Deep learning," 2016.

[61] S. Chen and W. Guo, "Auto-encoders in deep learning—a review with new perspectives," *Mathematics*, vol. 11, no. 8, p. 1777, 2023.

[62] Y. Wang, H. Yao, and S. Zhao, "Auto-encoder based dimensionality reduction," *Neurocomputing*, vol. 184, pp. 232–242, 2016.

[63] S. Lundberg, "A unified approach to interpreting model predictions," *arXiv preprint arXiv:1705.07874*, 2017.

[64] H. Baniecki, W. Kretowicz, P. PiÄ, J. WiĹ *et al.*, "Dalex: responsible machine learning with interactive explainability and fairness in python," *Journal of Machine Learning Research*, vol. 22, no. 214, pp. 1–7, 2021.

[65] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *International conference on machine learning*. PMLR, 2017, pp. 3319–3328.

[66] R. Calegari, G. Ciatto, J. Dellaluce, A. Omicini *et al.*, "Interpretable narrative explanation for ml predictors with lp: A case study for xai," in *CEUR WORKSHOP PROCEEDINGS*, vol. 2404. Sun SITE Central Europe, RWTH Aachen University, 2019, pp. 105–112.

[67] T. Miller, "Explanation in artificial intelligence: Insights from the social sciences," *Artificial intelligence*, vol. 267, pp. 1–38, 2019.

[68] A. Adadi and M. Berrada, "Peeking inside the black-box: a survey on explainable artificial intelligence (xai)," *IEEE access*, vol. 6, pp. 52 138–52 160, 2018.

[69] P. Linardatos, V. Papastefanopoulos, and S. Kotsiantis, "Explainable ai: A review of machine learning interpretability methods," *Entropy*, vol. 23, no. 1, p. 18, 2020.

[70] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[71] D. W. Apley and J. Zhu, "Visualizing the effects of predictor variables in black box supervised learning models," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 82, no. 4, pp. 1059–1086, 2020.

[72] M. Nagahisarchoghaei, N. Nur, L. Cummins, N. Nur, M. M. Karimi, S. Nandanwar, S. Bhattacharyya, and S. Rahimi, "An empirical survey on explainable ai technologies: Recent trends, use-cases, and categories from technical and application perspectives," *Electronics*, vol. 12, no. 5, p. 1092, 2023.

[73] M. T. Ribeiro, S. Singh, and C. Guestrin, "Model-agnostic interpretability of machine learning," *arXiv preprint arXiv:1606.05386*, 2016.

[74] A. Holzinger, A. Saranti, C. Molnar, P. Biecek, and W. Samek, "Explainable ai methods-a brief overview," in *International workshop on extending explainable AI beyond deep models and classifiers*. Springer, 2022, pp. 13–38.

[75] M. T. Ribeiro, S. Singh, and C. Guestrin, "" why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144.

[76] S. Wachter, B. Mittelstadt, and C. Russell, "Counterfactual explanations without opening the black box: Automated decisions and the gdpr," *Harv. JL & Tech.*, vol. 31, p. 841, 2017.

[77] T. Clement, N. Kemmerzell, M. Abdelaal, and M. Amberg, "Xair: A systematic metareview of explainable ai (xai) aligned to the software development process," *Machine Learning and Knowledge Extraction*, vol. 5, no. 1, pp. 78–108, 2023.

[78] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*, 2014.

[79] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation," *PloS one*, vol. 10, no. 7, p. e0130140, 2015.

[80] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

[81] H. Manthena, J. C. Kimmel, M. Abdelsalam, and M. Gupta, "Analyzing and explaining black-box models for online malware detection," *IEEE Access*, vol. 11, pp. 25 237–25 252, 2023.

[82] M. M. Alani, A. Mashatan, and A. Miri, "Xmal: A lightweight memory-based explainable obfuscated-malware detector," *Computers & Security*, vol. 133, p. 103409, 2023.

[83] L. S. Shapley, "A value for n-person games," *Contribution to the Theory of Games*, vol. 2, 1953.

[84] P. Biecek, "Dalex: Explainers for complex predictive models in r," *Journal of Machine Learning Research*, vol. 19, no. 84, pp. 1–5, 2018.

[85] Y. Swathi and M. Challa, "A comparative analysis of explainable ai techniques for enhanced model interpretability," in *2023 3rd International Conference on Pervasive Computing and Social Networking (ICPCSN)*. IEEE, 2023, pp. 229–234.

[86] M. AL-Essa, G. Andresini, A. Appice, and D. Malerba, "Striving for simplicity in deep neural models trained for malware detection," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2023, pp. 529–540.

[87] S. Mahdavifar, D. Alhadidi, and A. A. Ghorbani, "Effective and efficient hybrid android malware classification using pseudo-label stacked auto-encoder," *Journal of network and systems management*, vol. 30, no. 1, p. 22, 2022.

[88] L. De Rose, G. Andresini, A. Appice, and D. Malerba, "Vincent: Cyber-threat detection through vision transformers and knowledge distillation," *Computers & Security*, vol. 144, p. 103926, 2024.

[89] G. Liu, C. Yang, S. Liu, C. Xiao, and B. Song, "Feature selection method based on mutual information and support vector machine," *International journal of pattern recognition and artificial intelligence*, vol. 35, no. 06, p. 2150021, 2021.

[90] M. Al-Essa, G. Andresini, A. Appice, and D. Malerba, "Panacea: a neural model ensemble for cyber-threat detection," *Machine Learning*, vol. 113, no. 8, pp. 5379–5422, 2024.

[91] L. Dhanabal and S. Shantharajah, "A study on nsl-kdd dataset for intrusion detection system based on classification algorithms," *International journal of advanced research in computer and communication engineering*, vol. 4, no. 6, pp. 446–452, 2015.

[92] N. Moustafa and J. Slay, "Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set)," in *2015 military communications and information systems conference (MilCIS)*. IEEE, 2015, pp. 1–6.

[93] A. K. Phulre, M. Verma, J. P. S. Mathur, and S. Jain, "Approach on machine learning techniques for anomaly-based web intrusion detection systems: Using cicids2017 dataset," in *International Conference on MAchine inTelligence for Research & Innovations*. Springer, 2023, pp. 59–72.

[94] I. E. Nielsen, D. Dera, G. Rasool, R. P. Ramachandran, and N. C. Bouaynaya, "Robust explainability: A tutorial on gradient-based attribution methods for deep neural networks," *IEEE Signal Processing Magazine*, vol. 39, no. 4, pp. 73–84, 2022.

[95] M. Ancona, E. Ceolini, C. Öztireli, and M. Gross, "Gradient-based attribution methods," *Explainable AI: Interpreting, explaining and visualizing deep learning*, pp. 169–191, 2019.

[96] M. Melis, M. Scalas, A. Demontis, D. Maiorca, B. Biggio, G. Giacinto, and F. Roli, "Do gradient-based explanations tell anything about adversarial robustness to android malware?" *International journal of machine learning and cybernetics*, pp. 1–16, 2022.

[97] H. Liang, E. He, Y. Zhao, Z. Jia, and H. Li, "Adversarial attack and defense: A survey," *Electronics*, vol. 11, no. 8, p. 1283, 2022.

[98] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli, "Evasion attacks against machine learning at test time," in *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML*

PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part III 13. Springer, 2013, pp. 387–402.

[99] C. Szegedy, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[100] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples," *arXiv preprint arXiv:1605.07277*, 2016.

[101] S. Asha and P. Vinod, "Evaluation of adversarial machine learning tools for securing ai systems," *Cluster Computing*, vol. 25, no. 1, pp. 503–522, 2022.

[102] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," *arXiv preprint arXiv:1206.6389*, 2012.

[103] A. E. Cinà, K. Grosse, A. Demontis, S. Vascon, W. Zellinger, B. A. Moser, A. Oprea, B. Biggio, M. Pelillo, and F. Roli, "Wild patterns reloaded: A survey of machine learning security against training data poisoning," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–39, 2023.

[104] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *Artificial intelligence safety and security*. Chapman and Hall/CRC, 2018, pp. 99–112.

[105] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *stat*, vol. 1050, no. 9, 2017.

[106] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2574–2582.

[107] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *2016 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 2016, pp. 372–387.

[108] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh, "Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models," in *Proceedings of the 10th ACM workshop on artificial intelligence and security*, 2017, pp. 15–26.

[109] J. Chen, M. I. Jordan, and M. J. Wainwright, "Hopskipjumpattack: A query-efficient decision-based attack," in *2020 ieee symposium on security and privacy (sp)*. IEEE, 2020, pp. 1277–1294.

[110] M. Cheng, S. Singh, P. Chen, P.-Y. Chen, S. Liu, and C.-J. Hsieh, "Sign-opt: A query-efficient hard-label adversarial attack," *arXiv preprint arXiv:1909.10773*, 2019.

[111] K. Aryal, M. Gupta, M. Abdelsalam, and M. Saleh, "Intra-section code cave injection for adversarial evasion attacks on windows pe malware file," *arXiv preprint arXiv:2403.06428*, 2024.

[112] J. Lin, L. L. Njilla, and K. Xiong, "Secure machine learning against adversarial samples at test time," *EURASIP Journal on Information Security*, vol. 2022, no. 1, p. 1, 2022.

[113] Q. Card, K. Aryal, and M. Gupta, "Explainability-informed targeted malware misclassification," in *2024 33rd International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2024, pp. 1–8.

[114] D. S. Keyes, B. Li, G. Kaur, A. H. Lashkari, F. Gagnon, and F. Massicotte, "Entroplyzer: Android malware classification and characterization using entropy analysis of dynamic characteristics," in *2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)*. IEEE, 2021, pp. 1–12.

[115] Z. Wang, "Deep learning-based intrusion detection with adversaries," *IEEE Access*, vol. 6, pp. 38 367–38 384, 2018.

[116] E. Nowroozi, M. Mohammadi, M. Conti *et al.*, "An adversarial attack analysis on malicious advertisement url detection framework," *IEEE Transactions on Network and Service Management*, vol. 20, no. 2, pp. 1332–1344, 2022.

[117] C. Wang, L. Zhang, K. Zhao, X. Ding, and X. Wang, "Advandmal: Adversarial training for android malware detection and family classification," *Symmetry*, vol. 13, no. 6, p. 1081, 2021.

[118] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

[119] Y. Zhang, H. Li, Y. Zheng, S. Yao, and J. Jiang, "Enhanced dnns for malware classification with gan-based adversarial training," *Journal of Computer Virology and Hacking Techniques*, vol. 17, pp. 153–163, 2021.

[120] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "Ensemble adversarial training: Attacks and defenses," *arXiv preprint arXiv:1705.07204*, 2017.

[121] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 582–597.

[122] C. Xie, J. Wang, Z. Zhang, Z. Ren, and A. Yuille, "Mitigating adversarial effects through randomization," *arXiv preprint arXiv:1711.01991*, 2017.

[123] X. Liu, M. Cheng, H. Zhang, and C.-J. Hsieh, "Towards robust neural networks via random self-ensemble," in *Proceedings of the european conference on computer vision (ECCV)*, 2018, pp. 369–385.

[124] M. Lecuyer, V. Atlidakis, R. Geambasu, D. Hsu, and S. Jana, "Certified robustness to adversarial examples with differential privacy," in *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019, pp. 656–672.

[125] W. Xu, "Feature squeezing: Detecting adversarial exa mples in deep neural networks," *arXiv preprint arXiv:1704.01155*, 2017.

[126] A. Ross and F. Doshi-Velez, "Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[127] X. Ling, L. Wu, J. Zhang, Z. Qu, W. Deng, X. Chen, Y. Qian, C. Wu, S. Ji, T. Luo *et al.*, "Adversarial attacks against windows pe malware detection: A survey of the state-of-the-art," *Computers & Security*, vol. 128, p. 103134, 2023.

[128] J. Singh and J. Singh, "A survey on machine learning-based malware detection in executable files," *Journal of Systems Architecture*, vol. 112, p. 101861, 2021.

[129] S. J. Kattamuri, R. K. V. Penmatsa, S. Chakravarty, and V. S. P. Madabathula, "Swarm optimization and machine learning applied to pe malware detection towards cyber threat intelligence," *Electronics*, vol. 12, no. 2, p. 342, 2023.

[130] Ö. A. Aslan and R. Samet, "A comprehensive review on malware detection approaches," *IEEE access*, vol. 8, pp. 6249–6271, 2020.

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

[132] S. Abimannan and R. Kumaravelu, "A mathematical model of hmst model on malware static analysis," *International Journal of Information Security and Privacy (IJISP)*, vol. 13, no. 2, pp. 86–103, 2019.

[133] S. Schrittwieser and S. Katzenbeisser, "Code obfuscation against static and dynamic reverse engineering," in *Information Hiding: 13th International Confer-*

ence, IH 2011, Prague, Czech Republic, May 18-20, 2011, Revised Selected Papers 13. Springer, 2011, pp. 270–284.

[134] A. Mohanta, A. Saldanha, A. Mohanta, and A. Saldanha, "Static analysis," *Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware*, pp. 377–402, 2020.

[135] E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," *Journal of Information Security*, vol. 2014, 2014.

[136] S. M. Bidoki, S. Jalili, and A. Tajoddin, "Pbmmd: A novel policy based multi-process malware detection," *Engineering Applications of Artificial Intelligence*, vol. 60, pp. 57–70, 2017.

[137] E. Eilam, *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

[138] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges," *Journal of Network and Computer Applications*, vol. 153, p. 102526, 2020.

[139] V. Ndatinya, Z. Xiao, V. R. Manepalli, K. Meng, and Y. Xiao, "Network forensics analysis using wireshark," *International Journal of Security and Networks*, vol. 10, no. 2, pp. 91–106, 2015.

[140] C. Rathnayaka and A. Jamdagni, "An efficient approach for advanced malware analysis using memory forensic technique," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 1145–1150.

[141] I. Kara, "A basic malware analysis method," *Computer Fraud & Security*, vol. 2019, no. 6, pp. 11–19, 2019.

[142] Q. K. A. Mirza, I. Awan, and M. Younas, "Cloudintell: An intelligent malware detection system," *Future Generation Computer Systems*, vol. 86, pp. 1042–1053, 2018.

[143] A. Mohanta, A. Saldanha, A. Mohanta, and A. Saldanha, "Dynamic analysis," *Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware*, pp. 403–431, 2020.

[144] T. Alsmadi and N. Alqudah, "A survey on malware detection techniques," in *2021 international conference on information technology (ICIT)*. IEEE, 2021, pp. 371–376.

[145] Ö. Aslan and R. Samet, "Investigation of possibilities to detect malware using existing tools," in *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*. IEEE, 2017, pp. 1277–1284.

[146] J. Singh and J. Singh, "Detection of malicious software by analyzing the behavioral artifacts using machine learning algorithms," *Information and Software Technology*, vol. 121, p. 106273, 2020.

[147] N. A. Azeez, O. E. Odufuwa, S. Misra, J. Oluranti, and R. Damaševičius, "Windows pe malware detection using ensemble learning," in *Informatics*, vol. 8, no. 1. MDPI, 2021, p. 10.

[148] M. I. Yousuf, I. Anwer, A. Riasat, K. T. Zia, and S. Kim, "Windows malware detection based on static analysis with multiple features," *PeerJ Computer Science*, vol. 9, p. e1319, 2023.

[149] M. S. Nawaz, P. Fournier-Viger, M. Z. Nawaz, G. Chen, and Y. Wu, "Malspm: Metamorphic malware behavior analysis and classification using sequential pattern mining," *Computers & Security*, vol. 118, p. 102741, 2022.

[150] A. G. Kakisim, S. Gulmez, and I. Sogukpinar, "Sequential opcode embedding-based malware detection method," *Computers & Electrical Engineering*, vol. 98, p. 107703, 2022.

[151] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[152] A. Kamboj, P. Kumar, A. K. Bairwa, and S. Joshi, "Detection of malware in downloaded files using various machine learning models," *Egyptian Informatics Journal*, vol. 24, no. 1, pp. 81–94, 2023.

[153] P. P. Kundu, L. Anatharaman, and T. Truong-Huu, "An empirical evaluation of automated machine learning techniques for malware detection," in *Proceedings of the 2021 ACM Workshop on Security and Privacy Analytics*, 2021, pp. 75–81.

[154] J. Suaboot, Z. Tari, A. Mahmood, A. Y. Zomaya, and W. Li, "Sub-curve hmm: A malware detection approach based on partial analysis of api call sequences," *Computers & Security*, vol. 92, p. 101773, 2020.

[155] L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state markov chains," *The annals of mathematical statistics*, vol. 37, no. 6, pp. 1554–1563, 1966.

[156] S. Euh, H. Lee, D. Kim, and D. Hwang, "Comparative analysis of low-dimensional features and tree-based ensembles for malware detection systems," *IEEE Access*, vol. 8, pp. 76 796–76 808, 2020.

[157] E. Amer and I. Zelinka, "A dynamic windows malware detection and prediction method based on contextual understanding of api call sequence," *Computers & Security*, vol. 92, p. 101760, 2020.

[158] N. Ketkar and E. Santana, *Deep learning with Python*. Springer, 2017, vol. 1.

[159] B. Andrews, R. A. Davis, and F. J. Breidt, "Maximum likelihood estimation for all-pass time series models," *Journal of Multivariate Analysis*, vol. 97, no. 7, pp. 1638–1659, 2006.

[160] D. Rabadi and S. G. Teo, "Advanced windows methods on malware detection and classification," in *Proceedings of the 36th Annual Computer Security Applications Conference*, 2020, pp. 54–68.

[161] R. Sihwail, K. Omar, K. A. Zainol Ariffin, and S. Al Afghani, "Malware detection approach based on artifacts in memory image and dynamic analysis," *Applied Sciences*, vol. 9, no. 18, p. 3680, 2019.

[162] A. Walker and S. Sengupta, "Insights into malware detection via behavioral frequency analysis using machine learning," in *MILCOM 2019-2019 IEEE Military Communications Conference (MILCOM)*. IEEE, 2019, pp. 1–6.

[163] W. Han, J. Xue, Y. Wang, L. Huang, Z. Kong, and L. Mao, "Maldae: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics," *computers & security*, vol. 83, pp. 208–233, 2019.

[164] A. Mallik, A. Khetarpal, and S. Kumar, "Conrec: malware classification using convolutional recurrence," *Journal of Computer Virology and Hacking Techniques*, vol. 18, no. 4, pp. 297–313, 2022.

[165] S. K. J. Rizvi, W. Aslam, M. Shahzad, S. Saleem, and M. M. Fraz, "Proud-mal: static analysis-based progressive framework for deep unsupervised malware classification of windows portable executable," *Complex & Intelligent Systems*, pp. 1–13, 2022.

[166] X. Chen, Z. Hao, L. Li, L. Cui, Y. Zhu, Z. Ding, and Y. Liu, "Cruparamer: Learning on parameter-augmented api sequences for malware detection," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 788–803, 2022.

[167] B. Zou, C. Cao, F. Tao, and L. Wang, "Imclnet: A lightweight deep neural network for image-based malware classification," *Journal of Information Security and Applications*, vol. 70, p. 103313, 2022.

[168] C. Yuan, J. Cai, D. Tian, R. Ma, X. Jia, and W. Liu, "Towards time evolved malware identification using two-head neural network," *Journal of information security and applications*, vol. 65, p. 103098, 2022.

[169] M. N. Al-Andoli, S. C. Tan, K. S. Sim, C. P. Lim, and P. Y. Goh, "Parallel deep learning with a hybrid bp-pso framework for feature extraction and malware classification," *Applied Soft Computing*, vol. 131, p. 109756, 2022.

[170] V. Kelefouras, A. Kritikakou, and C. Goutis, "A matrix–matrix multiplication methodology for single/multi-core architectures using simd," *The Journal of Supercomputing*, vol. 68, pp. 1418–1440, 2014.

[171] D. Gibert, J. Planes, C. Mateu, and Q. Le, "Fusing feature engineering and deep learning: A case study for malware classification," *Expert Systems with Applications*, vol. 207, p. 117957, 2022.

[172] C. Li, Z. Cheng, H. Zhu, L. Wang, Q. Lv, Y. Wang, N. Li, and D. Sun, "Dmalnet: Dynamic malware analysis based on api feature engineering and graph learning," *Computers & Security*, vol. 122, p. 102872, 2022.

[173] W. Hu, B. Liu, J. Gomes, M. Zitnik, P. Liang, V. Pande, and J. Leskovec, "Strategies for pre-training graph neural networks," *arXiv preprint arXiv:1905.12265*, 2019.

[174] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio *et al.*, "Graph attention networks," *stat*, vol. 1050, no. 20, pp. 10–48 550, 2017.

[175] H. Gao and S. Ji, "Graph u-nets," in *international conference on machine learning*. PMLR, 2019, pp. 2083–2092.

[176] A. Tekerek and M. M. Yapici, "A novel malware classification and augmentation model based on convolutional neural network," *Computers & Security*, vol. 112, p. 102515, 2022.

[177] V. Ravi, M. Alazab, S. Selvaganapathy, and R. Chaganti, "A multi-view attention-based deep learning framework for malware detection in smart healthcare systems," *Computer Communications*, vol. 195, pp. 73–81, 2022.

[178] D. Demırcı, C. Acarturk *et al.*, "Static malware detection using stacked bilstm and gpt-2," *IEEE Access*, vol. 10, pp. 58 488–58 502, 2022.

[179] T. Rezaei, F. Manavi, and A. Hamzeh, "A pe header-based method for malware detection using clustering and deep embedding techniques," *Journal of Information Security and Applications*, vol. 60, p. 102876, 2021.

[180] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Malware detection by control-flow graph level representation learning with graph isomorphism network," *IEEE Access*, vol. 10, pp. 111 830–111 841, 2022.

[181] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," *Advances in Neural Information Processing Systems*, vol. 33, pp. 5776–5788, 2020.

[182] D. Li and Q. Li, "Adversarial deep ensemble: Evasion attacks and defenses for malware detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3886–3900, 2020.

[183] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," in *2018 26th European signal processing conference (EUSIPCO)*. IEEE, 2018, pp. 533–537.

[184] S. E. Coull and C. Gardner, "Activation analysis of a byte-based deep neural network for malware classification," in *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2019, pp. 21–27.

[185] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, "Adversarial examples on discrete sequences for beating whole-binary malware detection," *arXiv preprint arXiv:1802.04528*, pp. 490–510, 2018.

[186] Y. Qiao, W. Zhang, Z. Tian, L. T. Yang, Y. Liu, and M. Alazab, "Adversarial malware sample generation method based on the prototype of deep learning detector," *Computers & Security*, vol. 119, p. 102762, 2022.

[187] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," *arXiv preprint arXiv:1506.06579*, 2015.

[188] K. Li, W. Guo, F. Zhang, and J. Du, "Gambd: Generating adversarial malware against malconv," *Computers & Security*, vol. 130, p. 103279, 2023.

[189] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li, "Boosting adversarial attacks with momentum," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 9185–9193.

[190] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Functionality-preserving black-box optimization of adversarial windows malware," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3469–3478, 2021.

[191] H. S. Anderson and P. Roth, "Ember: an open dataset for training static pe malware machine learning models," *arXiv preprint arXiv:1804.04637*, 2018.

[192] R. Harang and E. M. Rudd, "Sorel-20m: A large scale benchmark dataset for malicious pe detection," *arXiv preprint arXiv:2012.07634*, 2020.

[193] L. Yang, A. Ciptadi, I. Laziuk, A. Ahmadzadeh, and G. Wang, "Bodmas: An open dataset for learning based temporal analysis of pe malware," in *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2021, pp. 78–84.

[194] F. O. Catak, A. F. Yazı, O. Elezaj, and J. Ahmed, "Deep learning based sequential model for malware analysis using windows exe api calls," *PeerJ computer science*, vol. 6, p. e285, 2020.

[195] B. Bosansky, D. Kouba, O. Manhal, T. Sick, V. Lisy, J. Kroustek, and P. Somol, "Avast-ctu public cape dataset," *arXiv preprint arXiv:2209.03188*, 2022.

[196] L. Demetrio and B. Biggio, "secml-malware: A python library for adversarial robustness evaluation of windows malware classifiers. arxiv," *Cryptography and Security (cs. CR)*, 2021.

[197] Z. Tian, L. Cui, J. Liang, and S. Yu, "A comprehensive survey on poisoning attacks and countermeasures in machine learning," *ACM Computing Surveys*, vol. 55, no. 8, pp. 1–35, 2022.

[198] A. Mohanta, A. Saldanha, A. Mohanta, and A. Saldanha, "Armoring and evasion: The anti-techniques," *Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware*, pp. 691–720, 2020.

[199] A. Guerra-Manzanares, "Machine learning for android malware detection: mission accomplished? a comprehensive review of open challenges and future perspectives," *Computers & Security*, vol. 138, p. 103654, 2024.

[200] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, "Intriguing properties of adversarial ml attacks in the problem space," in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 1332–1349.

[201] S. Chen, M. Xue, L. Fan, S. Hao, L. Xu, H. Zhu, and B. Li, "Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach," *computers & security*, vol. 73, pp. 326–344, 2018.

[202] D. B. Rubin, "Statistical disclosure limitation," *Journal of official Statistics*, vol. 9, no. 2, pp. 461–468, 1993.

[203] S. A. Assefa, D. Dervovic, M. Mahfouz, R. E. Tillman, P. Reddy, and M. Veloso, "Generating synthetic data in finance: opportunities, challenges and pitfalls," in

Proceedings of the First ACM International Conference on AI in Finance, 2020, pp. 1–8.

[204] J. Yoon, L. N. Drumright, and M. Van Der Schaar, "Anonymization through data synthesis using generative adversarial networks (ads-gan)," *IEEE journal of biomedical and health informatics*, vol. 24, no. 8, pp. 2378–2388, 2020.

[205] D. A. Van Dyk and X.-L. Meng, "The art of data augmentation," *Journal of Computational and Graphical Statistics*, vol. 10, no. 1, pp. 1–50, 2001.

[206] S. C. Wong, A. Gatt, V. Stamatescu, and M. D. McDonnell, "Understanding data augmentation for classification: when to warp?" in *2016 international conference on digital image computing: techniques and applications (DICTA)*. IEEE, 2016, pp. 1–6.

[207] M. Al-Essa and A. Appice, "Dealing with imbalanced data in multi-class network intrusion detection systems using xgboost," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2021, pp. 5–21.

[208] E. Choi, S. Biswal, B. Malin, J. Duke, W. F. Stewart, and J. Sun, "Generating multi-label discrete patient records using generative adversarial networks," in *Machine learning for healthcare conference*. PMLR, 2017, pp. 286–305.

[209] N. Park, M. Mohammadi, K. Gorde, S. Jajodia, H. Park, and Y. Kim, "Data synthesis based on generative adversarial networks," *arXiv preprint arXiv:1806.03384*, 2018.

[210] J. Lee, J. Hyeong, J. Jeon, N. Park, and J. Cho, "Invertible tabular gans: Killing two birds with one stone for tabular data synthesis," *Advances in Neural Information Processing Systems*, vol. 34, pp. 4263–4273, 2021.

[211] J. Engelmann and S. Lessmann, "Conditional wasserstein gan-based oversampling of tabular data for imbalanced learning," *Expert Systems with Applications*, vol. 174, p. 114582, 2021.

[212] J. Jordon, J. Yoon, and M. Van Der Schaar, "Pate-gan: Generating synthetic data with differential privacy guarantees," in *International conference on learning representations*, 2018.

[213] L. Xie, K. Lin, S. Wang, F. Wang, and J. Zhou, "Differentially private generative adversarial network," *arXiv preprint arXiv:1802.06739*, 2018.

[214] A. Torfi, E. A. Fox, and C. K. Reddy, "Differentially private synthetic medical data generation using convolutional gans," *Information Sciences*, vol. 586, pp. 485–500, 2022.

[215] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4, no. 4.

[216] Z. Lin, A. Khetan, G. Fanti, and S. Oh, "Pacgan: The power of two samples in generative adversarial networks," *Advances in neural information processing systems*, vol. 31, 2018.

[217] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," *Advances in neural information processing systems*, vol. 31, 2018.

[218] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," *Advances in neural information processing systems*, vol. 30, 2017.

[219] I. Mironov, "Rényi differential privacy," in *2017 IEEE 30th computer security foundations symposium (CSF)*. IEEE, 2017, pp. 263–275.

[220] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 308–318.

[221] T.-Y. Ross and G. Dollár, "Focal loss for dense object detection," in *proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2980–2988.

[222] T. Carrier, P. Victor, A. Tekeoglu, and A. H. Lashkari, "Detecting obfuscated malware using memory feature engineering." in *Icissp*, 2022, pp. 177–188.

[223] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 2623–2631.

[224] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *Journal of the american statistical association*, vol. 32, no. 200, pp. 675–701, 1937.

[225] ——, "A comparison of alternative tests of significance for the problem of m rankings," *The annals of mathematical statistics*, vol. 11, no. 1, pp. 86–92, 1940.

[226] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE transactions on software engineering*, vol. 34, no. 4, pp. 485–496, 2008.

Appendices

Notes

1. https://www.iolo.com/resources/articles/almost-1-million-new-malware-threats-are-rel
2. https://cybersecurityventures.com/cybercrime-damage-costs-10-trillion-by-2025/
3. {AccordingtothestatisticssharedbytheKasperksylabattheendof2021,WindowsPortableExecut filescomprisethemajorityofmalwarethreats,withmorethan90%ofdailymalwaredetected}.
4. https://www.virtualbox.org/wiki/Downloads
5. https://techdocs.broadcom.com/us/en/vmware-cis/desktop-hypervisors/workstation-pro/17-0/using-vmware-workstation-pro.html
6. https://github.com/lief-project/LIEF
7. https://www.ietf.org/rfc/rfc1321.txt
8. https://hashing.tools/sha-1
9. https://www.sciencedirect.com/topics/computer-science/principal-components
10. https://angr.io/
11. https://www.sbert.net/
12. https://networkx.org/
13. https://www.camlis.org/2017/jeffreyjohns
14. https://www.virustotal.com/gui/home/upload
15. https://virusshare.com/
16. https://malshare.com/index.php
17. https://github.com/sophos/SOREL-20M

18. http://dasmalwerk.eu/

19. https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/

20. https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/

21. https://archive.ics.uci.edu/dataset/848/secondary+mushroom+dataset

22. https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud

23. https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud

24. https://archive.ics.uci.edu/dataset/31/covertype

25. https://www.kaggle.com/datasets/meirnizri/covid19-dataset

26. https://www.kaggle.com/datasets/fedesoriano/stellar-classification-dataset-sdss17

27. https://research.unsw.edu.au/projects/unsw-nb15-dataset

28. https://www.kaggle.com/datasets/itsmesunil/bank-loan-modelling

29. https://archive.ics.uci.edu/dataset/2/adult

30. https://github.com/biolab/orange3

31. https://pandas.pydata.org/