Static Malware Detection through Ensemble Feature Selection and Supervised Classification

Isai Moreno-Lara, Alejandra Guadalupe Silva-Trujillo*, Juan C. Cuevas-Tello, Jose Nunez-Varela
Facultad de Ingeniería, Universidad Autónoma de San Luis Potosí, San Luis Potosí, Mexico
E-mail: isai.lara.dev@gmail.com, asilva@uaslp.mx, cuevas@uaslp.mx, jose.nunez@uaslp.mx
*Corresponding author

Keywords: Malware, classification, machine learning, cybersecurity, algorithms

Received: August 7, 2025

In a digital landscape where malicious software evolves faster than traditional defenses, intelligent and proactive detection has become essential. This study presents a machine learning framework for static malware detection based on the analysis of 138,047 Portable Executable samples, including both malware and benign files. The dataset comprises 56 static structural features extracted without code execution. Four supervised classifiers—Backpropagation Neural Network, Decision Tree, Random Forest, and Support Vector Machine—were evaluated following the Knowledge Discovery in Databases process. Ensemble-based feature selection methods (Random Forest and Extra Trees) were applied to identify the most informative attributes, while random undersampling was used to mitigate class imbalance. Experimental results show that the Random Forest classifier achieved the best performance, reaching 99.45% accuracy and a 0.9909 F1-score on imbalanced data, and 99.32% accuracy on the balanced dataset. These findings highlight the reliability and scalability of tree-based models for static malware detection. Overall, the proposed framework demonstrates that careful feature selection and balance adjustment can significantly enhance the performance and interpretability of cybersecurity classification systems.

Povzetek: Študija predstavi statično zaznavanje zlonamerne programske opreme iz 56 značilk PE datotek na velikem naboru vzorcev, kjer z izbiro značilk z ansambli in uravnoteženjem razredov pokaže, da drevesni modeli ponujajo zanesljivo in razložljivo detekcijo.

1 Introduction

In today's hyperconnected world, malware poses a persistent threat to individuals, organizations, and even critical infrastructure. Malware, a contraction of 'malicious software', encompasses any software developed with the intent to harm, exploit, or otherwise compromise computers, networks, or users. The U.S. National Institute of Standards and Technology (NIST) defines malware as a program inserted into a system with the aim of compromising the confidentiality, integrity, or availability of the system's data or operations [1]. This broad category includes viruses, worms, trojans, ransomware, spyware, and more sophisticated threats that constantly evolve to evade traditional security systems.

The increasing digitalization of services and the dependency on connected systems have increased the risks posed by malware attacks. These attacks can result in massive financial losses, data breaches, reputational damage, or even disruptions to critical infrastructure. Global statistics from cybersecurity firms show an upward trend in malware activity across sectors, with new variants being generated using automated tools and code obfuscation techniques, making manual detection approaches increasingly infeasible [2].

To address this growing threat, machine learning (ML) has emerged as a powerful tool for malware detection. Unlike signature-based methods that rely on predefined rules and known patterns, ML algorithms can learn from past data to identify new, previously unseen malware by recognizing subtle structural or behavioral indicators. This data-driven approach allows for improved generalization, early threat detection, and reduced reliance on expert-crafted features.

Malware detection techniques are broadly categorized into three types: **static analysis**, **dynamic analysis**, and **hybrid analysis** [3]. Static analysis involves examining the structure, code, and metadata of a program without executing it. Common static features include Application Programming Interface (API) call sequences, opcode frequency, string patterns, and control flow graphs elements that can be extracted safely and efficiently at scale. In contrast, dynamic analysis executes the malware in a sandbox or controlled environment to observe its runtime behavior, such as system calls, registry changes, or network activity. While dynamic techniques are more resilient to obfuscation and capable of detecting zero-day threats, they are time-consuming and resource-intensive, and can be bypassed by malware that detects virtualized environments. Finally, hybrid analysis combines static and dynamic methods to benefit from both perspectives. However, it inherits the limitations of both and often increases computational cost.

In this study, we have adopted a **static analysis** approach. This decision is driven by the nature of our dataset and practical constraints. Static analysis allows fast, large-scale inspection of software without the overhead or risk of execution. Since our dataset includes pre-extracted structural features from known samples, static analysis is both appropriate and scalable for building machine learning models.

Consequently, we reinforce the notion that the union of machine learning and cybersecurity, when guided by principled data science methodologies, has the potential to produce intelligent, adaptive, and ethically sound tools for modern malware detection. By demonstrating the effectiveness of well-calibrated models under different experimental conditions, this work contributes to the ongoing development of secure, data-driven defense mechanisms capable of facing the challenges of an increasingly hostile digital ecosystem. Moreover, this work investigates how class balance and feature selection strategies influence the performance and interpretability of supervised machine learning models for malware detection. These aspects are critical, as imbalanced datasets can bias classifiers toward the majority class, while inadequate feature selection may introduce redundancy or noise that reduces generalization. The study systematically evaluates these factors within the Knowledge Discovery in Databases (KDD) process, using both balanced and unbalanced datasets, and compares models through metrics such as precision, recall, F1-score and Area Under the Curve (AUC). This approach aims to identify the conditions that yield the most robust and explainable detection results.

This paper is structured as follows. Section 2 introduces the Knowledge Discovery in Databases process and explains its relevance in malware detection. The state of the art is presented in Section 3 to contextualize this study within existing research. Section 4 describes the dataset in detail, including the class distribution and a statistical analysis of the feature values. The preprocessing steps are then outlined in Section 5, with particular emphasis on the class balancing strategy through undersampling, which resulted in two versions of the dataset. Section 6 addresses the transformation stage, detailing the feature selection algorithms applied. The data mining phase is described in Section 7, including the classification models used and the configuration of their respective hyperparameters. Section 8 presents the results of the evaluation metrics and reports the performance of the models under both original and balanced datasets, followed by an interpretation of the outcomes. Finally, Section 10 concludes with a summary of the key findings and suggestions for future research.

2 Knowledge discovery in databases (KDD) Process

To build an effective malware detection system, we follow the KDD process — a systematic methodology for extracting meaningful patterns from large datasets. As defined by Fayyad et al. [4], KDD is "the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data". This methodology is especially important when the volume and complexity of data exceed human capacity for manual analysis.

In cybersecurity contexts such as malware detection, datasets are often high-dimensional and voluminous, making manual exploration of correlations and anomalies infeasible. KDD provides a structured framework that integrates not only data mining algorithms, but also essential steps such as data selection, preprocessing, transformation, and interpretation. Moreover, blindly applying data mining techniques — commonly referred to as *data dredging* — can lead to misleading conclusions. The KDD process mitigates this risk through iterative refinement, incorporation of prior knowledge, and user-guided evaluation. In our study, the KDD process structures the full pipeline, from analyzing static malware attributes and selecting the most informative features, to evaluating classification performance across balanced and imbalanced scenarios. This process includes the following stages:

- **Data Selection:** Involves choosing the subset of data relevant to the problem domain from potentially large and heterogeneous sources.
- **Preprocessing:** Selected data is cleaned and transformed to improve its quality and consistency. Tasks include removing noise, handling missing values, converting data formats, and normalizing features.
- **Feature Selection:** Reduces the number of input variables by selecting only the most relevant features for the task. It helps to improve model performance, reduce overfitting, and enhance interpretability by eliminating redundant or irrelevant attributes.
- **Feature Comparison:** Selected features are analyzed to understand their properties to identify correlations, redundancies, or informative patterns that may influence model outcomes.
- **Data Mining:** Algorithms are applied to extract patterns or models from data. Techniques may include classification, regression, clustering, or association rule mining, depending on the objective of the study.
- **Evaluation:** Assessment of the quality and usefulness of the knowledge discovered using performance metrics or validation techniques.

The core task in this study is formulated as a **binary classification problem**, where each software instance is labeled either as `malware` or `benign`. To solve this, we employ and compare the following machine learning algorithms: Backpropagation Neural Networks, Decision Trees, Random Forests, and Support Vector Machines. Each model is trained and evaluated using the same dataset and preprocessing pipeline. The overall workflow is illustrated in Figure 1, summarizing the steps from data acquisition to classification.

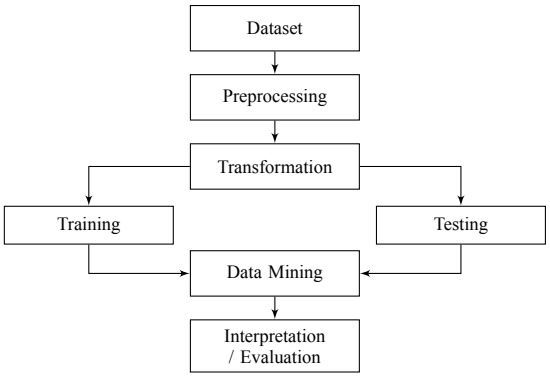


Figure 1: General workflow for malware detection using ML and the KDD process.

3 State of the art

Malware detection remains a core challenge in modern cybersecurity due to the rapid increase in novel attack variants and the evolving sophistication of evasion techniques. Traditional defenses struggle to keep pace with the speed and diversity of malware generation, especially in mobile and Internet of Things (IoT) ecosystems, where constraints on resources and real-time protection demand more intelligent and scalable approaches.

Recent research has focused on leveraging ML techniques to improve malware detection through both static and dynamic analysis. One of the most influential studies in this area is that of Shakib et al. [5], who introduced a benchmark dataset widely adopted for evaluating hybrid models that combine Genetic Artificial Intelligence, deep neural architectures, and Active Learning (AL). Their experiments achieved near-perfect accuracy levels above 99%, demonstrating the effectiveness of deep learning architectures for malware detection. However, their work primarily emphasized performance metrics such as accuracy and F1-score, without addressing potential class imbalance or feature redundancy issues that could affect generalization.

Building on this line of research, Tanuwidjaja and Kim [6] proposed an adaptation of the Deep Feature Extraction and Selection (DFES) method originally designed for intrusion detection. Their approach integrates Stacked Autoencoders (SAE) for deep abstraction, Weighted Feature Selection, and Artificial Neural Networks (ANN), resulting in a lightweight yet highly accurate pipeline. By focusing on dimensionality reduction and feature relevance, this study highlighted the importance of feature selection as a key factor in improving efficiency and performance in malware classification.

Complementing these works, Roy [7] investigated the detection of encrypted malware traffic, comparing Logistic Regression, Convolutional Neural Networks, and Random Forests. Despite the additional challenge of encryption, Random Forest achieved the highest accuracy (98.28%), reinforcing the robustness of tree-based models in constrained and obfuscated environments. Similarly, Baker del Aguila et al. [8] examined lightweight static analysis techniques under limited computational resources. Their findings showed that Artificial Neural Networks reached 94.74% accuracy, followed closely by Support Vector Machines and Gradient Boosting Machines, proving that traditional ML models can remain competitive in memory-restricted scenarios.

Overall, these studies illustrate the progressive refinement of malware detection methods, from complex hybrid architectures to efficient lightweight models. Nevertheless, most prior works—such as those by Shakib et al. [5], Tanuwidjaja and Kim [6] and Roy [7]—focused primarily on achieving high accuracy and F1-scores, while overlooking important aspects like dataset imbalance, feature redundancy, or model interpretability.

Addressing these gaps, the present study systematically evaluates the influence of class balance and feature selection strategies within the Knowledge Discovery in Databases process, aiming to provide more reliable and explainable results.

Table 1 summarizes the classification performance reported in recent malware detection studies. Each study employed different ML models and evaluation strategies, yet most relied primarily on accuracy as the main performance indicator. Although each study refers to the dataset differently, all of them were conducted using the same underlying dataset, which contains both benign and malicious program samples. To provide a more complete comparison, the table includes not only accuracy but also F1-score and loss values when available. Further details comparing these results with those obtained in the present study are discussed in Section 9.

4 Dataset

This study relies on a benchmark dataset comprising 138,047 labeled software samples, including 96,724 identified as malware and 41,323 as legitimate. Each instance is represented by 56 static features that encapsulate structural metadata and header-related attributes, extracted without executing the software binaries. This static approach enables scalable and reproducible analyses, particularly suitable for pre-execution detection pipelines. To ensure methodological consistency, the same dataset used in the studies listed in Table 1 was adopted in this work. This dataset was originally derived from *VirusShare* repository [9], which provides publicly available malware samples used for academic and security research.

Originally distributed in Comma-Separated Values (CSV) format — a widely accepted standard for structured data representation. For analytical processing, the dataset was loaded into a Pandas DataFrame [10], a powerful tabular data structure within the Python ecosystem that provides extensive functionality for data transformation, filtering, and statistical computation.

A preliminary structural inspection revealed that two

Table 1: Comparison of evaluation metrics achieved by different machine learning models in malware detection studies.

Ref.	Model / Approach	Accuracy (%)	F1-score	Loss
[5]	Convolutional Neural Network	99.94	0.9993	0.007
	Recurrent Neural Network	99.55	0.9954	0.026
	Long Short-Term Memory Network	77.66	0.8094	0.483
	Gated Recurrent Unit Network	81.05	0.8050	0.422
	Active Learning with Convolutional Neural Network	99.93	0.9961	0.006
	Active Learning with Recurrent Neural Network	99.89	0.9989	0.001
	Active Learning with Long Short-Term Memory Network	86.59	0.8673	0.325
	Active Learning with Gated Recurrent Unit Network	94.01	0.9402	0.175
	Genetic Artificial Intelligence with Convolutional Neural Network	95.93	0.9589	0.149
	Genetic Artificial Intelligence with Recurrent Neural Network	99.36	0.9935	0.055
	Genetic Artificial Intelligence with Long Short-Term Memory Network	99.27	0.9925	0.037
	Genetic Artificial Intelligence with Gated Recurrent Unit Network	99.77	0.9977	0.019
[6]	Original Deep Abstraction and Weighted Feature Selection with Stacked Autoencoder and Artificial Neural Network	99.917	0.9986	-
	Modified Deep Abstraction and Weighted Feature Selection with Stacked Autoencoder and Artificial Neural Network (reduced feature set)	99.974	0.9996	-
[7]	Random Forest	98.28	-	-
	Logistic Regression	70.15	-	-
	Convolutional Neural Network	97.00	-	-
[8]	Artificial Neural Network	94.74	0.94	0.1488
	Support Vector Machine	91.07	0.92	0.239
	Gradient Boosting Machine	92.47	0.91	0.04

features—`Name` and `md5`—served as unique identifiers and did not contribute meaningful semantic information for classification purposes. Therefore, these fields were excluded from subsequent analyses. After this refinement, the dataset contained a total of 55 columns: 54 informative features and one target class. All variables exhibited a consistent and well-defined schema, with 45 stored as 64-bit integers (`int64`) and 10 as 64-bit floating-point numbers (`float64`). Figure 2 illustrates the distribution of data types across the dataset after discarding the non-informative identifiers.

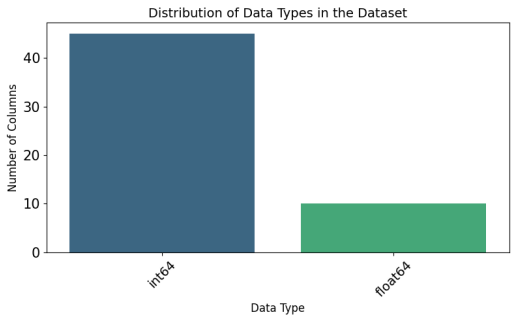


Figure 2: Distribution of data types in the dataset (excluding Name and md5)

To better understand the behavior and statistical profile of each feature across classes, descriptive metrics were computed independently for the malware and legitimate samples. These metrics include the minimum and maximum observed values, arithmetic mean, standard deviation, and interquartile range boundaries (Q1 and Q3). To enhance the readability and interpretability of the results, a formatting scheme was applied according to the magnitude of the values. Specifically, large numerical values were converted into more compact representations using metric suffixes (e.g., “K” for thousands, “M” for millions) or expressed in scientific notation for extremely large magnitudes. This transformation not only reduces visual clutter in the tables but also enables more efficient cross-feature comparisons across varying scales. Tables 6 and 7 (see Appendix) present these adapted statistics, highlighting relevant distributional characteristics that may differentiate benign and malicious instances. For example, features related to section entropy, import table sizes, or memory allocation regions often show marked discrepancies between classes, revealing potential signals for classification models.

Following the computation of descriptive statistics, the features of the dataset were systematically grouped according to their functional roles within the structure of Portable Executable (PE) files. This categorization serves a dual purpose: it not only enhances interpretability by organizing features into coherent blocks—such as file headers, section properties, import/export statistics, and resource descriptors—but also facilitates the identification of feature types that may be particularly informative for static malware detection. Grouping features in this manner supports the modeling process by allowing the analysis to reflect the underlying structure of PE files, which in turn can capture relevant patterns associated with malicious or benign behavior.

During this process, attributes deemed non-informative for the learning task—specifically `Name`, which serves only as an identifier; `md5`, a cryptographic hash used to reference each sample; and legitimate, the binary classification label—were excluded from the feature grouping to maintain clarity and focus on predictive variables. These excluded elements were nonetheless preserved in the dataset for tracking and evaluation purposes. Table 5 (see Appendix) provides a comprehensive summary of the grouped features retained for modeling, including concise descriptions of their technical significance. This structural organization aids both in the interpretability of the dataset and in the development of models that leverage domain-relevant features.

While static-analysis enables large-scale and reproducible experimentation, it inherently captures only structural characteristics of executables. The dynamic behaviors of malware—such as obfuscation or runtime evasion—remain beyond the scope of this dataset and are discussed further in Section 10.

5 Preprocessing

Following the workflow in Figure 1, the dataset underwent basic integrity checks to ensure its suitability for further processing. A systematic search confirmed the absence of missing (null) values and duplicate entries across all features. Additionally, two attributes—`Name` and `md5`—were removed from the dataset. These fields acted solely as unique identifiers and did not carry any semantic or statistical value relevant to the classification task. Their exclusion prevents potential information leakage and ensures that the model focuses exclusively on meaningful predictive features. These validations, while straightforward, are essential for preventing bias and ensuring that the subsequent modeling process is both reliable and reproducible. According to Sharifnia et al. [11], overlooking issues such as missing or inconsistent data can significantly compromise data quality, resulting in biased parameter estimates and misleading conclusions.

A class distribution analysis was then conducted to assess the balance between legitimate and malicious files. The results revealed a significant imbalance: out of a total of **138,047** samples, **96,724** were labeled as *malware* and only **41,323** as *benign*, corresponding to approximately **70.1%** and **29.9%** of the dataset, respectively. This imbalance poses a risk of biasing classifiers toward the majority class, reducing their ability to detect minority patterns accurately.

To address this, a random undersampling technique was applied to reduce the number of malware samples. After undersampling, both classes contained **41,323** samples, resulting in a balanced dataset of **82,646** instances. Despite this reduction, the dataset remains sufficiently large to support meaningful statistical inference and robust model training. This strategy is supported by recent research, which highlights that undersampling can reduce class overlap and enhance minority class visibility, especially when the majority class includes redundant or less informative samples [12]. Additionally, it improves training efficiency and can lead to more balanced decision boundaries, particularly in domains where minority class detection is critical, such as malware analysis.

To assess the effect of this resampling technique, both the original imbalanced dataset and the balanced version were retained for comparative modeling and evaluation. Figure 3 illustrates the class distributions before and after the undersampling process.

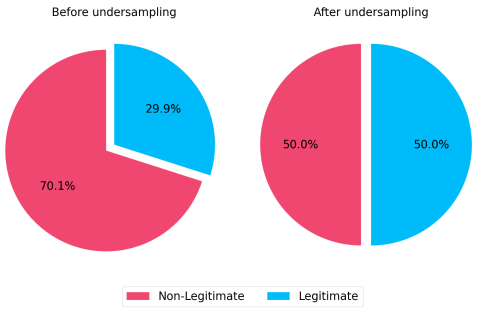


Figure 3: Class distribution before and after applying undersampling

6 Transformation

Dimensionality reduction techniques are essential in machine learning to address the challenges posed by high-dimensional data, often referred to as the "curse of dimensionality" [13, 14]. These techniques aim to reduce the number of input variables while preserving as much relevant information as possible, thereby enhancing model performance and interpretability. Common methods include:

- **Principal Component Analysis (PCA):** A linear technique that transforms the original variables into a new set of uncorrelated variables (principal components), ordered by the amount of variance they capture in the data.
- **Linear Discriminant Analysis (LDA):** A supervised method that seeks linear combinations of features that best separate two or more classes, maximizing class separability.
- **Tree-based methods:** Algorithms like Random Forest and Extra Trees assess feature importance based on their contribution to decision splits, effectively selecting the most informative features for classification tasks. Selecting the most informative features for classification tasks.

These methods not only reduce computational complexity but also help in mitigating overfitting and improving the generalization capability of models.

In this study, two ensemble-based methods were used to evaluate the relevance of each attribute: Random Forest Classifier and Extra Trees Classifier, both configured

with n_estimators=50 and default parameters as per the scikit-learn implementation [15]. These tree-based models compute feature importance based on the contribution of each attribute to the split decisions across trees. While Random Forests use optimized splits, Extra Trees introduce greater randomness by choosing thresholds at random, which can reduce overfitting and variance in high-dimensional contexts. Both models were independently trained on the original (imbalanced) dataset and the balanced version. The goal was to compare the stability and consistency of the selected attributes under varying class distributions. In each case, 14 features were automatically selected using the SelectFromModel method which are presented in Table 2 for comparison.

Table 2: Features selected by extra trees and random forest classifiers

Extra Trees Classifier	Random Forest Classifier
Machine	Characteristics
SizeOfOptionalHeader	BaseOfData
Characteristics	ImageBase
ImageBase	MajorOperatingSystemVersion
MajorOperatingSystemVersion	MinorImageVersion
MajorSubsystemVersion	Subsystem
Subsystem	DllCharacteristics
DllCharacteristics	SizeOfStackReserve
SizeOfStackReserve	SectionsMaxEntropy
SectionsMinEntropy	ExportNb
SectionsMaxEntropy	ResourcesNb
ResourcesMinEntropy	ResourcesMinEntropy
ResourcesMaxEntropy	ResourcesMinSize
VersionInformationSize	VersionInformationSize

A comparison of the selected features reveals that both models share 9 common attributes, including Subsystem, DllCharacteristics, and VersionInformationSize. These overlapping features suggest that both methods consistently identify them as informative for distinguishing between legitimate and malicious files. However, each classifier also selected 5 distinct features. Extra Trees included attributes such as Machine and SizeOfOptionalHeader, which showed strong positive correlations with the target, while Random Forest selected attributes like BaseOfData and ExportNb, which exhibited lower individual correlation but may provide complementary decision information. This divergence highlights how different ensemble strategies can lead to feature sets with varying levels of redundancy, diversity, and interpretability.

To further analyze the relationship between the selected attributes and the target variable, correlation matrices were computed to measure the linear association between each feature and the legitimate label. As shown in Figures 11 and 12, see Appendix, features selected by the Extra Trees Classifier exhibit generally stronger absolute correlations with the target, notably Machine and SizeOfOptionalHeader (both $r = 0.55$), and Subsystem ($r = 0.51$). Additionally, negatively correlated features such as DllCharacteristics ($r = -0.63$) and SectionsMaxEntropy ($r = -0.62$) show clear discriminatory power.

In contrast, features selected by the Random Forest Classifier display weaker overall correlations with the target variable, although some attributes like Subsystem ($r = 0.51$), DllCharacteristics ($r = -0.63$), and VersionInformationSize ($r = 0.38$) still demonstrate meaningful relationships. This suggests that Extra Trees may have prioritized features with higher individual predictive strength, while Random Forest may have selected a more diverse feature set with complementary interactions not solely captured by linear correlation.

Finally, four distinct experimental setups were constructed to evaluate model performance under different conditions:

1. Original dataset + features selected by Extra Trees
2. Original dataset + features selected by Random Forest
3. Balanced dataset + features selected by Extra Trees
4. Balanced dataset + features selected by Random Forest

Each experimental setup was evaluated using two different train-test split configurations: one with 80% of the data used for training and 20% for testing, and another with an equal split of 50% for training and 50% for testing. This dual evaluation strategy enables the assessment of each model's robustness, generalization capability, and sensitivity to the amount of training data available. The inclusion of a 50/50 split allows us to examine model performance under limited training data conditions, which is particularly relevant in real-world scenarios where labeled samples may be scarce. Together, these experimental variations enable a comparative analysis of how feature selection strategies, class distribution, and training size interact to influence classifier performance, interpretability, and computational cost.

7 Data mining

Data mining has emerged as a vital component in malware detection due to its ability to uncover hidden patterns and generalize from labeled examples, going beyond the limitations of traditional signature-based techniques. As highlighted by Souri and Hosseini [16], the integration of data mining with machine learning enables robust classification mechanisms that can detect both known and previously unseen malware threats.

In this study, we apply data mining techniques to perform supervised classification of static malware attributes. The selected algorithms—Backpropagation Neural Network (BPNN), Decision Tree, Random Forest, and Support Vector Machine—are widely established in the literature for their effectiveness in modeling diverse decision boundaries while balancing predictive performance with interpretability.

The following subsections detail the implementation and theoretical background of each algorithm used in this work.

7.1 Backpropagation neural network (BPNN)

BPNN are fundamental due to their ability to model complex, nonlinear relationships by automatically learning hierarchical feature representations from raw data [17]. BPNN is a type of supervised learning model based on artificial neural networks, composed of an input layer, one or more hidden layers, and an output layer [17]. Each neuron in a given layer is fully connected to all neurons in the subsequent layer, and these connections are associated with weights that determine the strength and direction of influence between neurons. Weights and biases are the core trainable parameters of the network and are adjusted during training to reduce prediction error.

The training process in BPNN consists of two main phases: forward propagation and backpropagation. During forward propagation, input data is passed through the network layer by layer to produce an output. The network's error is then calculated by comparing this output to the expected target. In the backward pass, the error is propagated backward through the network to compute gradients for each weight, using the chain rule of calculus. These gradients are then used to update the weights in a way that minimizes the loss function. This process is repeated for several iterations or epochs until convergence is achieved [17].

Figure 4 illustrates the general architecture of the BPNN implemented in this study. While each hidden layer in the actual model consists of 128 neurons, the figure provides a simplified visualization using ellipses to indicate additional units. The network receives an input vector of length n, processes it through two fully connected hidden layers, and produces a single binary output via a sigmoid-activated neuron.

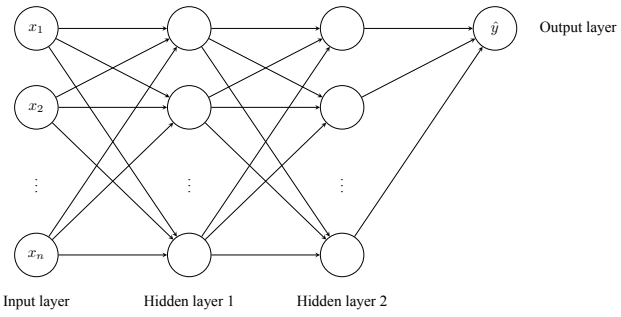


Figure 4: Simplified architecture of the BPNN.

In this study, we implemented a fully connected feed-forward BPNN using the `Keras` API with `TensorFlow` as the backend. The architecture consisted of two hidden layers, each with 128 neurons and ReLU activation functions (`activation="relu"`), and an output layer with a single neuron and a sigmoid activation function (`activation="sigmoid"`) for binary classification. The model was compiled using the Adam optimizer with a learning rate of 10^{-3} (`learning_rate=1e-3`) and the binary cross-entropy loss function (`loss="binary_crossentropy"`). Binary accuracy (`metrics=["BinaryAccuracy"]`) was used as the evaluation metric. All other hyperparameters were kept at their default values.

7.2 Decision tree (DT)

DT are simple, interpretable models that construct tree structures by recursively splitting the dataset based on feature values. While easy to understand and implement, their performance can degrade in high-variance or noisy environments due to overfitting [18]. DT is a supervised learning algorithm that recursively partitions the instance space using a rooted and directed tree structure, where each internal node performs a test on an input attribute, and each leaf node represents a decision outcome or a probability distribution over target classes [19]. This top-down approach, also known as recursive partitioning or "divide and conquer", results in a hierarchical model that is both interpretable and easy to visualize. In each step of the tree construction, the algorithm selects the attribute and corresponding split point that maximize a given splitting criterion—typically the information gain, Gini index, or reduction in variance—depending on the nature of the task (classification or regression).

The algorithm proceeds until a stopping criterion is met, such as reaching a maximum tree depth, a minimum number of samples per node, or the absence of further information gain. Each path from the root to a leaf node can be interpreted as a decision rule, which enables rule-based reasoning and transparent decision-making. In fact, each internal node acts as a conditional `if` statement, and each branch corresponds to a possible outcome (`else`), making the entire path a sequence of nested `if-else` conditions. This logic-based structure is visually illustrated in Figure 5, where each decision node leads to a branch representing a binary condition and eventually a classification output. Each branch reflects a conditional rule, and leaves represent final decisions.

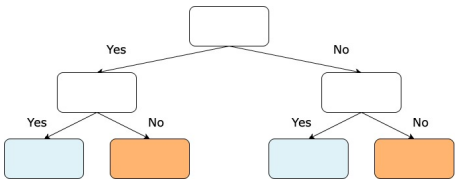


Figure 5: General structure of a binary DT.

In this study, we employed the `scikit-learn` implementation of DTs [15], which is based on the CART algorithm (Classification and Regression Trees), originally introduced by Breiman et al. (as summarized in [19]). Unlike other tree algorithms such as ID3 or C4.5, CART constructs strictly binary trees and supports both classification and regression tasks by using the Gini impurity and mean squared error as default criteria, respectively. In our configuration, the DT was initialized with a maximum depth of 10

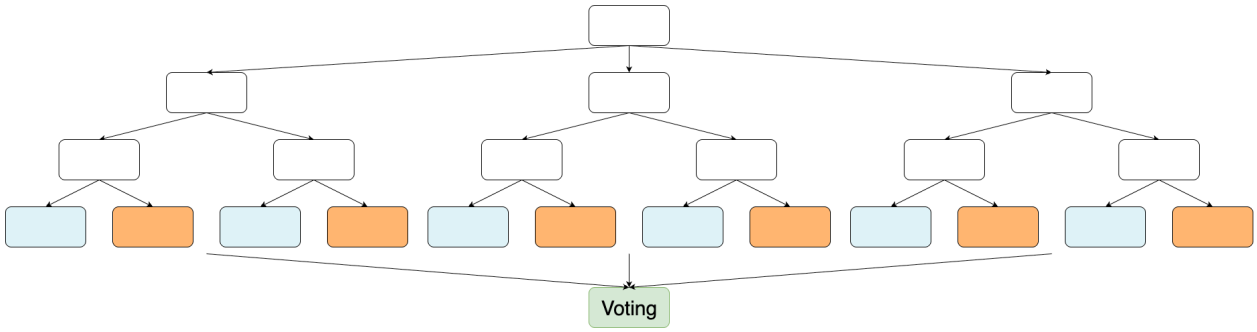


Figure 6: Simplified representation of a RF.

(`max_depth=10`) to control model complexity and reduce the risk of overfitting, while the remaining hyperparameters were left at their default values.

7.3 Random forest (RF)

RF address the limitation of DT by aggregating the predictions of multiple decision trees trained on random subsets of the data, leading to improved accuracy, reduced variance, and robustness across diverse data distributions [18]. Therefore, RF is a widely used ensemble learning method designed to improve the performance and robustness of decision tree classifiers by combining the predictions of multiple trees into a single output. RF was proposed by Leo Breiman and Adele Cutler and is considered an extension of the bagging method [20]. The core idea behind RF is to build a large collection of de-correlated decision trees and aggregate their outputs, using majority voting for classification or averaging for regression.

The algorithm constructs each tree using a bootstrap sample of the training data (sampling with replacement), and at each split, it selects the best feature from a randomly chosen subset of features—rather than considering all features as in a traditional decision tree. This dual randomness, in both data and feature selection, introduces diversity among the trees, which reduces overfitting and improves generalization performance. The underlying decision trees are typically trained using the CART algorithm, which employs criteria such as Gini impurity or mean squared error to determine optimal splits [20].

Figure 6 shows a simplified representation of a RF. Each tree in the ensemble produces its own prediction based on different subsets of data and features. Multiple decision trees are trained independently, and their predictions are aggregated through a majority voting mechanism (for classification), illustrated by the green node labeled *Voting*, which determines the final output of the model.

In this study, we used the `scikit-learn` implementation of RF [15], configuring the model with 100 trees (`n_estimators=100`) and a fixed random seed (`random_state=42`) to ensure reproducibility. All other hyperparameters were left at their default values.

7.4 Support vector machine (SVM)

SVM is a supervised learning algorithm that aims to find the optimal hyperplane that separates data points of different classes in an N-dimensional space [21]. This hyperplane is chosen to maximize the margin, defined as the distance between the closest points (support vectors) of opposing classes. The idea is to achieve the most generalizable separation between classes, reducing the risk of overfitting and improving performance on unseen data. Studies have demonstrated their strong discriminative power in educational and cybersecurity datasets, particularly when using sigmoid or radial basis kernels [22].

Figure 7 illustrates a typical linear SVM scenario in a 2D space. The solid line represents the optimal separating hyperplane, while the dashed lines define the margin boundaries. The hyperplane separates the two classes while maximizing the margin between the closest points, known as support vectors, which determine its position and orientation.

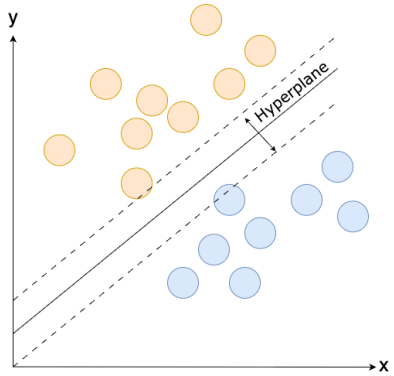


Figure 7: Linear SVM classification

Developed in the 1990s by Vladimir Vapnik and colleagues, SVMs support both linear and non-linear classification tasks. When data is linearly separable, a linear SVM fits the widest possible margin between classes. However, for more complex data distributions, SVMs use the kernel trick—a mathematical technique that implicitly maps the input features into a higher-dimensional space where linear separation becomes feasible. Common kernel functions in-

clude the polynomial kernel, sigmoid kernel, and radial basis function (RBF) kernel [21].

In this study, the SVM model was implemented using the `scikit-learn` library [15]. We employed the radial basis function (RBF) as the kernel (`kernel="rbf"`), with the regularization parameter set to `C=1.0` and `gamma=0.7`, which controls the influence of individual training samples. The remaining hyperparameters were kept at their default values.

8 Results

Building on the experimental setups introduced earlier, this section presents a detailed evaluation of the models across all scenarios. The goal is to analyze how the feature selection strategy, class distribution, and training data size affect classification performance.

To quantify the performance of each classifier, we used standard metrics: accuracy, F1-score, mean squared error (MSE), and the Area Under the Receiver Operating Characteristic Curve (AUC–ROC, hereafter AUC). These metrics were computed using `scikit-learn` [15], and are complemented by confusion matrices to visualize prediction outcomes.

8.1 Evaluation metrics

The evaluation metrics were calculated based on the confusion matrix, which summarizes prediction outcomes as illustrated in Figure 8. The formulas follow standard definitions in classification evaluation [23].

- **Accuracy:**

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

- **F1-score:**

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}},$$
$$\text{with Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}$$

- **Mean Squared Error (MSE):**

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

- **AUC:**

$$\text{AUC} = \int_0^1 \text{TPR}(x)\, d(\text{FPR}(x))$$

where TPR (True Positive Rate) is defined as $\text{TPR} = \frac{TP}{TP+FN}$ and FPR (False Positive Rate) is defined as $\text{FPR} = \frac{FP}{FP+TN}$.

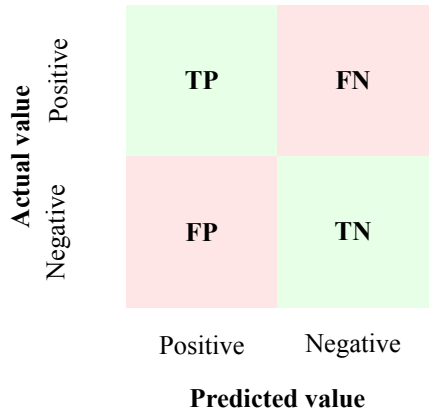


Figure 8: Structure of a confusion matrix

In this context, **TP** (True Positives) refers to instances correctly predicted as belonging to the positive class, **TN** (True Negatives) to instances correctly predicted as negative, **FP** (False Positives) to negative instances incorrectly classified as positive, and **FN** (False Negatives) to positive instances incorrectly classified as negative.

8.2 Results on the original dataset (unbalanced)

Table 3 summarizes the metrics obtained when training the models with the original imbalanced dataset. Two different feature selectors were evaluated: Extra Trees Classifier and Random Forest Classifier, each under 80/20 and 50/50 train-test splits. In general, the best performance was consistently achieved by the RF model, followed closely by DT, especially in terms of F1-score and MSE. The BPNN model showed the lowest accuracy and F1-score across most configurations, particularly under the Extra Trees 80/20 split.

Confusion matrices corresponding to these results are shown in Figures 9(a) to 9(d).

8.3 Results on the balanced dataset

Table 4 presents the results when the dataset was balanced through undersampling. In this case, BPNN performance improved significantly in both accuracy and F1-score across all settings. While tree-based models such as RF and DT retained their strong performance, the performance gap between them and BPNN/SVM narrowed considerably.

The corresponding confusion matrices are shown in Figures 10(a) to 10(d).

8.4 Interpretation

The evaluation results and confusion matrices across all experimental setups reveal critical insights into how the feature selection strategy, class distribution, and training size affect the behavior of different classifiers.

Table 3: Evaluation results on the original (unbalanced) dataset.

Feature Selector	Train/Test	Model	Accuracy (%)	F1-score	MSE	AUC
Extra Trees Classifier	80/20	BPNN	91.84	0.8458	0.0816	0.9404
		DT	99.11	0.9853	0.0089	0.9943
		RF	**99.43**	**0.9906**	**0.0057**	**0.9995**
		SVM	98.45	0.9749	0.0155	0.9958
	50/50	BPNN	96.04	0.9318	0.0396	0.9392
		DT	99.01	0.9834	0.0099	0.9934
		RF	**99.34**	**0.9890**	**0.0066**	**0.9995**
		SVM	98.06	0.9683	0.0194	0.9945
Random Forest Classifier	80/20	BPNN	96.73	0.9452	0.0327	0.9588
		DT	99.03	0.9840	0.0097	0.9944
		RF	**99.45**	**0.9909**	**0.0055**	**0.9995**
		SVM	95.96	0.9374	0.0404	0.9897
	50/50	BPNN	96.60	0.9422	0.0340	0.9568
		DT	99.06	0.9842	0.0094	0.9929
		RF	**99.37**	**0.9894**	**0.0063**	**0.9995**
		SVM	95.45	0.9290	0.0455	0.9871

Table 4: Evaluation results on the balanced dataset.

Feature Selector	Train/Test	Model	Accuracy (%)	F1-score	MSE	AUC
Extra Trees Classifier	80/20	BPNN	93.07	0.9327	0.0693	0.9572
		DT	98.96	0.9897	0.0104	0.9950
		RF	**99.32**	**0.9932**	**0.0068**	**0.9994**
		SVM	98.28	0.9832	0.0172	0.9954
	50/50	BPNN	94.36	0.9414	0.0564	0.9438
		DT	98.93	0.9894	0.0107	0.9944
		RF	**99.25**	**0.9925**	**0.0075**	**0.9993**
		SVM	97.98	0.9802	0.0202	0.9949
Random Forest Classifier	80/20	BPNN	94.59	0.9456	0.0541	0.9247
		DT	98.86	0.9887	0.0114	0.9952
		RF	**99.27**	**0.9928**	**0.0073**	**0.9993**
		SVM	96.37	0.9652	0.0363	0.9871
	50/50	BPNN	96.34	0.9630	0.0366	0.9353
		DT	98.82	0.9883	0.0118	0.9933
		RF	**99.23**	**0.9923**	**0.0077**	**0.9990**
		SVM	96.20	0.9634	0.0380	0.9846

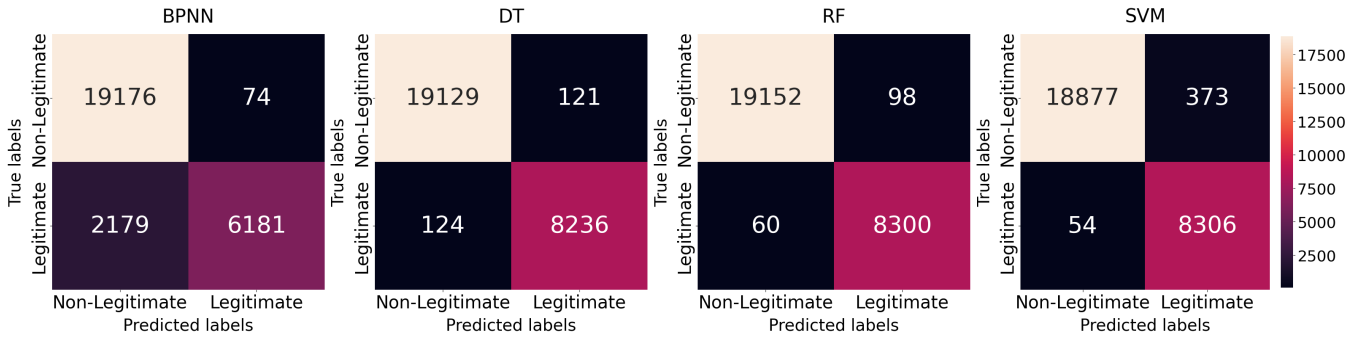
Under the original (unbalanced) dataset, both the **Extra Trees Classifier** and the **Random Forest Classifier** enabled the tree-based models (DT and RF) to achieve outstanding performance. In particular, RF consistently delivered the best results across nearly all configurations. For instance, when using the Random Forest Classifier with an 80/20 split on the unbalanced dataset, RF achieved an F1-score of 0.9909 and an MSE of 0.0055, with only 57 false negatives and 96 false positives (Figure 9(c)). Similarly, DT obtained 99.03% accuracy with only 134 false negatives and 134 false positives in the same setting.

The performance of BPNN under unbalanced data was comparatively lower, particularly with the Extra Trees Classifier in the 80/20 setting. As shown in Figure 9(a), BPNN misclassified 2179 legitimate samples as false negatives and 74 non-legitimate samples as false positives, resulting in a relatively low F1-score of 0.8458 and the highest MSE among all models (0.0816). Although its accuracy and F1-score improved with more training data (96.04% accuracy and 0.9318 F1-score in the 50/50 split), the gap with tree-based models remained noticeable (Figure 9(b)).

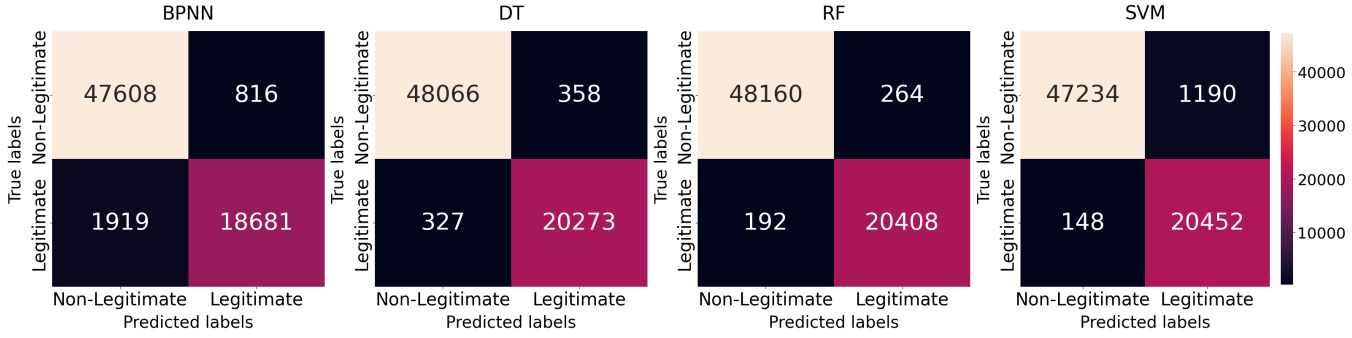
SVM also exhibited sensitivity to class imbalance. Particularly, in the Random Forest Classifier with 50/50 split (Figure 9(d)), SVM produced the highest number of false positives (3115) among all models evaluated and an F1-score of 0.9290, indicating potential overfitting or misclassification of non-legitimate samples.

After applying class balancing through random undersampling, substantial improvements were observed for both BPNN and SVM. The most dramatic gains were evident in the confusion matrices and F1-scores. For example, in the Extra Trees Classifier with 80/20 split, BPNNs number of false negatives decreased sharply from 2179 (Figure 9(a)) to 396 (Figure 10(a)), and its F1-score rose from 0.8458 to 0.9327. A similar trend occurred in the 50/50 split (Figure 10(b)), where BPNN reached an F1-score of 0.9414, rivaling the performance of the DT model.

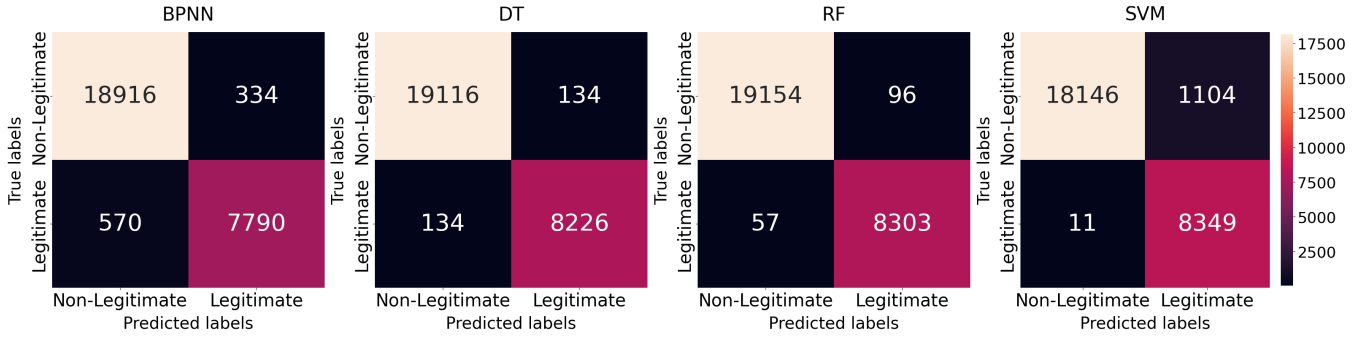
SVM also benefited from class balancing. Its false posi-



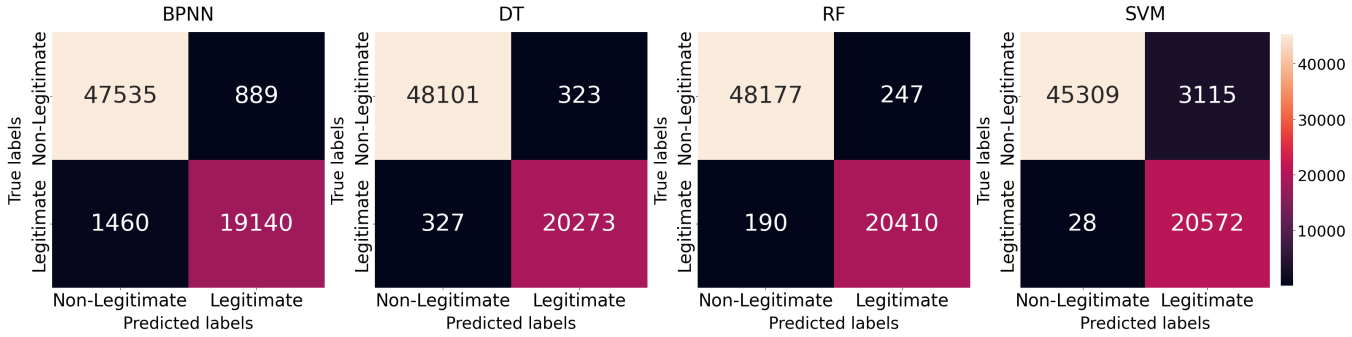
(a) Feature selector = Extra Trees Classifier (80/20 split)



(b) Feature selector = Extra Trees Classifier (50/50 split)



(c) Feature selector = Random Forest Classifier (80/20 split)



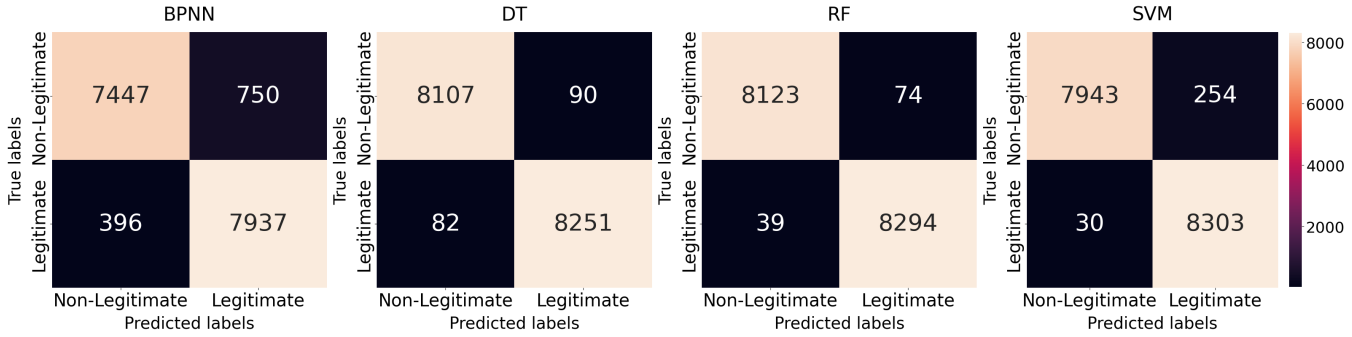
(d) Feature selector = Random Forest Classifier (50/50 split)

Figure 9: Confusion matrices derived from unbalanced datasets.

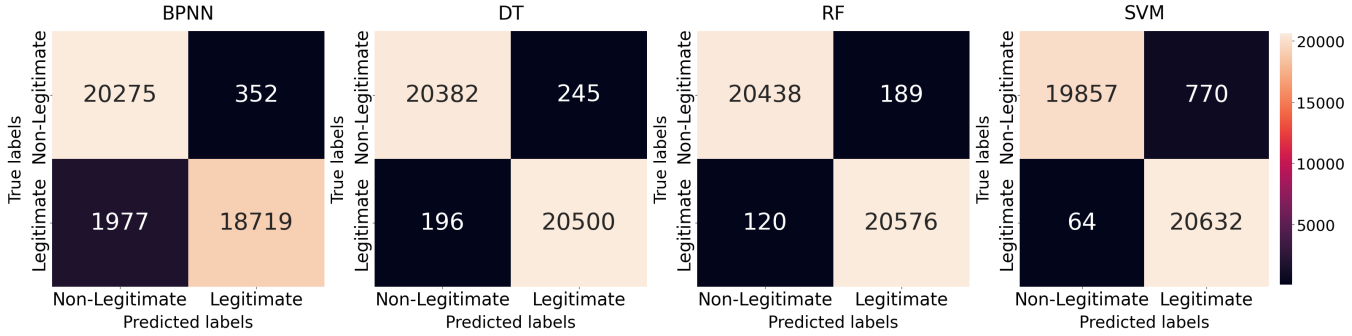
tives dropped from 373 (Figure 9(a)) to 254 (Figure 10(a)) and its F1-score improved to 0.9832. In the Random Forest Classifier 50/50 split (Figure 10(d)), SVM reached an F1-score of 0.9634, which was a significant leap compared to its previous 0.9290 under unbalanced conditions.

The Random Forest Classifier again proved to be the most stable and performant model, maintaining F1-scores above 0.99 in all conditions. Its robustness was evident not only in metrics but also in the confusion matrices (Figures 9(c) and 10(c)), where it consistently minimized both types of errors.

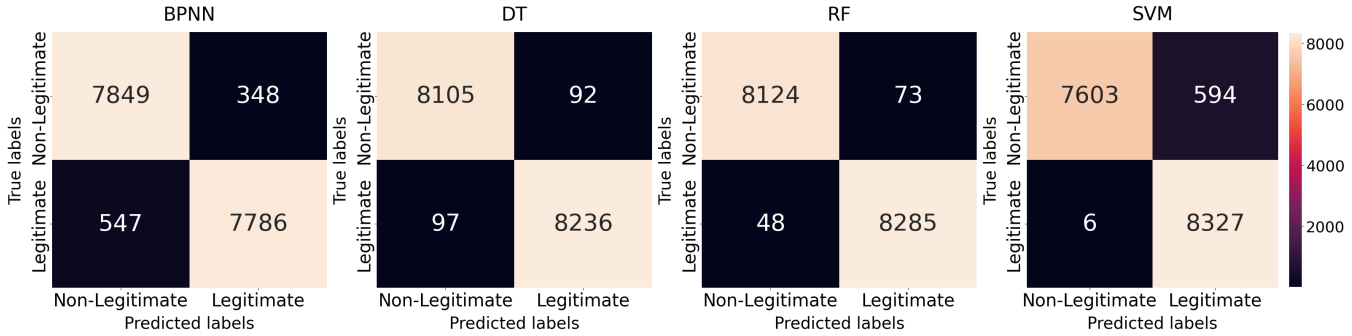
DT also demonstrated solid performance, though it was



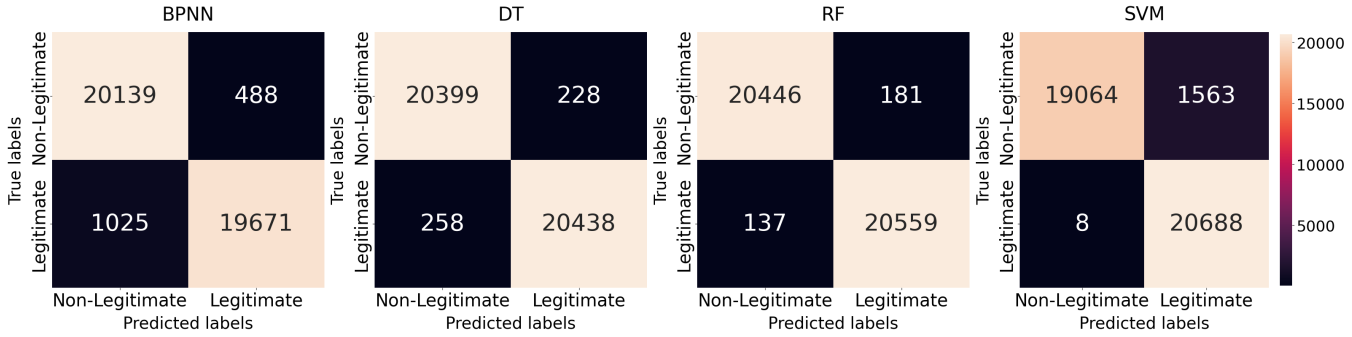
(a) Feature selector = Extra Trees Classifier (80/20 split)



(b) Feature selector = Extra Trees Classifier (50/50 split)



(c) Feature selector = Random Forest Classifier (80/20 split)



(d) Feature selector = Random Forest Classifier (50/50 split)

Figure 10: Confusion matrices derived from balanced datasets.

slightly less robust than RF when evaluated across multiple conditions. Nevertheless, DTs simplicity and interpretability, combined with its consistent accuracy above 98.8%, make it a compelling alternative in scenarios where model transparency is essential.

Taken together, class balancing emerged as a critical factor in enhancing the performance of BPNN and SVM. BPNN in particular showed the largest relative improvement, validating its ability to generalize well when trained with balanced data. The change was evident from the patterns observed in the confusion matrices, where the misclassification of legitimate samples drastically declined.

The findings also underline the importance of evaluating model performance not only through scalar metrics like accuracy and F1-score but also through the detailed inspection of confusion matrices, which reveal nuanced differences in behavior under various training conditions.

Ultimately, the RF and DT models consistently outperformed other classifiers, regardless of feature selection method or class balance. However, BPNN proved to be a viable and competitive option when supported with balanced data, highlighting the potential of neural models in binary classification tasks involving skewed class distributions.

In addition to the evaluation results, it is important to note that extensive experimentation was conducted to determine optimal hyperparameter settings for each model. The configurations reported in this study were selected after testing multiple combinations and correspond to those that yielded the best trade-off between accuracy and generalization. Notably, BPNN and SVM models required more careful tuning and exhibited longer training times compared to tree-based models. This difference in computational cost was particularly evident during iterative experimentation, where DT and RF consistently trained faster, making them more suitable for rapid prototyping and scenarios with limited computational resources.

In the case of BPNN, the architectural decision to use two hidden layers with moderate size was guided by empirical testing. Configurations with 128 neurons per layer offered a practical balance between representational capacity and computational efficiency, capturing complex patterns without overfitting or incurring excessive training costs. Although other configurations were considered, larger networks did not show meaningful improvements, reinforcing this choice as an effective trade-off.

9 Discussion

The experimental findings presented in Tables 3 and 4 reveal that the proposed models achieve performance levels comparable to those reported in previous malware detection studies (Table 1). Earlier works based on deep neural networks—such as Convolutional Neural Network, Recurrent Neural Network, or Active Learning architectures—reported accuracies above 99%. In contrast, the RF model in the present study reached a peak accuracy of 99.45% and an F1-score of 0.9909 on the unbalanced dataset, while maintaining similar performance (99.27% accuracy and 0.9928 F1-score) on the balanced dataset. These results position RF on par with advanced deep-learning approaches, despite its substantially lower computational complexity and training time.

It is important to note that although previous studies reported high accuracy values (while some also showed results around 70–80%), this metric alone can be misleading when the dataset is imbalanced. When one class is significantly more frequent than the other, a model can achieve high accuracy without truly learning to distinguish between classes. Therefore, complementary metrics such as Recall, F1-score, and AUC provide a more complete and fair assessment of model performance. In particular, AUC evaluates how well the model separates positive and negative samples—that is, the probability that the model assigns a higher score to a positive instance than to a negative one. As shown in Tables 3 and 4, the RF model achieved an AUC value of 0.9995, which is nearly perfect.

A notable finding is the consistent superiority of tree-based methods—particularly RF and DT—over both neural and kernel-based models. Several factors explain this behavior. First, tree ensembles inherently manage nonlinear feature interactions and handle irrelevant attributes effectively, which is a crucial advantage when using static features extracted from PE files. Second, tree-based algorithms are less sensitive to feature scaling and class imbalance, leading to stable convergence without extensive hyperparameter tuning. Moreover, the feature selection methods (Extra Trees and Random Forest) used in this work synergize naturally with DT and RF models, as they rely on the same impurity-based importance measures. This alignment likely amplified their discriminative power, yielding near-perfect accuracy and minimal MSE.

Although RF achieved near-perfect accuracy, its exceptional performance must be interpreted cautiously. Ensemble models can overfit when trained on high-dimensional data or when feature redundancy is high. Nevertheless, the low MSE and consistent results across training/test splits suggest strong generalization rather than overfitting. RF also offers partial interpretability through feature importance metrics, making it more explainable than most deep-learning approaches that function as black boxes. Runtime efficiency was another advantage: DT and RF required significantly shorter training and inference times than BPNN and SVM, supporting their suitability for real-time or resource-limited malware detection scenarios.

Regarding novelty, this work does not introduce a new algorithmic architecture but rather provides a systematic comparative evaluation of established ML techniques under a new dataset configuration with explicit class balancing and feature selection strategies. This approach bridges a gap left by previous studies that primarily emphasized raw accuracy without addressing imbalance or interpretability. The contribution therefore lies in demonstrating that traditional, interpretable methods—when properly tuned and combined with informed feature selection—can achieve state-of-the-art performance comparable to deep neural networks while remaining computationally efficient and transparent.

In summary, the discussion reinforces three core insights: (1) tree-based models remain a competitive and interpretable choice for static malware detection; (2) data balancing and feature selection significantly enhance neural and kernel-based classifiers; and (3) explainability, runtime efficiency, and model simplicity are decisive factors when

translating research findings into operational cybersecurity systems.

10 Conclusions and future work

This study presented an approach to the detection of malware using supervised machine learning models trained on static features extracted from malware and benign executable samples. The problem was addressed through the lens of binary classification, with a clear experimental structure grounded in the KDD process. By evaluating four well-known classifiers—BPNN, DT, RF, and SVM—across different training sizes, feature selection strategies, and class balance conditions, the work offers an in-depth perspective on the capabilities and limitations of each algorithm in real-world scenarios.

The integration of the KDD process provided a structured and reproducible pipeline, encompassing data selection, preprocessing, transformation, feature selection, and evaluation. This methodology allowed for a disciplined experimentation strategy that ensured consistency across tests while providing room for targeted insights. In particular, the use of tree-based feature selection methods, namely Extra Trees and Random Forest classifiers, was effective in reducing input dimensionality while preserving high predictive performance. These selectors not only improved training efficiency but also helped mitigate potential noise and redundancy in the feature space.

Quantitative results showed that the RF and DT models consistently outperformed other classifiers in nearly all configurations, achieving F1-scores above 0.99 and minimum MSE across both unbalanced and balanced datasets. Their robustness to class imbalance and their ability to produce reliable predictions make them strong candidates for malware detection systems in high-stakes environments. Notably, RF achieved outstanding performance even with fewer training samples, suggesting that it can generalize well with limited labeled data.

BPNN, on the other hand, initially struggled under unbalanced data conditions, presenting higher false negative rates and lower F1-scores. However, after balancing the dataset via undersampling, BPNN showed the most significant performance improvement among all classifiers. Its F1-score increased by nearly 10 percentage points in some scenarios, and its error rates decreased substantially. These results confirm that while neural networks may be more sensitive to class distribution, they also possess strong generalization capabilities when trained under balanced conditions with representative data. Similarly, SVM exhibited sensitivity to class imbalance, with a tendency to produce high false positive rates. Nevertheless, it also benefitted from class balancing, improving its F1-score and reducing misclassification rates.

Beyond model performance, this study highlights the strategic convergence of three core domains: data science, machine learning, and cybersecurity. The use of static malware analysis aligns with practices in ethical hacking, where reverse engineering and behavioral understanding of malicious software support the development of robust defense mechanisms. The incorporation of automated classification models into malware detection workflows represents a step forward toward proactive and intelligent cybersecurity systems capable of adapting to ever-evolving threats. In this context, data-driven approaches not only improve detection accuracy but also enable scalable and interpretable systems that align with ethical standards of cyber defense.

10.1 Limitations

Although the proposed approach demonstrates strong performance in detecting malware through static analysis, certain limitations should be acknowledged. The methodology relies exclusively on static features extracted from PE files, which may not capture behaviors exhibited only during runtime. As a result, malware employing advanced obfuscation or dynamic evasion techniques could remain undetected. Previous studies, such as the ARMED framework by Castro et al. [24], have shown that minor modifications to executable files can drastically reduce the effectiveness of static detectors, underscoring the inherent fragility of static-only approaches. Furthermore, the dataset used—while extensive—may not fully represent the diversity of emerging threats in real-world environments. These factors highlight the need for further research integrating complementary behavioral insights and broader datasets to address evolving malware techniques.

10.2 Future work

Future work should explore several avenues to extend the present findings. One direction involves incorporating dynamic features, such as runtime behavior, system calls, and network activity, which can provide richer contextual information and enhance the model's ability to detect complex or evasive behaviors. Another promising path includes the integration of hybrid or ensemble architectures that combine the strengths of multiple classifiers—e.g., a BPNN-RF stack—to capture both nonlinear patterns and decision-level robustness. Additionally, future research should consider adversarial robustness, particularly the model's resistance to manipulated input data, by exploring adversarial training or applying explainability frameworks such as SHapley Additive exPlanations (SHAP) or Local Interpretable Model-agnostic Explanations (LIME) to promote interpretability and accountability. Finally, applying the methodology to real-time or online detection settings, and generalizing it across platforms and malware families, would enhance the applicability and operational value of the proposed solutions.

References

[1] National Institute of Standards and Technology, "Malware definition." https://csrc.nist.gov/glossary/term/malware, 2023. Accessed: 2025-05-17.

[2] Symantec, "Internet security threat report 2023." https://symantec-enterprise-blogs.security.com, 2023. Accessed: 2025-05-17.

[3] R. Sihwail, K. Omar, and K. A. Z. Ariffin, "A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis," *International Journal on Advanced Science, Engineering and Information Technology*, vol. 8, no. 4-2, pp. 1662–1671, 2018.

[4] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "The kdd process for extracting useful knowledge from volumes of data," *Communications of the ACM*, vol. 39, no. 11, pp. 27–34, 1996.

[5] M. Shakib, "Android Malware Detection Approach's Based on Genetic Ai, Cnn, Rnn, Lstm, Gru, and Active Learning." Social Science Research Network, 2023.

[6] H. C. Tanuwidjaja and K.-j. Kim, "Enhancing malware detection by modified deep abstraction and weighted feature selection," in *In Proceedings of the 2020 Symposium on Cryptography and Information Security, Seoul, Republic of Korea*, pp. 1–8, 2020.

[7] R. K. ROY, "A few approaches in encrypted malware classifications." Zenodo, 2022.

[8] R. Baker del Aguila, C. D. Contreras Pérez, A. G. Silva-Trujillo, J. C. Cuevas-Tello, and J. Nunez-Varela, "Static malware analysis using low-parameter machine learning models," *Computers*, vol. 13, no. 3, p. 59, 2024.

[9] VirusShare. https://virusshare.com/, Accessed on March 25, 2025.

[10] The pandas development team, "pandas-dev/pandas: Pandas." Zenodo, Sept. 2025.

[11] A. M. Sharifnia, D. E. Kpormegbey, D. K. Thapa, and M. Cleary, "A primer of data cleaning in quantitative research: Handling missing values and outliers," *Journal of Advanced Nursing*, vol. 0, pp. 1–6.

[12] M. Carvalho, A. J. Pinho, and S. Brás, "Resampling approaches to handle class imbalance: a review from a data perspective," *Journal of Big Data*, vol. 12, no. 1, p. 71, 2025.

[13] R. E. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.

[14] D. Peng, Z. Gui, and H. Wu, "Interpreting the curse of dimensionality from distance concentration and manifold effect," *arXiv preprint arXiv:2401.00422*, 2023.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python." https://scikit-learn.org/stable/, 2011. Accessed: 2025-05-20.

[16] A. Souri and R. Hosseini, "A state-of-the-art survey of malware detection approaches using data mining techniques," *Human-centric Computing and Information Sciences*, vol. 8, no. 1, p. 3, 2018.

[17] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[18] A. Kinasih, A. Handayani, J. Ardiansah, and N. Damanhuri, "Comparative analysis of decision tree and random forest classifiers for structured data classification in machine learning," *Science in Information Technology Letters*, vol. 5, pp. 13–24, 11 2024.

[19] L. Rokach and O. Maimon, *Decision Trees*, pp. 165–192. Boston, MA: Springer US, 2005.

[20] IBM, "What is random forest?." https://www.ibm.com/think/topics/random-forest, 2021. Accessed: 2025-05-24.

[21] N. Cristianini and E. Ricci, "Support vector machines," in *Encyclopedia of algorithms*, pp. 928–932, Springer, 2008.

[22] T. Admassu, A. Salau, G. Chhabra, K. Kaushik, and S. Braide, "Evaluation of random forest and support vector machine models in educational data mining," 06 2024.

[23] A. Geron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 2nd ed., 2019.

[24] R. Castro, C. Schmitt, and G. Rodosek, "Armed: How automatic malware modifications can evade static detection?," pp. 20–27, 03 2019.

Appendix

In this section we present the grouped features and their descriptions for static malware analysis (Table 5), the descriptive statistics for the features separated by class (Tables 6 and 7), and the correlation matrices (Figures 11 and 12),.

Table 5: Grouped features and descriptions for static malware analysis

Group	Feature	Description
File Header	Machine	Identifies the required architecture for execution and helps determine compatibility with system hardware.
	SizeOfOptionalHeader	Defines the byte size of the optional header that contains execution-specific fields.
	Characteristics	Specifies attributes of the file such as its type and layout via a set of binary flags.
Optional Header	MajorLinkerVersion	Stores the primary version number of the linker used during compilation.
	MinorLinkerVersion	Stores the secondary version number of the linker.
	SizeOfCode	Indicates the combined size in bytes of all executable code sections.
	SizeOfInitializedData	Indicates the total size in bytes of initialized data sections.
	SizeOfUninitializedData	Refers to the size of memory blocks declared but not initialized during compilation.
	AddressOfEntryPoint	Represents the memory offset where execution starts once the binary is loaded.
	BaseOfCode	Marks the beginning memory address of the code section.
	BaseOfData	Marks the beginning memory address of the initialized data section.
	ImageBase	Indicates the default memory location where the file should be loaded by the operating system.
	SectionAlignment	Describes how the sections are aligned in virtual memory during load time.
	FileAlignment	Describes how the sections are aligned on disk.
	MajorOperatingSystemVersion	Declares the major version of the minimum supported operating system.
	MinorOperatingSystemVersion	Declares the minor version of the minimum supported operating system.
	MajorImageVersion	Indicates the major version label assigned to the image by the developer.
	MinorImageVersion	Indicates the minor version label assigned to the image.
	MajorSubsystemVersion	Provides the major version of the subsystem environment required.
	MinorSubsystemVersion	Provides the minor version of the subsystem environment required.
	SizeOfImage	Specifies the full size in memory of the loaded binary including headers and sections.
	SizeOfHeaders	Indicates the total byte size of all headers in the file.
	CheckSum	Used to verify integrity of the file; often unused during execution.
	Subsystem	Determines the expected runtime environment required to execute the binary.
	DllCharacteristics	Contains binary flags that determine runtime features related to memory and execution behavior.
	SizeOfStackReserve	Declares the total memory reserved for the process stack.
	SizeOfStackCommit	Declares the portion of the stack that is initially committed.
	SizeOfHeapReserve	Declares the total memory reserved for the process heap.
	SizeOfHeapCommit	Declares the portion of the heap that is initially committed.
	LoaderFlags	Reserved field used for system-specific loading behaviors.
	NumberOfRvaAndSizes	Number of data directory entries present in the optional header.
Section Entropy	SectionsNb	Counts the number of sections defined within the binary.
	SectionsMeanEntropy	Measures the average entropy across all sections indicating data randomness.
	SectionsMinEntropy	Identifies the lowest entropy score among all sections.
	SectionsMaxEntropy	Identifies the highest entropy score among all sections.
Section Size	SectionsMeanRawsize	Reports the average disk size of all sections.
	SectionsMinRawsize	Records the smallest section size on disk.
	SectionMaxRawsize	Records the largest section size on disk.
	SectionsMeanVirtualsize	Reports the average size of all sections when loaded in memory.
	SectionsMinVirtualsize	Records the smallest memory size among sections.
	SectionMaxVirtualsize	Records the largest memory size among sections.
Imports	ImportsNbDLL	Counts the total number of dynamic link libraries used.
	ImportsNb	Counts all imported functions across all libraries.
	ImportsNbOrdinal	Counts functions imported using ordinal references rather than names.
Exports	ExportNb	Counts the number of functions made accessible to other modules.
Resources	ResourcesNb	Counts the embedded assets included in the binary.
	ResourcesMeanEntropy	Measures the average entropy of the embedded resources.
	ResourcesMinEntropy	Identifies the lowest entropy score across the embedded resources.
	ResourcesMaxEntropy	Identifies the highest entropy score across the embedded resources.
	ResourcesMeanSize	Measures the average size of the embedded resources.
	ResourcesMinSize	Records the smallest embedded resource by size.
	ResourcesMaxSize	Records the largest embedded resource by size.
Configuration	LoadConfigurationSize	Declares the byte size of the load configuration structure.
	VersionInformationSize	Declares the byte size of the file versioning metadata.

Table 6: Descriptive statistics for features (Malware class only)

Feature	Range	Mean	Std	Q1	Q3
Machine	332 – 34404	355.95	903.10	332.00	332.00
SizeOfOptionalHeader	224 – 352	224.01	0.597	224.00	224.00
Characteristics	2 – 49551	3.3K	9.3K	258.00	259.00
MajorLinkerVersion	0 – 255	8.57	4.82	8.00	10.00
MinorLinkerVersion	0 – 255	4.96	13.83	0.000	0.000
SizeOfCode	0 – 1818586738	176.9K	6.8M	37.9K	120.3K
SizeOfInitializedData	0 – 4294966272	518.6K	25.1M	119.8K	385.0K
SizeOfUninitializedData	0 – 4294940713	143.7K	19.5M	0.000	0.000
AddressOfEntryPoint	0 – 1074484297	172.3K	4.1M	14.8K	61.6K
BaseOfCode	0 – 2028711251	80.3K	6.6M	4.1K	4.1K
BaseOfData	0 – 268435456	223.0K	2.5M	45.1K	127.0K
ImageBase	65536.0 – 6442450944.0	13.1M	148.8M	4.2M	4.2M
SectionAlignment	16 – 134217728	8.3K	747.5K	4.1K	4.1K
FileAlignment	16 – 4096	588.85	519.26	512.00	512.00
MajorOperatingSystemVersion	0 – 36868	4.94	118.53	4.00	5.00
MinorOperatingSystemVersion	0 – 17757	1.18	92.65	0.000	1.00
MajorImageVersion	0 – 28619	3.32	190.06	0.000	0.000
MinorImageVersion	0 – 20512	2.86	168.51	0.000	0.000
MajorSubsystemVersion	3 – 6	4.70	0.461	4.00	5.00
MinorSubsystemVersion	0 – 47600	1.59	216.45	0.000	1.00
SizeOfImage	0 – 1410035712	827.1K	7.5M	311.3K	528.4K
SizeOfHeaders	448 – 786432	1.3K	6.5K	1.0K	1.0K
CheckSum	0 – 4294967295	256.1M	699.7M	300.4K	573.7K
Subsystem	1 – 3	2.01	0.076	2.00	2.00
DllCharacteristics	0 – 36864	28.7K	11.3K	32.8K	33.1K
SizeOfStackReserve	0 – 33554432	1.1M	484.2K	1.0M	1.0M
SizeOfStackCommit	0 – 2097152	5.5K	17.8K	4.1K	4.1K
SizeOfHeapReserve	0 – 13631488	1.1M	137.4K	1.0M	1.0M
SizeOfHeapCommit	0 – 2077323491	47.3K	9.4M	4.1K	4.1K
LoaderFlags	0 – 2328297507	51.5K	10.6M	0.000	0.000
NumberOfRvaAndSizes	7 – 3402309701	112.9K	17.5M	16.00	16.00
SectionsNb	1 – 40	5.25	1.81	5.00	5.00
SectionsMeanEntropy	0.0 – 7.98994115645	4.88	1.02	4.26	5.65
SectionsMinEntropy	0.0 – 7.98994115645	2.44	1.92	0.000	4.15
SectionsMaxEntropy	0.0 – 7.99999379205	7.38	0.732	6.59	7.96
SectionsMeanRawsize	64.0 – 1431641941.33	181.2K	9.2M	38.3K	101.1K
SectionsMinRawsize	0 – 2754048	5.8K	11.2K	0.000	9.7K
SectionMaxRawsize	64 – 4294885376	666.2K	35.9M	105.5K	330.8K
SectionsMeanVirtualsize	0.0 – 1431673348.0	183.7K	5.0M	46.3K	102.6K
SectionsMinVirtualsize	0 – 3399680	8.0K	20.1K	1.2K	9.4K
SectionMaxVirtualsize	0 – 4294884876	646.2K	15.3M	161.5K	339.7K
ImportsNbDLL	0 – 280	5.35	3.87	3.00	8.00
ImportsNb	0 – 3046	103.86	75.23	87.00	113.00
ImportsNbOrdinal	0 – 1051	1.43	11.11	0.000	1.00
ExportNb	0 – 8015	1.53	61.04	0.000	0.000
ResourcesNb	0 – 3060	13.99	32.64	6.00	14.00
ResourcesMeanEntropy	0.0 – 7.99972286753	4.15	1.23	3.50	4.45
ResourcesMinEntropy	0.0 – 7.99972286753	2.28	0.686	2.16	2.46
ResourcesMaxEntropy	0.0 – 7.99999295933	5.93	1.48	5.22	7.39
ResourcesMeanSize	0.0 – 2415919166.0	74.9K	9.3M	1.8K	12.7K
ResourcesMinSize	0 – 2415919166	25.7K	7.8M	48.00	48.00
ResourcesMaxSize	0 – 4294902624	323.4K	25.4M	7.3K	28.8K
LoadConfigurationSize	0 – 4294967294	664.6K	31.2M	0.000	72.00
VersionInformationSize	0 – 26	10.68	7.38	0.000	15.00

Table 7: Descriptive statistics for features (Legitimate class only)

Feature	Range	Mean	Std	Q1	Q3
Machine	332 – 34404	13.4K	16.6K	332.00	34.4K
SizeOfOptionalHeader	224 – 240	230.14	7.78	224.00	240.00
Characteristics	2 – 33679	7.2K	3.0K	8.2K	8.4K
MajorLinkerVersion	2 – 48	8.73	1.16	8.00	9.00
MinorLinkerVersion	0 – 60	1.16	3.51	0.000	0.000
SizeOfCode	0 – 51634176	396.5K	1.4M	7.2K	264.2K
SizeOfInitializedData	0 – 322908160	291.1K	2.2M	6.1K	112.6K
SizeOfUninitializedData	0 – 4197376	951.22	51.5K	0.000	0.000
AddressOfEntryPoint	0 – 45251088	171.3K	654.8K	4.6K	97.4K
BaseOfCode	0 – 3014656	5.1K	28.7K	4.1K	4.1K
BaseOfData	0 – 33599488	217.6K	939.5K	0.000	106.5K
ImageBase	65536.0 – 1.844673527761653e+19	1.8e+15	1.8e+17	268.4M	6442.5M
SectionAlignment	16 – 2097152	5.0K	21.1K	4.1K	4.1K
FileAlignment	16 – 65536	941.40	2.2K	512.00	512.00
MajorOperatingSystemVersion	0 – 10	5.46	0.806	5.00	6.00
MinorOperatingSystemVersion	0 – 3	0.891	0.679	1.00	1.00
MajorImageVersion	0 – 21315	221.90	2.1K	1.00	6.00
MinorImageVersion	0 – 20512	209.96	2.1K	0.000	1.00
MajorSubsystemVersion	1 – 10	5.24	0.838	5.00	6.00
MinorSubsystemVersion	0 – 50	1.11	1.77	0.000	1.00
SizeOfImage	3072 – 324210688	790.0K	3.1M	49.2K	540.7K
SizeOfHeaders	512 – 65536	1.4K	2.1K	1.0K	1.0K
CheckSum	0 – 4257612032	1.2M	38.6M	64.0K	488.7K
Subsystem	1 – 16	2.51	0.700	2.00	3.00
DllCharacteristics	0 – 49504	7.4K	13.4K	320.00	1.3K
SizeOfStackReserve	0 – 10000000	487.3K	447.6K	262.1K	1.0M
SizeOfStackCommit	0 – 1048576	5.4K	18.7K	4.1K	4.1K
SizeOfHeapReserve	0 – 4194304	988.5K	252.0K	1.0M	1.0M
SizeOfHeapCommit	0 – 65536	4.0K	1.6K	4.1K	4.1K
LoaderFlags	0 – 0	0.000	0.000	0.000	0.000
NumberOfRvaAndSizes	16 – 16	16.00	0.000	16.00	16.00
SectionsNb	1 – 38	4.39	2.01	3.00	5.00
SectionsMeanEntropy	0.173444442902 – 7.99555757619	4.06	1.05	3.25	4.83
SectionsMinEntropy	0.0 – 7.99555757619	1.83	1.51	0.334	2.96
SectionsMaxEntropy	1.11230271386 – 7.99999271595	5.96	0.984	5.78	6.54
SectionsMeanRawsize	333.714285714 – 92973226.6667	201.1K	1.1M	8.5K	114.7K
SectionsMinRawsize	0 – 29000704	51.7K	645.8K	512.00	4.1K
SectionMaxRawsize	512 – 321623040	614.3K	3.1M	26.6K	401.9K
SectionsMeanVirtualsize	226.0 – 64839490.2	199.5K	952.0K	8.4K	116.8K
SectionsMinVirtualsize	1 – 29000672	51.7K	645.8K	131.00	2.3K
SectionMaxVirtualsize	288 – 321622732	608.8K	2.7M	25.6K	407.9K
ImportsNbDLL	0 – 39	5.72	5.34	1.00	9.00
ImportsNb	0 – 4432	135.16	191.62	1.00	179.00
ImportsNbOrdinal	0 – 3450	11.15	60.70	0.000	4.00
ExportNb	0 – 16596	75.51	446.93	0.000	11.00
ResourcesNb	0 – 7694	40.93	243.39	1.00	8.00
ResourcesMeanEntropy	0.0 – 7.41281622478	3.66	0.648	3.41	3.92
ResourcesMinEntropy	0.0 – 5.14079953855	2.81	0.961	2.47	3.54
ResourcesMaxEntropy	0.0 – 7.99999997868	4.56	1.44	3.54	5.17
ResourcesMeanSize	0.0 – 3907290.66667	9.8K	89.9K	832.00	2.0K
ResourcesMinSize	0 – 7224	528.41	433.61	126.00	928.00
ResourcesMaxSize	0 – 312479744	66.8K	1.7M	928.00	9.6K
LoadConfigurationSize	0 – 160	30.78	40.25	0.000	72.00
VersionInformationSize	0 – 26	16.31	2.15	16.00	17.00

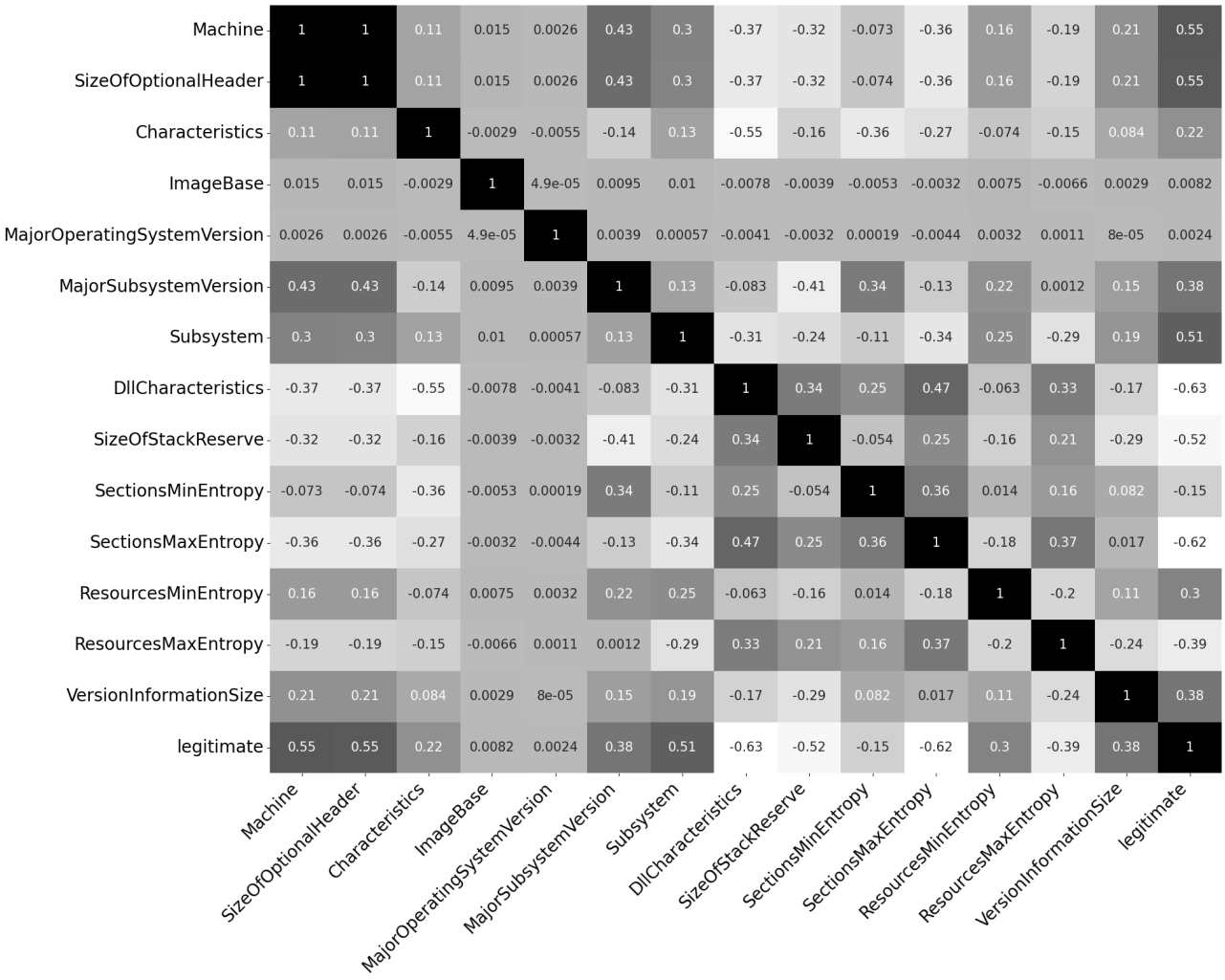


Figure 11: Correlation matrix of features selected by Extra Trees

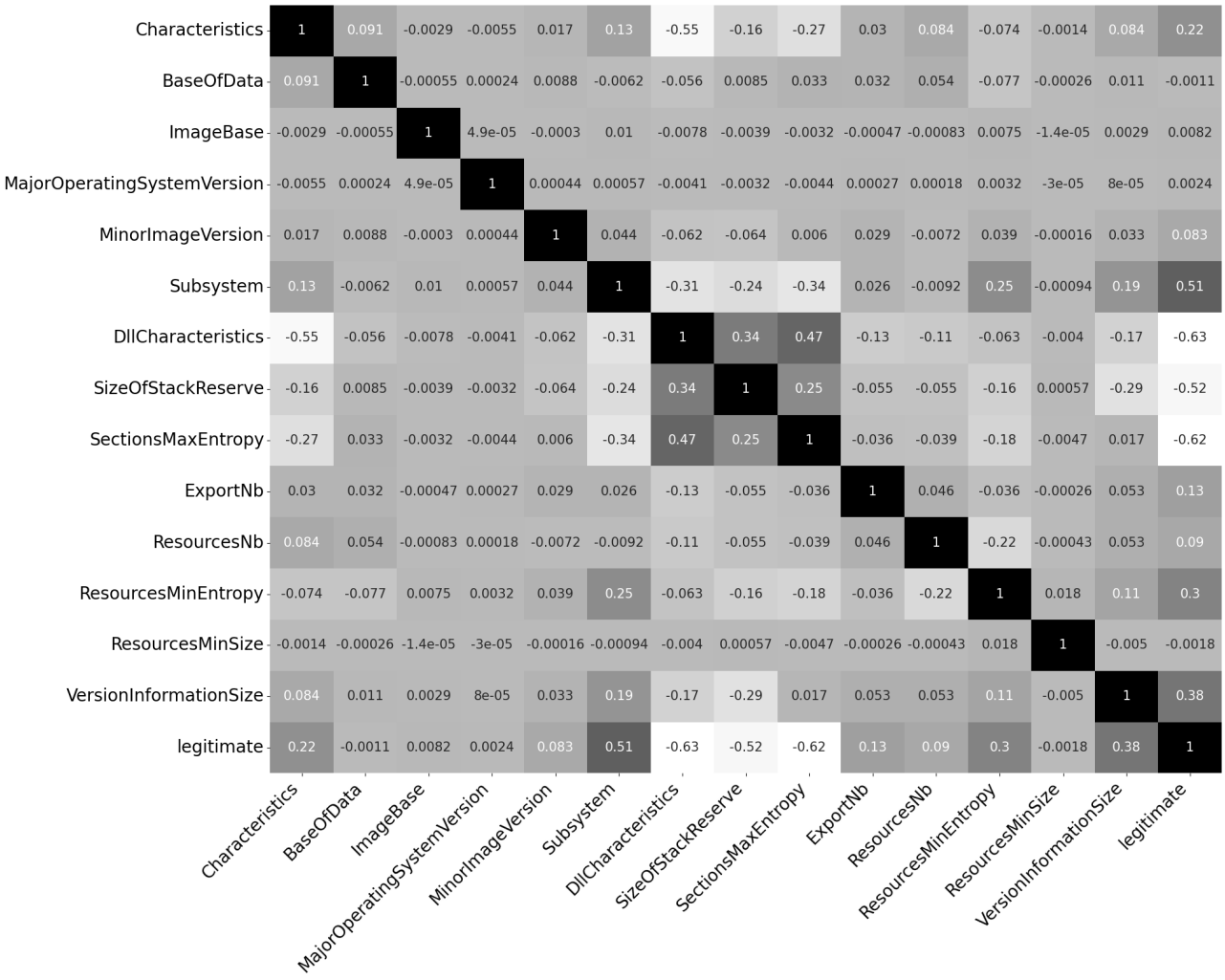


Figure 12: Correlation matrix of features selected by Random Forest