Analysis of Machine Learning Approaches to Packing Detection

Charles-Henry Bertrand Van Ouytsel[a,*], Khanh Huu The Dam[a], Axel Legay[a]

[a]*INGI, ICTEAM, Universite Catholique de Louvain, Place Sainte Barbe 2, LG05.02,01, 1348 Louvain-La-Neuve, Belgium*

Abstract

Packing is a widely used obfuscation technique by which malware hides content and behavior. Much research explores how to detect a packed program via such varied approaches as entropy analysis, syntactic signatures, and, more recently, machine learning classifiers using various features. Yet no robust results indicate which algorithms perform best or which features are most significant. Reviews of these results highlight how accuracy, cost, generalization of capabilities, and other measures complicate evaluations. Our work addresses deficiencies by assessing nine different machine-learning approaches using 119 features to identify which features are most significant for packing detection, which algorithms offer the best performance, and which algorithms are most economical.

Keywords: Malware, Machine Learning, Packing, Features analysis, SHAP values, Experimental Comparison, Adversarial attack

1. Introduction

Malware detection represents a critical and expensive problem for current computer security. Most malware detection programs are based on signatures and thus reactive. Unfortunately, new malware are created constantly. VirusTotal [4] registers 600,000 new pieces of malware (malicious programs) detected daily on average. This situation privileges writers of malware over detection.

Packing is a widely used technique to evade or disrupt malware analysis. Packing operates via various methods, e.g., compressing or encrypting a program's sections, encoding a program in a virtual machine, or fragmenting a program's behavior. These packing methods make malware analysis more difficult as they obfuscate static features (i.e., features extracted without running the binary) and make the behavior of the sample, observed with dynamic analysis, more complex and harder to analyze. While benign programs typically use packing to compress programs or to protect intellectual property, VirusTotal [4] states that most new malware samples are merely re-packed versions of previously seen malware.

An effective malware detection or classification engine must detect whether a program under analysis is packed. A packed program can be unpacked or treated with greater suspicion to improve analysis and response. Combined with a fast and efficient static malware detector such as that proposed by Baldangombo et al. [9], a complete malware detection engine offers high-end performance malware detection and classification.

*Corresponding author

Email address: charles-henry.bertrand@uclouvain.be (Charles-Henry Bertrand Van Ouytsel)

Syntactic signatures are a popular approach to packer detection since there are easy to compute and can achieve high accuracy with known packers. For example, to build signatures, YARA [54] typically considers simple static features such as byte sequences, strings, or file features such as file size or hash. Although such signatures are effective for known packers, they are easily bypassed by unknown packers [14]. Using entropy is also effective in detecting packing since packed files *often* have higher entropy than unpacked files. Lyda et al. [32] were the first to address the problem of packing detection using entropy in an academic publication. Although studies like [25] improved this detection method, [52] shows that inserting a selected set of bytes in an executable maintains a low entropy. Mantovani et al. [33] demonstrates that most advanced methods (i.e., machine learning models based on several features) do not perform well on low entropy packing schemes. Finally, different works proposed heuristics-based techniques to build their detectors. For example, Choi et al. [18] proposed PHAD, a packing detector based on eight features that include the number of executable and writable sections, the number of section names that are not printable, the sum of the size of every section compared to file size, or the executable permission of entry point section. Treadwell et al. [48] highlight the importance of IAT table, TLS section, and specific section names in their detector. This wide range of features, extracted statically with low computation cost, is widely used as the basis for popular antiviruses such as ClamAV [1] or tools such as PEiD [3].

To improve packing detection, different work has used machine learning techniques involving supervised learning on a variety of features extracted statically. Perdisci et al. [40] train different models (e.g., multiple layer perceptron, naive Bayes model, etc.) on nine features related to section properties, IAT table, and entropy. Ugarte-Pedrero et al. [51] use the same features (adding n-grams) and train different models and rules-based classifiers concentrating on anomaly detection. Following a literature survey, Biondi et al. [13] extracted 119 features and divided these into six categories: Byte entropy, Entry bytes, Import functions, Metadata, Resource, and Sections. They then trained naive Bayes models and distinct tree models with different ground truths, investigating which models offer the best trade-off between accuracy and training cost. Mantovani et al. [33] and Aghakhani et al. [5] further highlight limitations of current machine-learning approaches. The former illustrates the difficulties of ML models in detecting low entropy packing schemes; the latter shows the overlap between packing detection and malware detection, based on static features, in current ML models. Appendix A includes a detailed comparison of existing ML approaches for packing detection. Recent investigations into unsupervised learning [38] look at how to build clusters related to packers to identify variants of known packers or new packers. While interesting, this is beyond the scope of our work here; for this paper, we study supervised learning algorithms. Finally, as explained in [46], recent studies illustrate that machine learning models tend to learn spurious correlations from data, pushing some features as more decisive than they might be. In the context of vulnerability discovery, the authors suggest seeking a better understanding of feature importance via explainability methods.

Unfortunately, there is a lack of knowledge about the impact of all the proposed features on the decision processes of machine learning models. No conclusive evidence cites which existing model is more efficient and robust. Costs related to different models and their features are uncertain. While a recent survey [49] introduces economical analyses related to malware detection, aiming to present existing trade-offs between analysis accuracy, time, and cost when deploying a malware analysis infrastructure, researchers have proposed limited analyses related to re-training costs for packing detection [13].

In this paper, we chose to evaluate the features set proposed in in [13] because this gathers a wide range of features well-known in the literature. We extend this study by examining the

importance and impact of these features using nine different classifiers with multiple strategies: variance, mutual information, Shapley values, and feature selection. We compare and contrast the nine supervised machine learning classifiers for effectiveness and resilience against adversarial attacks, and we looked at deployment costs. We can propose the following research contributions:

- **Feature analysis.** We deploy different methods to explore the importance of 119 features in packing detection, examining variance and mutual information of the features and Shapley values. We then reduced the feature set using features selection methods, principal component analysis, or Shapley values selection methods.
- **Classifiers effectiveness.** We study the effectiveness of nine supervised machine learning classifiers applied to a large dataset of samples observed 'in the wild' and several binaries we packed ourselves. We use Wilcoxon tests to ensure the significance of algorithm comparison.
- **Adversarial examples.** We generate adversarial examples against our classifiers by applying simple modifications on packed binaries. This allows us to study classifiers' robustness and better understand limitations.
- **Economical analysis.** We then train and evaluate our classifiers using different samples collected chronologically. An economical analysis (adhering to the approach proposed in [49] and following the work of [13]) highlights trade-offs per classifier concerning accuracy, efficiency, training time, and memory.

The structure of the paper is as follows: Section 2 reviews background on executables, packing, machine learning, feature importance/selection and assessment methodology. Section 3 presents dataset composition and related labeling. We describe our methodology for the study of the 119 features and the nine algorithms. We explain the configuration we use for our ML algorithms (i.e., pre-processing and hyper-parameter tuning.) Section 4 investigates the relevance of different features in the decision process of classifiers, including SHAP values computation, feature selection methods, and principal component analysis. Section 5 evaluates the ML algorithms for effectiveness, relevant limitations, and robustness against simple adversarial attacks. Section 6 explores economical trade-offs for the different classifiers and tests resilience using a chronological dataset. Finally, Section 7 concludes with discussions of our results' validity, inherent limitations, and further work.

2. Background

This section covers background information on packing, machine learning, and metrics.

2.1. Portable Executable Format (PE)

The Portable Executable (PE) format is a file format used by Microsoft Windows. PE files are the primary target of malware authors and the primary means of distributing malicious software on Windows [4]; it is prudent to focus packing detection studies on this file format. As shown in Figure 1, a PE file can be separated into two categories: Header and Sections. The header category encapsulates information about the executable, such as the file structure, which includes timestamp, number of sections, number of symbols, etc. In the header category, the Import Address Table (`IAT`) is a call-table that includes the addresses of imported library functions, and the *Section Table* contains descriptions of the sections found in the Sections category. The Sections category

comprises the code and data of the executable. A PE file could contains sections such as the `.text` section with the actual code of the program, the `.data` section used to declare initialized variables, and the `.bss` section that contains the uninitialized variables.

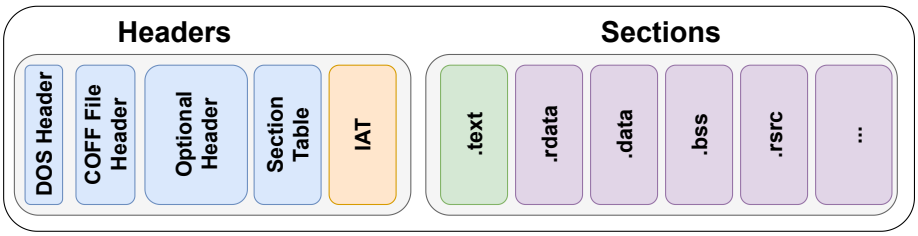


Figure 1: PE File Format Structure

2.2. Packing

A packing program, or packer, is a program that compresses/encrypts an executable, thus creating a new executable by bundling the original compressed executable with its decompression/decryption code. When executed, the packed executable first executes a decompression/decryption routine and then passes control to the rebuilt executable. Typical packing techniques are implemented with a small stub of the program that performs the initial decompression/decryption or runs a virtual machine to allow the program to execute.

Figure 2 illustrates the life-cycle of a typical packed program. First, the original program is transformed by obfuscating its content (e.g., compression, encryption, virtualization, erasing imports, junk code, etc.) Then an unpacking stub, containing the routine to recover the original executable file, is added. The stub becomes the new entry point of the executable. When the program is executed, the unpacking routine permits the reconstruction of the original executable in the process memory. The IAT table is rebuilt to enable the executable to call functions from dlls, and the control flow finally passes to the original entry point of the executable.

Taxonomy. A recent survey [36] describes four main types of packers:

- Compressors: Compressors compress sections of the original file to reduce the size of the original executable. Generally, compressors do not insert anti-reversing tricks in packed files.
- Crypters: Crypters encrypt/obfuscate sections of the executable file and decrypt these when the unpacking stub is executed. This makes the file difficult to analyze statically.
- Protectors: Protectors combine the capabilities of both compressors and crypters.
- Bundlers: Bundlers combine multiple executable and data files into a single bundled executable that can access these multiple files without writing them to disk.

While packers like compressors usefully reduce the size of an executable, adversaries can leverage packing properties to conceal code. Since packing an executable changes the file's signature and obfuscates its content, packing can help evade static analysis. This forces analysts to unpack the executable if they want to deploy static analysis tools. Anti-unpacking techniques incorporated by the packer can also hinder dynamic analysis tools (anti-debugging [24], anti-emulation [23], etc.)

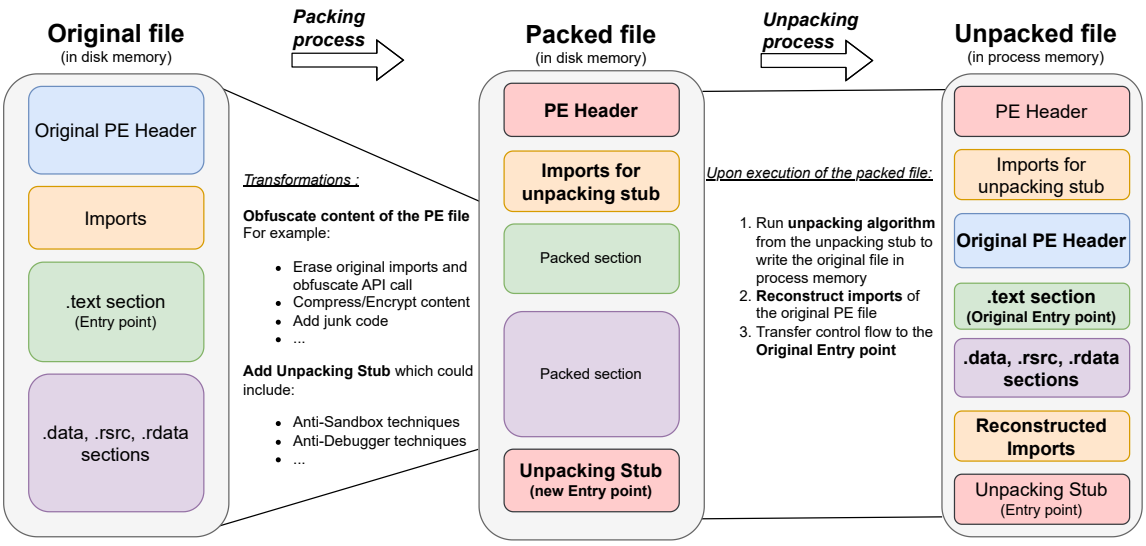


Figure 2: Life-cycle of a packed program

and impact the effectiveness of malware detectors. In [50], the authors discuss the different packers' complexities, ranging from I to VI. While the simplest packers (i.e., type I, such as UPX) execute a single unpacking routine before transferring the control flow to the unpacked executable, other packers may contain multiple unpacking layers organized in a complex topology (i.e., type III, such as UPolyX). Finally, the most complex packers can unpack a single fragment of the original executable at a time (i.e., type VI, such as Armadillo).

While this paper focuses on a single packer used on an executable, multiple packers are often used on the same executable. A goal of this "multi-stage packing" is to evade signature-based detection. Malware is harder to detect when a well-known packer is used to achieve a high compression ratio in conjunction with a custom packer that is unknown to the antivirus software. We next analyze three well-known packers with different capabilities.

UPX. The Ultimate Packer for eXecutables (`UPX`) is an open-source packer that supports Windows Programs/DLLs, DOS programs, and Linux executables/kernels. This packer works by adding three sections to an executable: **UPX0**, an empty section acting as an area where the decompressed code will go; **UPX1**, the entry point section where the decompression code will go; and **UPX2** or .rsrc, containing resources such as `IAT`. When an UPX-packed executable is executed, the content of section UPX1 (the entry point) is executed first to decompress the original program into the UPX0 section. The execution flow then jumps into the section UPX0, and the original program is executed. This process uses predictable section names, so looking for sections called UPX0 or UPX1 can detect if an executable was packed using the `UPX` packer.

Yoda's protector. Yoda's protector is an open-source packer that supports Windows programs/DLLs and is based on *Yoda's Crypter*. This packer has the ability to compress or encrypt each section of an executable while preserving the section's structure. It adds its unpacking stub to a new section **.yP**. It supports several protection features: anti-reversing tricks (e.g., CRC check to ensure the

packed executable is not modified by an analyst), removing debug information, obfuscating API by IAT redirection (only those exported by *kernel32.dll*, *user32.dll* and *advapi32.dll*), and similar.

BobSoft Mini Delphi. Written in Delphi, BobSoft Mini Delphi implements a wide range of checks to detect if the packed executable runs in a monitored environment (e.g., in a virtual machine). Checks include cursor movements, running processes, processor information, etc. The original executable is divided into many parts, encrypted, and stored in different resources sections.

2.3. Machine Learning

Machine Learning (`ML`) is a branch of artificial intelligence (AI) based on the assumption that systems can use data to learn and make decisions. `ML` algorithms build a model from which to learn based on a set known as training data. There are three broad categories of learning types: Supervised learning, Unsupervised learning, and Reinforcement learning. All the classifiers of our study rely on supervised learning, which consists of using an already labeled data set to train the model. We will now explain common terminology in machine learning.

Over-fitting. is a common problem in machine learning in which a model exactly fits its training data and is thus incapable of performing accurately with new data. Instead, the model captures noise and results in poor generalization performance.

Bagging. is a technique that combines multiple base classifiers (e.g, decision trees) to improve classification performance. To produce these base classifiers, different data sets are created by drawing randomly multiple points from the original data set.

Boosting. combines multiple base classifiers trained **sequentially**. Each base classifier is trained to improve using the performance of previous classifiers. Boosting gives greater weight to misclassified data points to improve their classification when using subsequent base classifiers.

Types of models. The classifiers studied in this work use six types of models to make predictions:

KNN. The *K-Nearest Neighbors* (KNN) is a learning algorithm that treats each data point input as a point in the feature space with a particular label. When first given a data point to classify, KNN finds the K-nearest points in the feature space in terms of a defined distance (e.g., Euclidean distance). The data point is then assigned a class label based on the majority class of its K nearest neighbors. While KNN requires no training time and is simple to implement, the speed of this algorithm declines significantly as the dataset grows.

Probabilistic models. Naive Bayes classifiers (NBC) are probabilistic models from Bayes theorem:

$$\Pr(y_k|x_i) = \frac{\Pr(x_i|y_k)\Pr(y_k)}{\Pr(x_i|y_k)\Pr(y_k) + \Pr(x_i|\neg y_k)\Pr(\neg y_k)} \tag{1}$$

where y_k is a class in the set of classes $k \in \{1, .., K\}$ and x_i is a feature in the set of feature vectors $i \in \{1, .., n\}$ and $\Pr(a)$ is the probability of a (where a could be a class label or feature vector) and $\Pr(a|b)$ is the probability of a given b (where a is a class label and b is a feature vector or vice versa). These classifications use Equation 1 to evaluate probability to observe label y_k for a sample according to the value of feature vector x_j. These estimated probabilities are then combined to construct the classifier $\hat{y}$ given in Equation 2:

$$\hat{y} = \underset{k \in \{1,..,K\}}{\arg\max}\ p(y_k) \prod_{i=1}^{n} p(x_i|y_k) \tag{2}$$

The naive adjective of the classifier comes from the assumption that each data point's feature is independent. While this assumption is generally invalid, NBC is known to work well in practice [15]. In this paper, we deploy variants of NBC such as *Gaussian Naive Bayes Classifier* GNBC.

Linear Models LIN. There are classification models built as a linear combination of features in which the decision boundary $z(x)$ can be represented as a line or a plane that separates the data into two classes. Consider Equation 3 with features values x_i and coefficients β_i.

$$z(x) = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p \tag{3}$$

Following Equation 4, this decision boundary can construct the classifier:

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Since LIN comprises a very large family of algorithms, this paper looks at only two forms of LIN: logistic regression and linear support vector machines. *Logistic regression* (LR) is based on the logistic function[1] used to modify the decision boundary. *Linear support vector machine* (LSVM) aims to maximize the margin between two parallel hyper-planes that separate the data classes.

Tree-based Models. Decision trees (DT) are classification models based on the sequential application of simple binary rules. 'Trees' are models that start at the 'root' and grow 'leaves.' A 'leaf' is a node without any children nodes. Each leaf gains a label according to a simple rule (e.g., "if X_1 of the sample is smaller than one, go to the right branch; if not, go to the left branch"). When combined, simple DT models construct more complex classifiers, solving the variance and bias from which single DT can suffer. *Random Forests* (RF) combine slightly different DT with bagging. While a single DT can be effective, there is a risk of overfitting. RF addresses this risk by incorporating outcomes from multiple DT to reduce over-fitting while maintaining each tree's predictive power. *Gradient Boosted Decision Trees* (GBDT) combine DT using *boosting*. A GBDT often starts with extremely shallow trees, called weak learners, which provide good predictions only for some parts of the data. More trees grow progressively to increase global performance by reducing the loss margin. Compared to RF, GBDT is more efficient with memory but harder to tune.

Neural networks. (NN) are used often in deep learning approaches and employed in hundreds of different classifiers for security tasks like anomaly detection or malware classification. [37, 28]. We use the most popular NN variation: *Multi-Layer Perceptrons* (MLP) algorithm. MLP can be viewed as a generalization of LIN that comprises various layers of neurons. Each neuron is treated as a unit that accepts its input, applies a weighted sum of its features, and passes this through a non-linear function before forwarding the modified input to the subsequent layer. Each neuron of each layer with input features X will apply the following equation to calculate its output y:

$$y = f(x[0] \cdot w[0] + w[1] \cdot x[1] + ... + w[p] \cdot x[p] + b) \tag{5}$$

[1] $\sigma(z) = \frac{1}{1+e^{-z}}$

where f is a non-linear function (e.g., hyperbolic tangent) applied to a weighted sum on x features. With MLP, this process of weighted sums is repeated multiple times. Data are supplied to the input layer where there may be one or more hidden layers representing intermediate processing steps, before eventually, the output layer provides predictions.

NN have demonstrated efficiency with large data sets and created incredibly complex models for only the cost of training time, wise parameter tuning, and accurate pre-processing.

Kernelized Support Vector Machines. (KSVM) are an extension of LIN that uses hyperplanes to handle non-linearly separable data. In KSVM, lower dimensional data are converted to higher dimensional data via a *kernel function* (e.g., polynomial, Gaussian, Sigmoid, etc.) Planes used to separate the data become higher-dimensional hyperplanes. While KSVM work well on various datasets, they can suffer from performance degradation when scaled with larger sample numbers.

Interpretable Machine Learning. To better understand and explain a model decision, interpretable machine learning has increased in popularity [35]. Techniques in this domain help pinpoint the impact of each feature on the decision process of a model. One of the techniques, the Shapley values, has an interesting approach that explains the independent contribution of each feature to the outcome. In the Shapley values approach, each feature is assumed to be a 'player' in a game where the prediction is the payout. The objective is to attribute each part of the payout (i.e., prediction) to a feature. Let S be a subset (i.e., coalition) and x be the vector of feature values of the instance to be explained. Define $val_x(S)$ as the prediction for marginalized feature values in coalition S over features that are not included in set S :

$$val_x(S) = \int \hat{f}(x_1, ...x_p) d\mathbb{P}_{x \notin S} - E_X(\hat{f}(X)) \tag{6}$$

The Shapley value related to feature j is its contribution to the prediction, weighted and summed over all possible feature combinations:

$$\phi_j(val) = \sum_{S \subseteq \{1,...,p\} \setminus j} \frac{|S|!(p - |S| - 1)!}{p!} (val(S \cup \{j\} - val(S)) \tag{7}$$

where p is the number of features.

Since evaluating Shapley values requires significant computation time, most real applications use approximations like **SHAP values**. SHAP stands for SHapley Additive exPlanations and represents the Shapley value as a linear model. [35] provides in-depth details.

2.4. Dimensionality reduction and feature selection

Variance filtering. The variance $Var(.)$ is a statistical measure that describes the variability of a feature X and is defined as :

$$Var(X) = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - E(X))^2 \tag{8}$$

where n designates the number of data points, X_i is the feature value of data point i, and $E(X)$ is the mean value of the variable X. Examining the variance of each feature allows us to discover if there are features that do not vary in the dataset (i.e., the variance around 0). Filtering those features discards features bringing limited information to the model.

Mutual information. This statistical measure $I(.,.)$ quantifies the amount of information shared by two variables. This allows us to measure the dependence between these variables using the concept of entropy (i.e., the measure of uncertainty of a variable). The mutual information between a feature X and the label of the data point Y is defined as:

$$I(X;Y) = H(X) - H(X|Y) \tag{9}$$

where $H(X)$ is the entropy of X, and $H(X|Y)$ is the conditional entropy of X given Y (i.e., the uncertainty of X when Y is known). Mutual information allows the discovery of nonlinear relationships between variables and is often used to select features according to their relevance.

Principal component analysis (PCA). This algorithm reduces high-dimensional data by projecting it linearly while preserving as much data variance as possible. From a normalized dataset $D \in \mathbb{R}^{n \times d}$ where n designates the number of data points, and d describes the dimension of the data, it first calculates the covariance matrix $cov(.)$ of D like so:

$$cov(D) = \frac{1}{n-1} \sum_{i=1}^{n} (D_i)(D_i)^T \tag{10}$$

It then computes the eigenvectors and corresponding eigenvalues of $cov(X)$. The k eigenvectors with the largest eigenvalues are selected as *principal components*, and the dataset is projected onto selected principal components.

Features selection. This approach reduces the machine learning input dimension by removing irrelevant/redundant features and preserving only relevant features [16]. Reasons for deployment include reducing feature extraction time, improving training time, decreasing memory usage, simplifying the models, or avoiding the curse of dimensionality [15]. Three categories of features selection methods include: filter, wrapper, and embedded. Filter methods do not take into account machine learning models' characteristics (e.g., variance analysis). Wrapper methods use model predictions to select features (e.g., Recursive Feature Elimination), and embedded methods include feature selection in the training process (e.g., Lasso for linear model [35]). On behalf of variance analysis and mutual information, we deploy two feature wrapper selection methods. Recursive Feature Elimination (RFE) recursively eliminates features of less importance (i.e., features given less weight in a model or features less present in the tree-based model). This ensures we focus on smaller but more pertinent sets of features. SHAP selection method [34] calculates SHAP values to order features by importance and then trains different models by gradually considering the best features for training.

This feature selection directly reduces the dimensionnality of our dataset by removing redundant/useless features while PCA reduces dimensionality by creating new features.

2.5. Assessment methodology

Terminology. In our packing detection problem, **false positives** are non-packed binaries classified as packed by a classifier. Conversely, **false negatives** are packed binaries classified as non-packed by a classifier. Finally, true positives (as well as true negatives) are packed (or non-packed, respectively) binaries correctly classified.

Performance metrics. We evaluate and compare different classifier performances using the metrics:

- The **precision** metric is defined as the ratio of samples correctly labeled *packed* over all samples labeled *packed*. In a typical malware analysis workflow, a *packed* sample will likely require pre-processing with an unpacker. It is important to limit false positives to avoid overflowing the unpacker.
- The **recall** metric is the ratio of samples correctly classified as *packed* over all *packed* samples. Optimizing the recall limits false negatives and ensures proper analysis of the real content of packed samples.
- The F**-score** metric is defined as $\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$. The F**-score** is a harmonic mean that represents both precision and recall symmetrically in one metric.

Receiver Operating Characteristics (ROC) curve. This curve represents the performance of a binary classifier according to different classification thresholds. The classification threshold is the value used to convert the output of a binary classifier into the predicted label. The ROC curve is created by plotting the true positive rate or the TPR, also known as *recall*, versus the false positive rate, or the FPR. The FPR is the ratio of false positives compared to all non-packed binaries. Different ROC curves are in Figure 3. A diagonal represents a totally random classifier since it equally classifies a sample as packed or not packed (i.e., leading to the same TPR and FPR). The more the curve skews above the diagonal, the more the classifier effectively distinguishes between packed and not packed. A perfect classifier would have a TPR of 1 with no false positives.

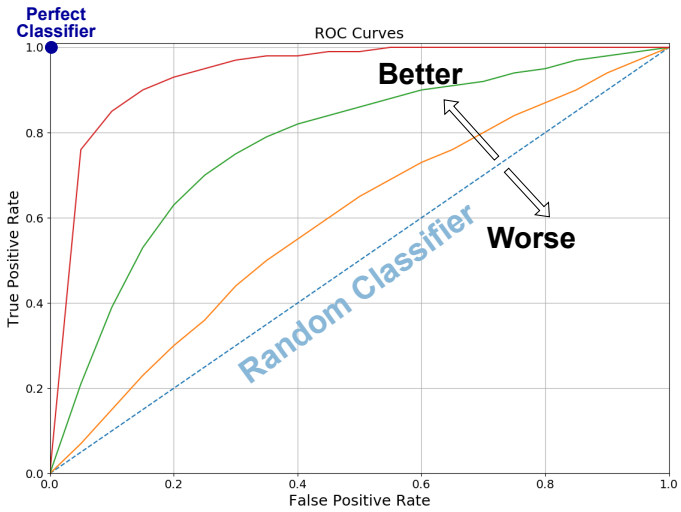


Figure 3: Illustration of different ROC curves

The ROC curve illustrates the trade-off between the TPR and the FPR and allows a more precise comparison of different classifiers' performance.

The Wilcoxon rank sum test. This statistical significance test compares the classification performance (e.g., accuracy/F-score) of ML algorithms. The test assesses if two independent sets of measures M (i.e., the performance of two algorithms on different datasets) come from identical continuous distributions with equal medians versus an alternative in which they do not have equal medians. This verifies if an algorithm is significantly more efficient than another example [26].

The different performance of the two classification algorithms A and B according to the measure M on the dataset X is given by:

$$\delta(M, X) = M(A, X) - M(B, X)$$

Differences are calculated for each dataset X and assigned a rank corresponding to their place in ascending order. Ranks related to positive (and respectively negative) differences are summed. Finally, the smaller of the two values serves as the output value of the Wilcoxon test. This value deduces the p-value associated with the test and the (non-)superiority of one algorithm over another. The smaller the p-value, the stronger the indication that there is a significant difference between algorithms A and B. The outcome is decided by a threshold α, called the significance level, i.e., $\alpha \leq 0.05$. If p-value $< \alpha$, then there is no difference between the algorithms. In our experiment, we use the one-sided Wilcoxon rank sum test that is available in the scientific computing library SciPy[53] and takes the significance level $\alpha = 0.05$ which is a popular value in the literature [55].

3. Methodology

This section details the three phases of our experimental setup: (1) dataset constructions; (2) experimental methodologies; and (3) tuning ML algorithms.

3.1. Data Sets and Labelling Tools

All datasets derive from a malware feed provided by Cisco that offers 1,000 new malware samples per day as these samples are found **in real-time online**. These malware samples were collected from 2019-09-17 to 2020-07-29. Occasional errors in the extraction software require the exclusion of an insignificant number of samples; we can safely ignore these. For each sample included in our work, we extract the 119 static features identified in [13].

Samples are labeled as packed or not packed to obtain a ground truth to train ML classifiers. We took a similar approach to VirusTotal; we employed multiple detector engines for classification and then built on these results. We used the four following classification engines:

- **PEframe** [6] is an open-source tool written in Python and freely available to perform static analyses of portable executable (PE) files, object code, DLLs, etc. We used the tool for packer detection, but it is possible to deploy it to obtain other digital forensics information such as macros, anti-forensics techniques, etc.

- **Manalyze** [31] is a static analyzer composed of several plugins frequently used by large enterprises. Its packer detection plugin works through signatures and custom heuristics. Signatures are generally related to the names of PE sections. For example, UPX renames some sections after packing as UPX0, UPX1, etc. Heuristics are more related to entropy-based features or anomalies in the PE structure, such as unusual section names or sections that are both writable and executable, etc.

- **PEiD** [3] is a free, well-known signature-based detector. Relevant signatures contain low-level byte patterns that can match either at the entry point or anywhere within a PE file.
- **DIE** Detect-it-Easy [2] is an open-source application allowing users to write signatures in a JavaScript-like language.

When classifying a sample, a key metric is the number of classifiers that label the sample as packed, or the threshold of detectors that regard a sample as packed. In [13], the authors build different ground truths based on classical packing detectors. Their first ground truth uses the threshold of one detector, and their second ground truth uses a threshold of three detectors. They find that a ground truth based on a large consensus (three detectors) reduces the number of packing families and improves model performance in a lab environment. However, this methodology generalizes poorly when applied to new packers. The ground truth built using a threshold of one detector proves to be more flexible and allows the presence of more myriad packers. Our dataset is thus labeled using a ground truth built using a one-detector threshold. While our experiments usually require identifying when a sample is packed or not, we also need to know which packer is used to investigate false negatives. If a conflict between labels proposed by our detector arises, we manually investigate the false negative to identify its label.

First, we use the dataset BIG of 168,410 samples for the broader experimental results. This data set consists of samples collected from 2019-09-17 to 2020-04-30 and, according to the labeling, has 94,621 packed samples and 73,789 not packed samples. The distribution of packers detection per each detector is visible in Figure 1, and Table 3 shows malware families.

Packers	PEiD	DIE	Manalyze	PEframe	Packers	PEiD	DIE	Manalyze	PEframe
ah	2928	0	0	2905	nullsoft	1863	0	0	1859
armadillo	65	0	11	1036	obsidium	1	0	1	20
ascrypt	0	0	17	0	packman	224	224	0	0
aspack	3560	5717	3990	3508	pe diminisher	63	0	0	0
asprotect	75	0	9	564	pecompact	1827	942	870	975
bobsoftminidelphi	1270	0	0	0	peninja	0	0	380	0
crypto-lock	72	0	0	0	pestuboep	0	0	3789	686
d1s1g	0	0	195	0	petite	1	0	1	1
dxpack	0	6202	0	0	pklite	8	17	7	9
enigma	66	0	39	0	polyene	0	0	76	0
fsg	166	68	63	246	protect shareware	0	0	55	0
inno setup module	11	0	0	0	safeguard	248	0	0	234
kbys	19	20	54	0	ste@lth pe	104	0	0	94
kkrunchy	271	0	173	259	telock	807	0	645	903
mew	1	3	0	3	themida	1	0	1	2
molebox	3	0	13	0	upack	125	152	27	124
mpress	0	2388	0	1179	upolyx	0	0	760	0
neolite	139	138	0	5	upx	15134	70665	23685	14959
nspack	21	58	15	26	yoda's protector	0	0	0	1472

Table 1: Detection made by the different packing detectors on data set BIG.

We use a second dataset CHRONO with 45,795 samples for an economical analysis of our classifiers. We compare long-term accuracy, memory, and time required for training/testing. This dataset consists of samples collected from 2020-05-01 to 2020-07-29 and, according to the labeling, has 22,071 packed samples and 23,724 not packed samples. The dataset is divided into four subsets, each corresponding to a separate time period: CHRONO 1 designates malware obtained from

2020-05-01 to 2020-05-14. CHRONO 2 designates malware obtained from 2020-05-15 to 2020-05-28. CHRONO 3 designates malware obtained from 2020-07-01 to 2020-07-13. CHRONO 4 designates malware obtained from 2020-07-14 to 2020-07-29. The distribution of packer detections can be seen in Table **??**, and Table 3 shows the malware families detected by AVClass. As with the BIG dataset, half of the samples are not detected as belonging to any particular family.

Packers	PEiD	DIE	Manalyze	PEframe	Packers	PEiD	DIE	Manalyze	PEframe
ah	1477	0	0	1443	nullsoft	211	0	0	210
armadillo	3	0	3	125	obsidium	0	0	0	31
ascrypt	0	0	3	0	packman	23	23	0	0
aspack	946	1100	1081	945	pe diminisher	9	0	0	0
asprotect	5	0	0	59	pecompact	166	42	29	43
bobsoftminidelphi	405	0	0	0	peninja	0	0	37	0
crypto-lock	16	0	0	0	pestuboep	0	0	175	118
d1s1g	0	0	16	0	petite	0	0	1	0
dxpack	0	412	0	0	pklite	1	2	0	0
enigma	3	0	1	0	polyene	0	0	22	0
fsg	77	1	2	36	protect shareware	0	0	17	0
inno setup module	5	0	0	0	safeguard	34	0	0	39
kbys	0	0	0	0	ste@lth pe	81	0	0	166
kkrunchy	24	0	34	24	telock	605	0	376	604
mew	0	0	0	0	themida	0	0	0	0
molebox	1	0	8	0	upack	42	44	7	42
mpress	0	363	0	129	upolyx	0	0	192	0
neolite	0	0	0	3	upx	3675	13762	3093	3767
nspack	1	2	1	2	yoda's protector	0	0	0	654

Table 2: Detection made by the different packing detectors on data set CHRONO.

BIG	#	CHRONO 1	#	CHRONO 2	#	CHRONO 3	#	CHRONO 4	#
dinwod	5833	virlock	1498	virlock	1633	mintluks	1164	mintluks	700
razy	4632	zusy	657	zusy	872	virlock	491	virlock	465
mansabo	4316	upatre	497	berbew	593	berbew	340	mansabo	411
sytro	3079	berbew	393	delf	396	mansabo	285	berbew	286
berbew	3023	delf	385	wapomi	359	delf	265	wabot	271
upatre	3001	genkryptik	379	sillywnse	270	wapomi	255	zusy	252
eati	2644	wapomi	313	genkryptik	255	zusy	220	sfone	248
banload	1987	downloadguide	290	blackmon	193	upatre	194	drolnux	225
ipamor	1976	dinwod	171	upatre	181	stihat	185	upantix	212
wacatac	1231	sytro	133	dinwod	153	sfone	184	upatre	168

Table 3: Top malware families recognized in BIG data set and CHRONO data set according to AVClass [44].

3.2. Experimental methodology

All ML algorithm implementations used are from the scikit-learn library [39] version 0.23.2. All experiments are performed on a desktop PC with an Intel Core i7-8665U CPU (1.90GHz x 8) and 16GB RAM running Ubuntu 18.04.5. Our methodology is as follows:

First, we explore which features are the more significant in different ML algorithms. We use different approaches, notably inspecting variance, mutual information, and Shapley values. We reduce the dimensionality of the data using the K-best features selection and principle components analysis. These approaches were applied to all nine ML algorithms using the Big dataset. The Big dataset is divided using 70% of the data for training and 30% for experimental evaluation of the model after hyper-parameters tuning and dimensionality reduction. The split is stratified to ensure the same proportion of packed and unpacked samples between the training and test sets. Section 4 contains details on the approach and results.

We then evaluate the effectiveness of all ML algorithms with different metrics on the dataset Big. We use the hyperparameters and dimensionality reduction obtained at the beginning and the same dataset distribution (i.e., the same 70% of data for training and 30% for testing). The significance of those results is assessed thanks to Wilcoxon tests. Then we investigate false positives and false negatives to better understand the models' efficiency as well as accuracy on binaries that we pack ourselves. Next, we evaluate the robustness of the nine ML algorithms against adversarial examples. Section 5 contains our work's details and results.

Lastly, we train the tuned ML algorithms on the Big dataset and assess their economical and chronological performances. We test the classifiers on the Chrono dataset that we earlier divided into small chronological periods (four periods in total, each comprising two weeks). This approach evaluates the effectiveness of the ML algorithms over time, relating this information to training costs. The experiments we conduct use datasets from various periods, so we employ the entirety of the early dataset for training. We apply no cross-validation. Section 6 details the approach and results.

3.3. Hyper-parameters Tuning

Each of the nine ML algorithms has unique hyper-parameters that can be tuned to improve performance on different datasets and classification problems. We test different values to obtain the best combination of hyper-parameters. We define the best combination using accuracy, with equal accuracy outcomes determined by training time. We test each combination using 70% of the Big dataset for training and applying 10-fold cross-validation with this training set.

The tested hyper-parameters for each ML algorithm are listed below with their final values. Details about tuning and the results for the different values are in Appendix B.

- KNN: Number of neighbors $[1; 30]$. Chosen value : 16.
- DT: Criterion to measure quality of split (entropy or Gini impurity), minimal number of samples required for a leaf $[2; 12]$, and the maximal depth of the tree $[3; 12]$. Chosen values : gini as criterion to measure quality of split, ten as the minimal number of samples required for a leaf and eight as the maximal depth of the tree.
- RF and GBDT: Number of estimators $[2; 50]$ (RF) and (GBDT), we use the best parameters of DT for individual trees. Chosen value: 20 trees as estimators for both algorithms.
- MLP: Architecture of the network (i.e. $[1; 3]$ hidden layers and $25, 50, 100$ neurons), activation functions (identity, hyperbolic tangent, logistic function, rectified) and solver (adam, sgd or lbfgs). Chosen values: 50 neurons in the first layer and 100 in the second layer as architecture; stochastic descent and logistic function as solver and activation function.

- KSVM: Kernel used (linear, polynomial, rbf, or sigmoid). Chosen kernel: radial basis functions (rbf).

Note that GNBC, LR, and LSVM do not have interesting hyper-parameters for our problem.

4. Features Analysis

This section studies the relevance and significance of the 119 different features gathered in [13]. Initially, Biondi et al. grouped these features into six categories (Metadata, Section, Entropy, Entry bytes, Import functions and Resource features) and studied the impact of each group. In this section, we propose how to analyze features individually to better understand their influence on the classifiers as proposed in [46]. Appendix C contains a full list of these features and their categories.

We consider relevance and significance of features using different approaches: variance analysis, mutual information analysis, and SHAP values. We then apply feature selection and dimensionality reduction methods to reduce the set of features.

4.1. First insights

Variance analysis. We first apply a simple variance analysis on the 119 features to explore the relevance of each feature. This analysis aims to find features with low variance (i.e., majority features with the same value). We observe nine features with very low variance (less than 0.1). These include **DLL2**, **DLL4**, **DLL5**, **DLL6**, **DLL7**, **CODE_NOT_X**, **EP_NOT_X**, **ADDR_NOT_ALIGN** and **DBG_DIR**. We could expect these features to show low impact on the model's decision. Feature selection method and dimensionality reduction will most likely discard these features.

Mutual information analysis. To continue the exploration of the feature space, we calculate mutual information between each feature and the target label. We observe ten features with larger mutual information including **ENT_HEADER**, **ENT_EP_SEC**, **RATIO_SEC_EP**, and **RATIO_MAL_API** as well as features related to specific **BYTE**. The ten worst features are the same identified via the variance analysis as well as **STACKC_SIZE**, **ALIGN** and **X_NOT_CODE**.

4.2. SHAP values

As discussed in Section 2.3, SHAP values can help better interpret model decisions after the training phase. To examine the different classifiers, we randomly select 1000 samples from each class and plot their SHAP values in Figure 4-12 to present the 20 most impactful features. Each point corresponds to a sample, and color indicates if the feature has a high value (red) or a low value (blue). Points to the right indicate that the feature value tends to classify the sample as packed while points to the left indicate the opposite.

From these figures, we discern the more influential features:

- **STAND_SEC** feature corresponds to *the number of standard sections in the binary.* As explained in Section 2.2, some packers rename existing or create new sections. In all classifiers, a low number of standard sections pushes the model to classify the binary as packed.
- **EP_NOT_CODE** corresponds to the *entry point not in the code section* while **EP_NOT_STD** corresponds to the *entry point not in a standard section.* This could happen when the entry point which corresponds with the unpacking stub is in a dedicated section.

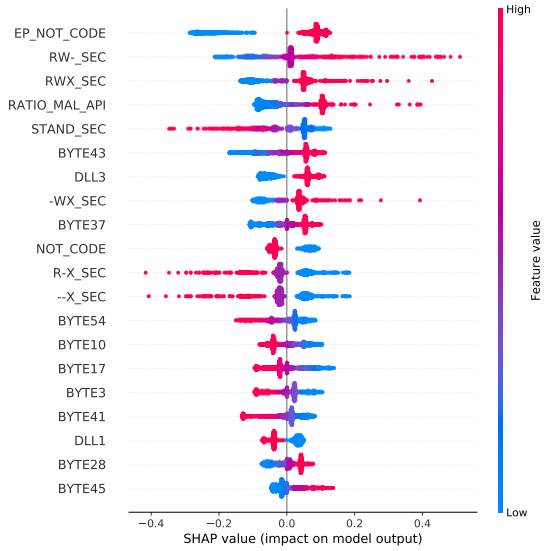


Figure 4: SHAP values for the LR classifier. Only the 20 most important features are represented.

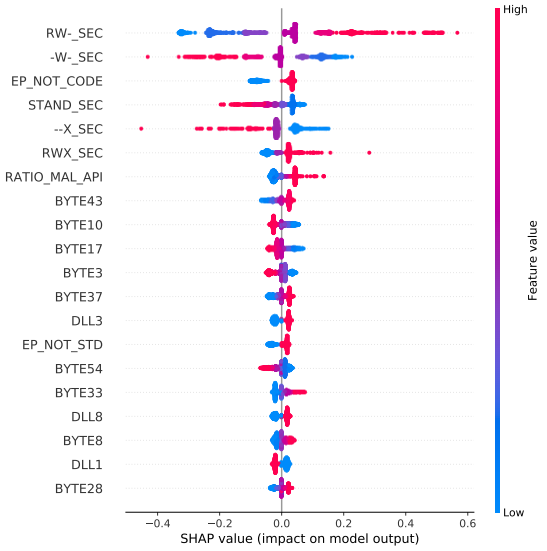


Figure 5: SHAP values for the LSVM classifier. Only the 20 most important features are represented.

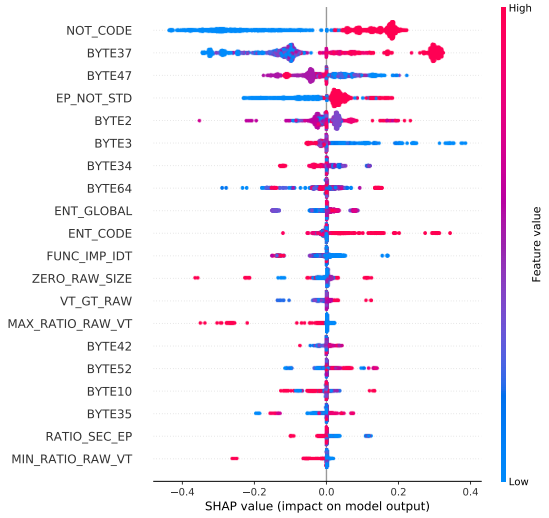


Figure 6: SHAP values for the DT classifier. Only the 20 most important features are represented.

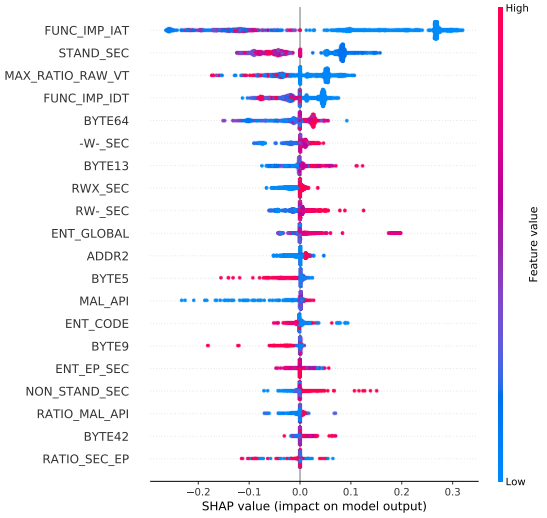


Figure 7: SHAP values for the GBDT classifier. Only the 20 most important features are represented.

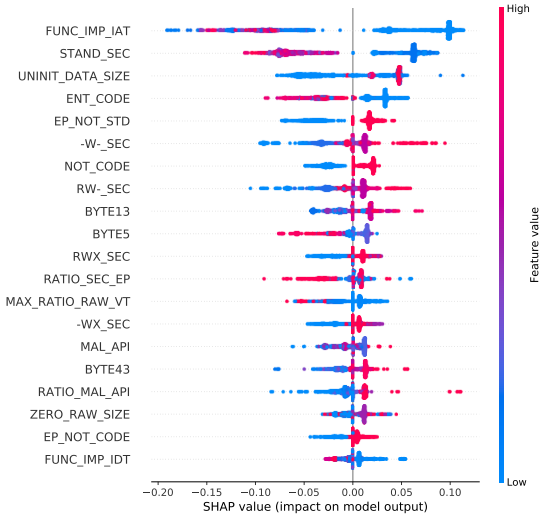


Figure 8: SHAP values for the RF classifier. Only the 20 most important features are represented.

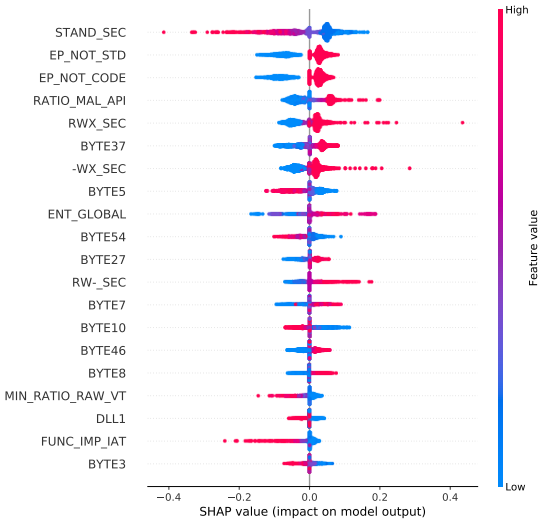


Figure 9: SHAP values for the MLP classifier. Only the 20 most important features are represented.

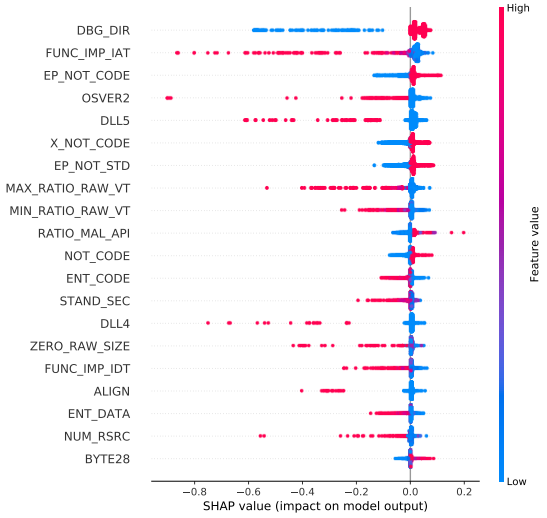


Figure 10: SHAP values for the GNBC classifier. Only the 20 most important features are represented.

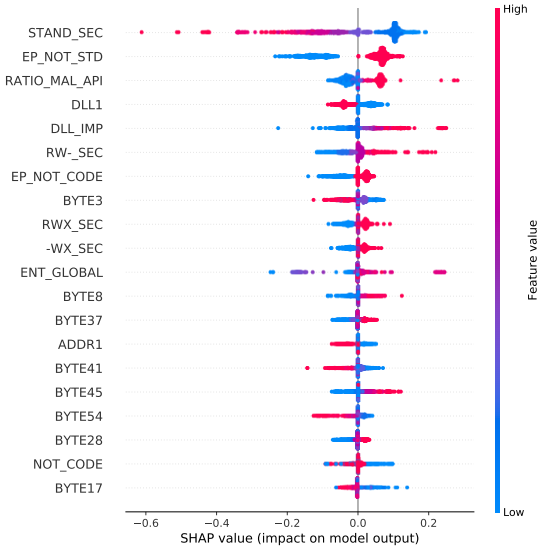


Figure 11: SHAP values for the KSVM classifier. Only the 20 most important features are represented.

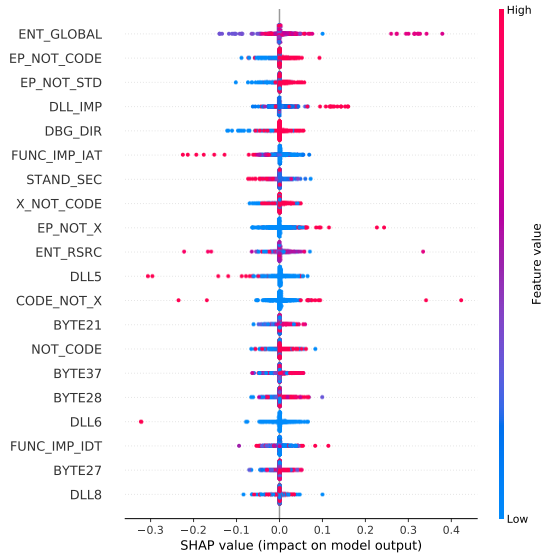


Figure 12: SHAP values for the KNN classifier. Only the 20 most important features are represented.

- **RATIO_MAL_API** is related to *the number of API calls imported* by the binary and *often used in malicious software* according to [56]. More precisely, it represents the ratio between the number of malicious API calls imported and the total number of API calls imported. These API calls are not intrinsically malicious but allow easy alteration or obfuscation of the control flow of a running program. This is achieved by allocating memory to store unpacked files, getting the address of specific API function calls after unpacking, etc. As expected, a bigger ratio means a higher chance of a packed binary.

- **NOT_CODE** corresponds to the *absence of code sections in the PE file*. This links strongly to the non-standard sections. Some packers modify the structure of the PE file by creating new sections that replace/embed classical code sections.

- **RWX_SEC** and **-WX_SEC** features correspond respectively to the *number of Readable, Writable, and Executable sections* and the *number of Writable and Executable sections*. Usually, a section that is both writable and executable denotes a self-modifying code (e.g., a code that will decrypt its content at runtime.) Although some packers could try to modify section properties at runtime, we observe a significant amount in our dataset that directly creates them with required properties.

- **FUNC_IMP_IAT** and **FUNC_IMP_IDT** are related to *the number of functions imported and found respectively in IAT or in IDT*. This observation links directly to the importance of **RATIO_MAL_API**. Packed files generally hide their behavior by loading via dynamic functions after unpacking. Thus, a limited number of functions are frequently imported directly, which explains the low value for these features in packed samples (see Figures 7 8 10 12).

- **ENT_GLOBAL** corresponds to the *byte entropy of the entire PE file* and **ENT_CODE** corresponds to the *byte entropy of the code section*. Since packing implies compression and/or

Classifier	Old Accuracy	Method	New # features/components	Accuracy
KNN	0.972	PCA	13	0.971
GNBC	0.745	SHAP	19	0.811
LR	0.936	RFE	38	0.931
LSVM	0.935	RFE	55	0.935
DT	0.944	SHAP	42	0.936
RF	0.952	RFE	37	0.953
GBDT	0.973	RFE	54	0.973
MLP	0.973	PCA	32	0.968
KSVM	0.976	SHAP	26	0.972

Table 4: Top results from applying RFE/SHAP selection/PCA.

encryption, entropies of sections are often high. This explains why this feature is used in literature [32, 18, 27, 42, 51].

- **BYTEx** features represent raw values of the xth byte after the entry point. These values seem to impact many models. This is likely due to the dominance of some packers in the dataset. As a traditional packing detector, ML models mimic a partially signature-based approach.

We can summarize our findings by recalling the process of packing and unpacking. A packed file will typically begin by executing the stub responsible for unpacking the file. This explains the role of **STAN_SEC**, **EP_NOT_CODE**, **EP_NOT_STD** and **NOT_CODE** which likely reflect the layout of the binary. A limited number of powerful imported functions hide the binary functionalities. These limited number of functions (e.g.,*GetProcAddress* which dynamically loads other functions) are labeled "malicious" by [56], explaining the features **FUNC_IMP_IAT**, **FUNC_IMP_IDT** and **RATIO_MAL_API**. Finally, packers are commonly compressors and/or encryptors. The code needs to modify itself to access the original binary. This explains importance of the features **ENT_GLOBAL**, **ENT_CODE** and **ENT_GLOBAL**.

4.3. Feature selection and PCA

Some features prove more relevant than others. Thus we can safely improve training time, memory space usage, and feature extraction time by reducing the number of features.
In this section, we apply two different wrapper methods: **Recursive Feature Elimination (RFE)** and SHAP selection [34]. We also compare these methods with the application of **Principal Component Analysis (PCA)** to reduce data dimensionality. We ultimately select the method that results in the best reduction of features while leading only to a limited loss of recall (i.e., 0.5%).

We apply RFE on the models: DT,RF,GBDT,LR, and LSVM (RFE is not applied on the other models since it is not directly applicable). We apply SHAP selection method to all the classifiers by considering features from one to 119. Finally, we apply PCA to all the classifiers. We follow a similar methodology to RFE: trying a different number of components and keeping the smaller number that leads to a limited loss of recall. This allows us to ignore multiple features with low variance. We keep the best of the three different methods. Table 4 shows our results.

We think many features have limited impact given the minimal effects of dimensionality reduction on the ML algorithm performance (see Section 4.1.) Our results confirm that we can discard low variance features (identified previously) without reducing the high performance of principal

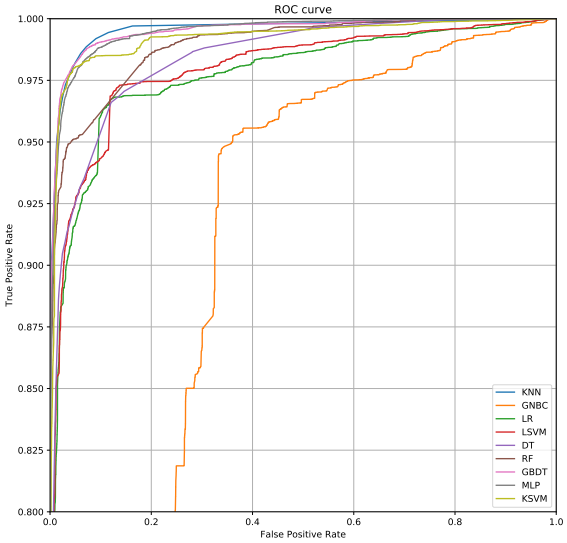


Figure 13: ROC curves for all 9 ML using Big data set.

Classifier	Precision	Recall	F-score
KNN	0.9762	**0.9733**	0.9748
GNBC	0.7675	0.9531	0.8503
LR	0.9554	0.9208	0.9378
LSVM	0.9521	0.9313	0.9416
DT	0.9647	0.9206	0.9421
RF	0.9812	0.9358	0.958
GBDT	0.9822	**0.9707**	0.9764
MLP	0.9777	**0.9668**	0.9722
KSVM	0.9804	**0.9694**	0.9749

Table 5: Detailed metrics for all 9 ML trained classifiers on Big data set.

components analysis (PCA). PCA aims to preserve as much variance as possible by creating new features via the linear combination of others. Limiting the number of principal components (i.e., the amount of variance conserved) shows features with lower variance will have negligible impact after dimensionality reduction.

We observe in this section that some features discussed broadly in the literature have a lot of impact on the decision processes of Machine Learning models, while other features are less influential. Features like the **number of standard sections**, properties of **the entry point section**, number and type of **imported API functions**, **entropy** and specific **permission of sections** seem to guide aspects of the ML decision processes. The importance of **BYTE** features is likely due to the dominance of specific packers like *UPX* (which reproduce a partially signature-based approach). We find identifying these key features to detect packing according to ML models could reduce feature extraction cost, memory consumption and training costs, and provide a better understanding of the packing detection problem and the limitations of static approaches.

5. Classifier Effectiveness

This section focuses on the effectiveness of the classifiers by looking at different classification metrics observed on a large dataset. We examine false negatives to identify the packers that evade our classifier and investigate the respective packer characteristics. Finally, we assess the robustness of our classifiers by determining how easy it is to create adversarial examples that evade detection.

5.1. Evaluation of Classification Metrics

We first examine the ROC curves related to the different classifiers as shown in Figure 13. From these curves, we can observe four classifiers that perform well: KNN, GBDT, MLP and KSVM. GNBC performs poorly.

	KNN	GNBC	LR	LSVM	DT	RF	GBDT	MLP	KSVM
KNN	1.0	0.0625	0.0625	0.0625	0.0625	0.125	0.4375	0.125	0.3125
GNBC	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
LR	1.0	0.625	1.0	0.9375	0.9375	1.0	1.0	1.0	1.0
LSVM	1.0	0.0625	0.125	1.0	0.9375	1.0	1.0	1.0	1.0
DT	1.0	0.0625	0.125	0.125	1.0	0.8125	1.0	1.0	1.0
RF	0.9375	0.0625	0.0625	0.0625	0.3125	1.0	0.9456	0.9375	0.9375
GBDT	0.6875	0.0625	0.0625	0.0625	0.0625	0.0544	1.0	0.4375	0.4375
MLP	0.9375	0.0625	0.0625	0.0625	0.0625	0.125	0.6875	1.0	1.0
KSVM	0.8125	0.0625	0.0625	0.0625	0.0625	0.125	0.6875	0.0625	1.0

Table 6: p-values of the one-sided Wilcoxon rank sum hypothesis test for the comparison of the results in Table 5. Each cell in the table indicates whether the outcome median of the algorithm in line i is greater than the outcome median of the algorithm in column j.

Table 5 presents more detailed explorations of the metrics to evaluate the different classifiers. The related Wilcoxon test in Table 6, evaluated using 10-fold , reinforces our results.

Statistically, GNBC offers the worst F-score but is the faster model to train. Linear models LR and LSVM offer average performances, providing good baselines from which to perceive the advantages of other models. Looking at the overhead required to train LSVM, LR serves as a better baseline. Tree-based models DT, RF, and GBDT perform well with less training time, suggesting why the literature may have extensively explored these models. RF and GBDT both show a high precision in their decisions; GBDT has the best recall compared to the other classifiers. Finally, MLP and KSVM perform well, similar to the top classifiers.

According to these metrics and the Wilcoxon test, GBDT and KNN offer statistically the best performance. However, KNN requires more time to classify a sample as it must compute the distance between sample and training data. GBDT appears a better solution. We see that a majority of classifiers have a higher precision (lower false positives) than their recall (linked to number of false negatives).

To improve our understanding of false negatives, we decide to extract which samples are badly classified and the characteristics of the related packer. Figure 14 presents the top classes of poorly classified packed files and where this happens in the training set for GBDT. The same packers appear in the results of the other classifiers (with the addition of *nullsoft* and *yoda's protector*).

If we check which features lead those classifiers to a poor decision, we observe the misclassified packed binaries do not exhibit the same characteristics as the other packed binaries. Most misclassified packed binaries have a **FUNC_IMP_IAT** and **FUNC_IMP_IDT** similar to non-packed binaries. Most misclassified packed binaries also exhibit a high number of **STAND_SEC**, or standard sections, and a high **MAX_RATIO_RAW_VT**, or the maximum ratio of raw data to virtual size among all sections. As seen in Section 4, these features all highly impact the decision processes of most classifiers. (The **MAX_RATIO_RAW_VT** feature is used mainly in tree-based models.) Our classifier seems to exhibit a bias towards these packers because of their small occurrence in the data set when compred with other well-known packers like `UPX` or `ASPack`. This calls to mind previous research [5, 33] concerning the limitations of static approaches and paves the way for future research around new features or hybrid approaches.

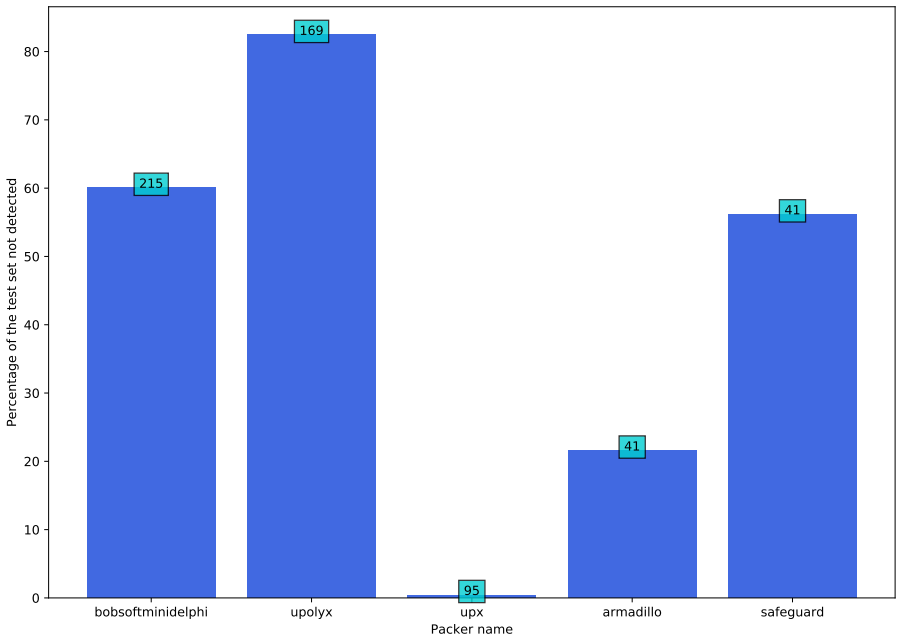


Figure 14: Top packers evading GBDT classifier. For each packer, the Y-axis represents the percentages of evading samples found in the test set while the number indicates how many binaries it represents.

5.2. Robustness

To evaluate the robustness of our classifiers, we create adversarial examples by applying simple modifications to packed samples. To do this, we randomly extract 3000 correctly classified packed binaries from the Big dataset. Then we run MAB-malware [45], a framework to generate adversarial examples, on the selected binaries. MAB-malware modifies various elements of a binary by, for instance, renaming, adding, or appending sections. MAB-malware may also add an overlay, remove or debug information or a certificate, or break the checksum, etc. Figure 15 shows how some classifiers that are very effective in detecting packing can be easily bypassed by the MAB-malware-modified binaries. When we apply the MAB-malware, for example, DT and KSVM are so easily bypassed that the results do not appear in Figure 15; the escape rate for these classifiers is just too high after a single modification.

After sufficient mutations with MAB-malware, LSVM, LR, and RF are also evaded. MLP, GNBC, KNN, and GBDT seem to be more resilient and able to cope with these mutations. GBDT in particular remains largely unimpacted.

The MAB-malware mutation with the most impact is the **addition of a section** (SA: Section Add) or **the change of a section name** (SR: Section Rename). Figure 16 illustrates the sequence of modifications used to escape LSVM and GBDT. The classifiers prove over-reliant on section-related features. They mimic some simple heuristics implemented in detectors on section names (e.g., Manalyze [31]) and fail to generalize packing detection properly.

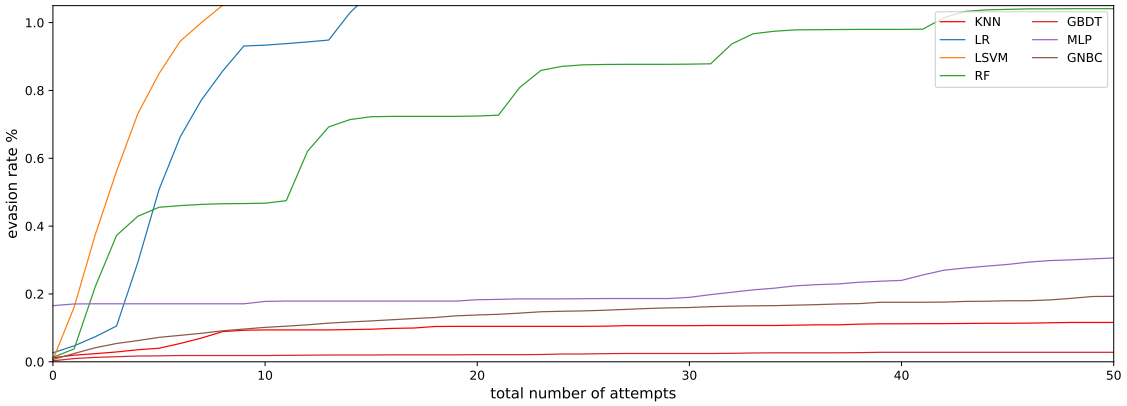
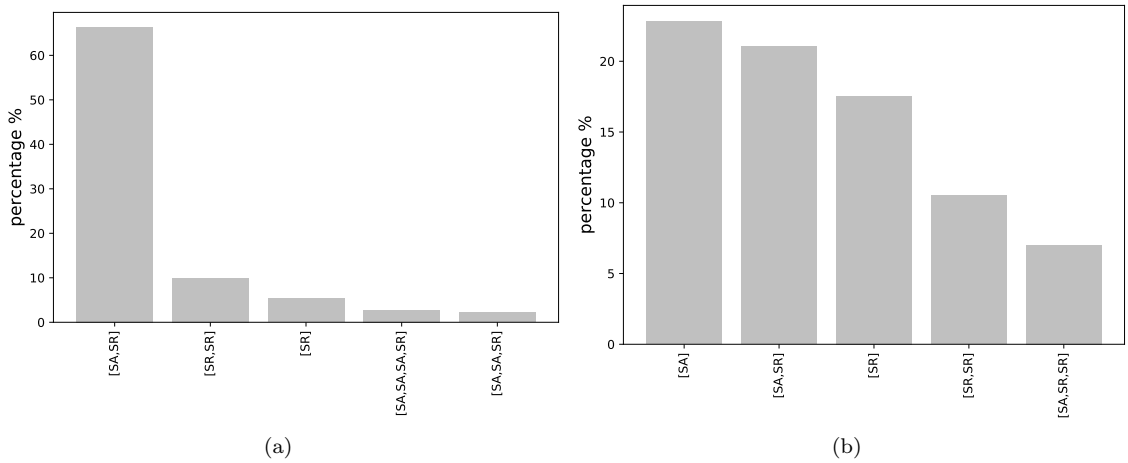


Figure 15: Evasion rate observed by MAB-malware on our classifiers. DT and KSVM are not shown since they are totally evaded after the first mutation.



(a) (b)

Figure 16: Modification(s) applied for samples which successfully evade (a) LSVM classifier (b) GBDT classifier

5.3. Self-packed binaries

Finally, we assess our different classifiers on self-packed binaries with the Packing-Box [22]. This framework allows users to create and manipulate datasets of binaries packed with selected **known** packers. This experiment assesses the classifiers' performance against a known ground truth (compared with our previous ground truth that might contain errors due to the labeling process). Table 7 shows that all classifiers except for KNN and KSVM detect `UPX` packer. With MPRESS, KSVM performs poorly; KNN works better. The other packers are not well represented in our dataset with regard to the packer's distribution of Section 3. This might explain why most of our classifiers do not easily detect packers like Amber. Overall, RF, GBDT and MLP exhibit the most interesting results. Out of the six packers, RF and GBDT detect five correctly while MLP detects four correctly. However, MLP is the only classifier to detect Amber. With speed and

simplicity, GNBC detects four packers.

Classifier	UPX	Amber	Eronana	mew	MPRESS	rlpack
KNN	0.69	0.0	0.85	0.95	0.99	0.99
GNBC	1.0	0.0	0.01	0.99	1.0	0.99
LR	1.0	0.0	1.0	0.99	1.0	0.0
LSVM	1.0	0.0	0.99	0.99	1.0	0.0
DT	1.0	0.0	0.01	0.99	1.0	0.99
RF	1.0	0.0	0.99	0.99	1.0	0.99
GBDT	1.0	0.0	1.0	0.98	0.81	0.99
MLP	1.0	1.0	0.64	0.22	1.0	0.99
KSVM	0.55	0.0	0.99	0.99	0.34	0.99

Table 7: Accuracy of the different classifiers on binaries packed with the packing box.

6. Packing detection economics

This section focuses on deployment costs of classifiers by proposing an economical analysis looking at long-term accuracy, memory, and training/inference time. To this purpose, we explore the effectiveness of all nine ML algorithms over chronological data. Then, we briefly look at the economics behind the different models (i.e., the required resources and resulting performance) as already proposed by [49] in the context of malware detection.

Our analysis first uses the Big dataset to train each classifier, training on samples collected from 2019-09-17 to 2020-04-30. Then we test the results using the Chrono dataset, comprised of malware samples obtained between 2020-05-01 and 2020-07-29. As described in 3.1, the Chrono dataset includes four subsets, each corresponding to a separate time period, labeled T1 to T4. **Our malware are harvested from a regular malware feed from Cisco that consists of malware observed[2] 'in the wild' on the given date. As a result, we observe changes in malware distribution during the different time periods as exposed in Section 3.1**. Separating datasets per time period allows us to represent evolving trends in packers and the impact of this distribution shift.

The accuracy of ML classifiers over time is in Table 8. As expected, the effectiveness of all ML-trained classifiers decreases with time. This corresponds with a distribution shift suggested in [13] that can be explained by different trends in the packers used depending on the period of interest. According to the table, KNN, RF, GBDT, MLP, and KSVM are less impacted by the distribution shift. To make an informed decision about which model to select, we can put our observations into perspective by considering the training time, inference time (average time for 10,000 samples), and memory requirements of the different models listed in Table 8. This allows us to choose the model that best fits our available resources and specific requirements. Such considerations are particularly important in real-world applications where these classifiers will be integrated into larger toolchains for malware analysis. For instance, KNN makes use of the entire dataset, resulting in

[2]It is worth observing that observation time differs from creation time, i.e. there are situation where it takes a considerable amount of time for a specialist to observe an active malware. As it is impossible to know the creation time, we base our analysis on observation times given in data sets acknowledged by the community and specialists.

higher inference time and memory usage. MLP and KSVM have longer training times but smaller inference time and require less memory space. In contrast, GBDT and RF offer a good trade-off between efficiency, training/inference time, and memory usage while using a small number of features. Table 8 also illustrates how reducing the number of features improves training/inference time and reduces memory for most of the models while preserving their accuracies.

Classifier	Baseline	T1	T2	T3	T4	Training time[s]	Inference* time [s]	Memory [kb]	# features
KNN	0.9697	0.9709	0.9707	0.9354	0.9378	0.5	50	113 200	119
	0.9722	0.9709	0.9672	0.9245	0.9297	0.5	1	15 100	13 (PCA)
GNBC	0.7457	0.6221	0.5955	0.7714	0.8549	0.2	0.01	4	119
	0.8114	0.7849	0.7386	0.8406	0.8921	0.1	0.01	1	19
LR	0.9364	0.924	0.9343	0.9227	0.9264	22	0.01	2	119
	0.9314	0.8979	0.8954	0.9103	0.926	9.5	0.0005	1	38
LSVM	0.9355	0.9164	0.9297	0.9131	0.9258	2082	0.01	7	119
	0.9351	0.9166	0.9333	0.9114	0.9239	1281	0.007	4	55
DT	0.9448	0.9081	0.9096	0.901	0.9223	1	0.002	9	119
	0.9365	0.8731	0.8901	0.9091	0.9116	0.25	0.001	9	42
RF	0.9521	0.937	0.9424	0.949	0.9479	2	0.02	154	119
	0.9539	0.942	0.9516	0.9472	0.9476	2	0.02	159	37
GBDT	0.9734	0.9721	0.9737	0.9383	0.9371	22	0.01	164	119
	0.9736	0.9733	0.9741	0.9368	0.9358	11	0.01	164	54
MLP	0.9736	0.9741	0.9735	0.9381	0.9369	476	0.045	190	119
	0.9665	0.9612	0.9614	0.93	0.9374	466	0.045	121	32 (PCA)
KSVM	0.9766	0.9792	0.9749	0.9306	0.9355	2781	21.3	8900	119
	0.972	0.9736	0.9721	0.937	0.9388	1212	5.9	3100	26

Table 8: Evolution of the accuracy of nine ML-trained classifiers trained on Big data set and tested on Chrono data set split by chronological period. Metrics of each classifier are presented with and without features reduction methods. *The inference time is the average time to evaluate 10,000 samples.

7. Future works and conclusion

This section briefly discusses future research directions and concludes.

Ground truth. One challenge to the validity of much work in the area of machine learning applied to malware is the selection of an accurate **ground truth** [8]. Building our ground truth relies on different approaches (i.e., PeID, Manalyze, PEframe, and DIE) each of which are subject to limitations. Our use of the consensus approach should mitigate some limitations (see also [13]). However, the approach is vulnerable to rare or unknown packers. The robustness exploration (see Section 5) highlights this problem by showing packers that are badly classified, reasons behind misclassification, and how to create adversarial examples. We show how, like in tools used for ground truth generation, section-related features are important in some classifiers. This parallel could explain some possible limitations of our ground truth.

Dynamic features. The problem of packing detection as part of a larger malware detection and defense system motivates our choice to focus on static features. However, as illustrated in literature [33] and in our experiments with adversarial examples, some static features can be bypassed. Different related work has thus considered dynamic features [11, 17, 29]. Dynamic features could likely represent the behavior of the analyzed binaries and better generalize packing detection. Yet extracting these features can be costly, and trade-offs must be examined.

Explainable AI. Investigating correlated features and their impact on the SHAP values presents another challenge. While SHAP values supply good indicators concerning the decision processes of classifiers, SHAP values do not account for possible dependencies between features. Regarding the SHAP calculation [35], replacing feature values with random instance values is easier than sampling from the marginal distribution. However, this ignores dependencies between features and puts significant weight on unlikely data points.

Packing classification. The problem of packing detection has also been extended to packing classification [47, 19]. While those works generally focus on already known packers, they do not investigate which features influence the decision process nor which approach is more resilient in an adversarial context. Moreover, those classifiers could also be generalized to detect properties of the packers using the terminology proposed in [21] (i.e., compressor, cryptors, virtualizer, etc.)

Adversarial examples. We used MAB-malware [45] to generate adversarial examples. To our knowledge, no dedicated tool exists that targets packing detectors. Another research direction could design specific attacks on packing detectors by targeting their more influential features. The potential benefits of dynamic features to counteract these attacks are worthy of investigation.

Uncertainty quantification. Finally, real-world applications would benefit from estimating the uncertainty of our classifiers' decisions. Knowing our classifiers' confidence level could be exploited to design a modular approach in a malware analysis toolchain. A high confidence would mean sending the binary to an unpacker, and a moderate confidence could initiate an evaluation of more (dynamic) features to improve the confidence level. Recent research [10] looks at estimating uncertainty (e.g., using conformal prediction) and shown benefits against adversarial examples or concept drifts in the malware ecosystem.

Detecting packed binaries is a critical aspect of modern cybersecurity defense. To this end, researchers continue to develop numerous approaches. A wide range of algorithms are proposed and applied with various features. Our study aims to offer contributions related to packed binaries detection: which features are most significant, which machine learning algorithms are most effective, and which methods offer the best trade-offs in terms of resources.

The feature significance analysis shows that across nine different ML algorithms, there are a small number of static features that highly influence the decision process. These are related to: the number of standard sections, the entry point not in the code/standard section, malicious API calls, the absence of a code section, the number of readable-writable-executable and writable-executable sections, the number of functions imported and found in IAT/IDT, the global/code entropy, and specific byte values.

We compare the effectiveness of nine different ML algorithms with classical metrics, incorporating adversarial examples and economics. Overall, KNN, GBDT, and MLP prove the most effective and robust classifiers. These three algorithms score well in accuracy, precision, recall, and F-score while remaining resilient to adversarial examples. KNN and GBDT perform particularly well. The other classifiers are more vulnerable to simple modifications of section names and numbers. Observing which known packers are classified poorly allows us to discover the limitations of our static features and highlight needed areas related to the ground truth building.

Finally, we use chronological samples to evaluate the long-term effectiveness of nine ML algorithms. The results show that, as expected, all classifiers have reduced accuracy over time. However,

GBDT combines a low training cost with a modest memory requirement, making GBDT the most effective algorithm.

Acknowledgments. Charles-Henry Bertrand Van Ouytsel is FRIA grantee of the Belgian Fund for Scientific Research (FNRS-F.R.S.). We would like to thank Cisco for their malware feed, VirusTotal for giving us access to their academic API, and the CyberExcellence project funded by the Walloon Region under convention 2110186. Khanh Huu The Dam also would like to thank the Service Public de Wallonie Recherche under Grant No. 2110102 - PrEDICT by WIN2WAL.

References

[1] ClamAV. `https://www.clamav.net`. November 2019.

[2] Detect-it-easy version 2.06. `https://github.com/horsicq/Detect-It-Easy`. November 2019.

[3] PEiD, version 0.95. `https://appnee.com/peid/`. November 2019.

[4] Virustotal report. `https://www.virustotal.com/gui/stats`.

[5] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel. When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In Network and Distributed Systems Security (NDSS) Symposium 2020, 2020.

[6] G. Amato. peframe version 6.0.3. `https://github.com/guelfoweb/peframe`. November 2019.

[7] R. Arora, A. Singh, H. Pareek, and U. R. Edara. A heuristics-based static analysis approach for detecting packed pe binaries. International Journal of Security and Its Applications, 7(5):257–268, 2013.

[8] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck. Dos and don'ts of machine learning in computer security. In Proc. of the USENIX Security Symposium, 2022.

[9] U. Baldangombo, N. Jambaljav, and S.-J. Horng. A static malware detection system using data mining methods. arXiv preprint arXiv:1308.2831, 2013.

[10] F. Barbero, F. Pendlebury, F. Pierazzi, and L. Cavallaro. Transcending transcend: Revisiting malware classification with conformal evaluation. arXiv preprint arXiv:2010.03856, 2020.

[11] M. Bat-Erdene, T. Kim, H. Li, and H. Lee. Dynamic classification of packing algorithms for inspecting executables using entropy analysis. In 2013 8th International Conference on Malicious and Unwanted Software:" The Americas"(MALWARE), pages 19–26. IEEE, 2013.

[12] E. Bergenholtz, E. Casalicchio, D. Ilie, and A. Moss. Detection of metamorphic malware packers using multilayered lstm networks. In International Conference on Information and Communications Security, pages 36–53. Springer, 2020.

[13] F. Biondi, M. A. Enescu, T. Given-Wilson, A. Legay, L. Noureddine, and V. Verma. Effective, efficient, and robust packing detection and classification. Computers & Security, 85:436–451, 2019.

[14] F. Biondi, T. Given-Wilson, A. Legay, C. Puodzius, and J. Quilbeuf. Tutorial: An overview of malware detection and evasion techniques. In Leveraging Applications of Formal Methods, Verification and Validation. Modeling: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I 8, pages 565–586. Springer, 2018.

[15] C. M. Bishop and N. M. Nasrabadi. Pattern recognition and machine learning, volume 4. Springer, 2006.

[16] A. Bommert, X. Sun, B. Bischl, J. Rahnenführer, and M. Lang. Benchmark for filter methods for feature selection in high-dimensional classification data. Computational Statistics & Data Analysis, 143:106839, 2020.

[17] B. Cheng, J. Ming, J. Fu, G. Peng, T. Chen, X. Zhang, and J. Marion. Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018, pages 395–411. ACM, 2018.

[18] Y.-s. Choi, I.-k. Kim, J.-t. Oh, and J.-c. Ryou. Pe file header analysis-based packed pe file detection technique (phad). In International Symposium on Computer Science and its Applications, pages 28–31. IEEE, 2008.

[19] K. H. T. Dam, T. Given-Wilson, A. Legay, and R. Veroneze. Packer classification based on association rule mining. Applied Soft Computing, 127:109373, 2022.

[20] D. Devi and S. Nandi. Pe file features in detection of packed executables. International Journal of Computer Theory and Engineering, 4(3):476, 2012.

[21] A. D'Hondt, C.-H. B. Van Ouytsel, and A. Legay. Experimental toolkit for manipulating executable packing. arXiv preprint arXiv:2302.09286, 2023.

[22] A. D'Hont, C.-H. Bertrand Van Ouytsel, and A. Legay. Packing Box : Study executable packing easy with this dedicated platform. `https://github.com/packing-box/docker-packing-box`, 2022.

[23] P. Ferrie. Anti-unpacker tricks–part one. Virus Bulletin, 4, 2008.

[24] M. N. Gagnon, S. Taylor, and A. K. Ghosh. Software protection through anti-debugging. IEEE Security & Privacy, 5(3):82–84, 2007.

[25] S. Han, K. Lee, and S. Lee. Packed pe file detection for malware forensics. In 2009 2nd International Conference on Computer Science and Its Applications, CSA 2009, page 5404211, 2009.

[26] M. Hollander, D. A. Wolfe, and E. Chicken. Nonparametric statistical methods, volume 751. John Wiley & Sons, 2013.

[27] G. Jeong, E. Choo, J. Lee, M. Bat-Erdene, and H. Lee. Generic unpacking using entropy analysis. In 2010 5th International Conference on Malicious and Unwanted Software, pages 98–105. IEEE, 2010.

[28] M. Kalash, M. Rochan, N. Mohammed, N. D. Bruce, Y. Wang, and F. Iqbal. Malware classification with deep convolutional neural networks. In 2018 9th IFIP international conference on new technologies, mobility and security (NTMS), pages 1–5. IEEE, 2018.

[29] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In Proceedings of the 2007 ACM workshop on Recurring malcode, pages 46–53, 2007.

[30] J.-W. Kim, J. Namgung, Y.-S. Moon, and M.-J. Choi. Experimental comparison of machine learning models in malware packing detection. In 2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS), pages 377–380. IEEE, 2020.

[31] I. Kwiatkowski. Manalyze. `https://github.com/JusticeRage/Manalyze`. November 2019.

[32] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. IEEE Security & Privacy, 5(2):40–45, 2007.

[33] A. Mantovani, S. Aonzo, X. Ugarte-Pedrero, A. Merlo, and D. Balzarotti. Prevalence and impact of low-entropy packing schemes in the malware ecosystem. In NDSS, 2020.

[34] W. E. Marcílio and D. M. Eler. From explanations to feature selection: assessing shap values as feature selection mechanism. In 2020 33rd SIBGRAPI conference on Graphics, Patterns and Images (SIBGRAPI), pages 340–347. Ieee, 2020.

[35] C. Molnar. Interpretable machine learning. Lulu. com, 2020.

[36] T. Muralidharan, A. Cohen, N. Gerson, and N. Nissim. File packing from the malware perspective: Techniques, analysis approaches, and directions for enhancements. ACM Computing Surveys, 55(5):1–45, 2022.

[37] S. Naseer, Y. Saleem, S. Khalid, M. K. Bashir, J. Han, M. M. Iqbal, and K. Han. Enhanced network anomaly detection based on deep neural networks. IEEE access, 6:48231–48246, 2018.

[38] L. Noureddine, A. Heuser, C. Puodzius, and O. Zendra. Se-pac: A self-evolving packer classifier against rapid packers evolution. In CODASPY'21: Eleventh ACM Conference on Data and Application Security and Privacy, 2021.

[39] 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. Journal of Machine Learning Research, 12:2825–2830, 2011.

[40] R. Perdisci, A. Lanzi, and W. Lee. Classification of packed executables for accurate computer virus detection. Pattern recognition letters, 29(14):1941–1946, 2008.

[41] R. Perdisci, A. Lanzi, and W. Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In 2008 Annual Computer Security Applications Conference (ACSAC), pages 301–310. IEEE, 2008.

[42] J. Raphel and P. Vinod. Information theoretic method for classification of packed and encoded files. In Proceedings of the 8th International Conference on Security of Information and Networks, pages 296–303, 2015.

[43] I. Santos, X. Ugarte-Pedrero, B. Sanz, C. Laorden, and P. G. Bringas. Collective classification for packed executable identification. In Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference, pages 23–30, 2011.

[44] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. Avclass: A tool for massive malware labeling. In International symposium on research in attacks, intrusions, and defenses, pages 230–253. Springer, 2016.

[45] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin. Mab-malware: A reinforcement learning framework for blackbox generation of adversarial malware. In Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, pages 990–1003, 2022.

[46] A. Sotgiu, M. Pintor, and B. Biggio. Explainability-based debugging of machine learning for vulnerability discovery. In Proceedings of the 17th International Conference on Availability, Reliability and Security, ARES '22, New York, NY, USA, 2022. Association for Computing Machinery.

[47] L. Sun, S. Versteeg, S. Boztaş, and T. Yann. Pattern recognition techniques for the classification of malware packers. In Australasian Conference on Information Security and Privacy, pages 370–390. Springer, 2010.

[48] S. Treadwell and M. Zhou. A heuristic approach for detection of obfuscated malware. In IEEE International Conference on Intelligence and Security Informatics, ISI 2009, Dallas, Texas, USA, June 8-11, 2009, Proceedings, pages 291–299. IEEE, 2009.

[49] D. Ucci, L. Aniello, and R. Baldoni. Survey of machine learning techniques for malware analysis. Computers & Security, 81:123–147, 2019.

[50] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In 2015 IEEE Symposium on Security and Privacy, pages 659–673. IEEE, 2015.

[51] X. Ugarte-Pedrero, I. Santos, I. García-Ferreira, S. Huerta, B. Sanz, and P. G. Bringas. On the adoption of anomaly detection for packed executable filtering. Computers & Security, 43:126–144, 2014.

[52] X. Ugarte-Pedrero, I. Santos, B. Sanz, C. Laorden, and P. G. Bringas. Countering entropy measure attacks on packed software detection. In 2012 IEEE Consumer Communications and Networking Conference (CCNC), pages 164–168. IEEE, 2012.

[53] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, 17:261–272, 2020.

[54] VirusTotal. Virustotal: Yara in a nutshell (2019).

[55] F. Wilcoxon. Individual comparisons by ranking methods. Biometrics Bulletin, 1(6):80–83, 1945.

[56] M. Zakeri, F. Faraji Daneshgar, and M. Abbaspour. A static heuristic approach to detecting malware targets. Security and Communication Networks, 8(17):3015–3027, 2015.

Appendix A. Comparison of related works

As suggested by a reviewer, the Table A.9 presents a detailed comparison of existing approaches to packing detection.

Paper	Algorithm	Features	Comments
Lyda et al. [32]	Bintropy with two modes of operation : Entropy of the whole file or inspection of individual entropy section.	Global entropy	They used cleanware not packed and malware packed obtained from a honeypot. Their global entropy threshold of 6.67 allows a 3.8% false positive rate and 0.5% false negative rate. It was not assessed on other datasets. They study also section names with a high entropy on a larger dataset.
Choi et al. [18]	PHAD, an algorithm based on euclidian distance and custom threshold	eight features related to PE structure.	Their dataset of benign and malicious files is manually analyzed. They used a threshold, estimated on the dataset, to consider a file as packed.
Perdisci et al. [40] (1)	Naive Bayes Classifier, decision tree (J48 with and without bagging), KNN, Multilayer Perceptron and entropy threshold	nine features : number of standard and non-standard sections; number of executable sections; number of readable, writable, and executable sections; number of entries in IAT; PE header entropy; code sections entropy; data sections entropy and entire PE file entropy.	Their dataset is composed of benign non-packed executables, self-packed executables (on benign files), and packed malware from MALFEASE project.

Perdisci et al. [41] (2)	Bagged decision tree and multilayer perceptrons	Same as [40] and n-grams (with $n = 2$ or $n = 3$)	Their dataset is made of malware from MALFEASE project as well as benign files (self-packed). Their ground truth is established thanks to know detectors such as PEiD. Their objective is packing detection, extracting hidden code and detection of malicious code.
Han and Lee [25]	Check entrypoint section properties and entropy threshold.	Entrypoint section writable and entrypoint section entropy.	They used a balanced dataset of 400 executables (200 cleanware from windows installation and 200 malware from a honeypot) and an entropy threshold of 6.58.
Treadwell et al. [48]	Threshold related to a risk score (weighted sum of features)	8 Features : non-standard section names, flagged section names, entrypoint not in code section, TLS section, DLL with no export, low number of imports, abnormal ordinals in import table and bad PE format.	Their dataset is made of packed files (malware) and non-packed files (cleanware and malware). PEiD is used to decide packed status. Their initial goal is to detect obfuscated malware.
Santos et al. [43]	KNN, decision trees, random forest and variants of tree based models.	209 structural features of PE files (PE header, sections characteristics, EP section features and entropy)	They used benign and malicious executables from VxHeavens. Some samples are self-packed while others from Zeus malware family are known to be packed with custom packer (not detected with PEiD).
Devi et al. [20]	Feature representation after manual analysis	After manual analysis, focus on four features : global entropy, size of uninitialized data, header size, and raw data size	They focus on feature changes before and after self-packing with UPX on cleanware and malware.

Arora et al. [7]	Threshold based on score obtained thanks to several heuristics and power distance.	14 features, the five more important being : presence of specific section names, code section not executable, number of section writable and executable, data section executable and IAT in non-standard section.	They used benign non-packed and malicious packed/non-packed samples from windows, honeypot and websites. They observe some features need to be adapted for specific files (DLL, PE built with debug mode and PE built with CLR).
Ugarte-Pedrero et al. [51]	Bayesian networks, SVM, random forest, MLP and rules based classifiers	Same as [43] and n-grams.	Their approach focuses on anomaly detection to detect packed samples. The dataset is made of packed samples from Zeus family and manually packed samples (detected as non-packed by PEiD). For the non-packed samples, PEiD is used as well as custom heuristics to build the dataset.
Biondi et al. [13]	Naive Bayes Gaussian Classifier, Decision Tree, Random Forest and Extra Trees	The 119 features included also in this work	Dataset obtained from Cisco and labeled thanks to several tools : PEiD, YARA rules of VirusTotal and custom engine from Cisco. It highlights the necessity to favor datasets with more labels (even if there is non consensus over them) over reliable labels (with a high consensus). A simple economical analysis is performed, showing decision trees are a good trade-off to balance training cost and efficiency.

Kim et al. [30]	KNN, logistic regression, linear SVM, Random Forest, MLP, Nayve Bayes, SVM and decision tree.	13 features : matching a known packing signature, entrypoint section characteristics, file entropy and sections characteristics.	Their dataset is labeled thanks to author's previous work. A file is considered packed if : no entrypoint section, matching of known signature, entrypoint section is writeable, and entropy within a specific range.
Bergenholtz et al. [12]	Long short term memory (LSTM) neural network	The first 128 x86 instructions of the sample.	Samples are cleanware from a fresh windows installation packed with several metamorphic packers.
Mantovani et al. [33]	SVM, MLP and RF	Features extracted from several other papers related to PE structure, heuristics, N-grams, Opcodes, statistical features and entropy features.	Malicious binaries from VirusTotal for packed and non-packed samples. To build a ground truth, they develop a tool based on PANDA, inspect and compare their results with PEiD, Manalyze and DIE. The focus of the paper is to illustrate how models do not perform well when detecting low entropy packing schemes.
Aghakhani et al. [5]	SVM, MLP and RF	56,543 features : PE headers, PE sections, DLL & API imports, rich header, Byte & Opcode N-gram and strings.	They used a dataset from AV vendor and EMBER labeled thanks to vendors and VirusTotal. Authors illustrate the current overlap between packing and malware detection when based only on static features.

Table A.9: Summary of related works

Appendix B. Hyper-parameters choice

KNN. The accuracy of the classifier depending the number of neighbors is presented in Figure B.17.

DT. The accuracy for different parameters are evaluated. For the criterion to measure quality of a split, both gini and entropy give an accuracy of 0.963. That is why we keep the default gini

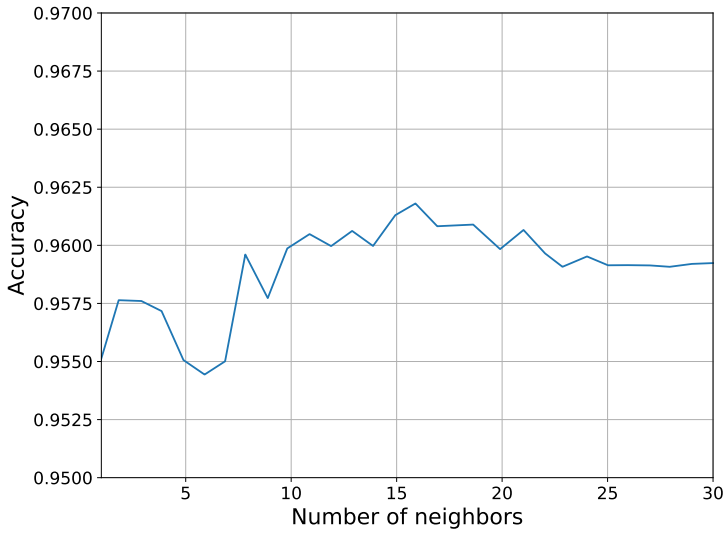


Figure B.17: Accuracy of the KNN classifier with different values of neighbor(s).

criterion. The accuracy for different depths and different minimum number of samples for a leaf are presented in Figure B.18 and Figure B.19.

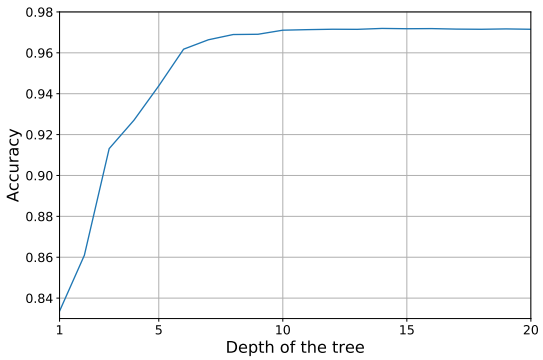


Figure B.18: Accuracy of the DT classifier with different depths.

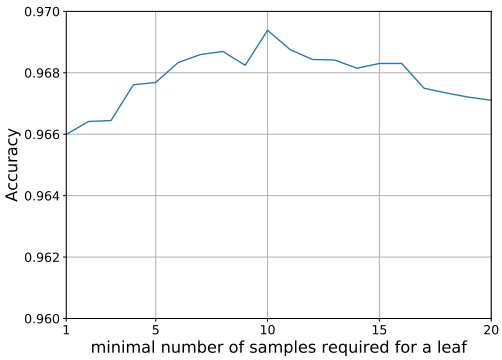


Figure B.19: Accuracy of the DT classifier with different minimum number of samples for a leaf.

RF *and* GBDT. The accuracy of those classifiers with respect to the number of estimators is presented in Figure B.20 and B.21.

MLP. Different architectures are tested. The results are presented in Table B.10. The different accuracy for the activation functions are 0.932 for identity, 0.972 for hyperbolic tangent, 0.973 for logistic function, and 0.972 for rectified linear unit. Regarding the choice of optimizer, the obtained accuracy is : 0.973 for adam, 0.974 for sgd, and 0.973 for lbfgs.

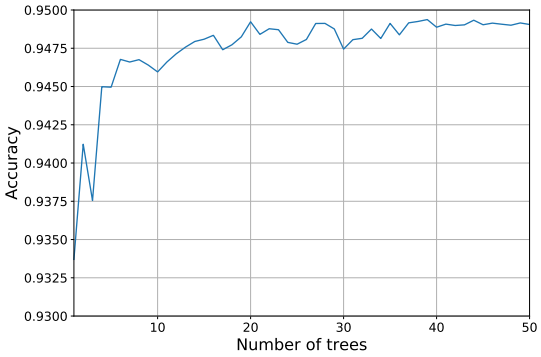


Figure B.20: Accuracy of the RF classifier with different number of trees.

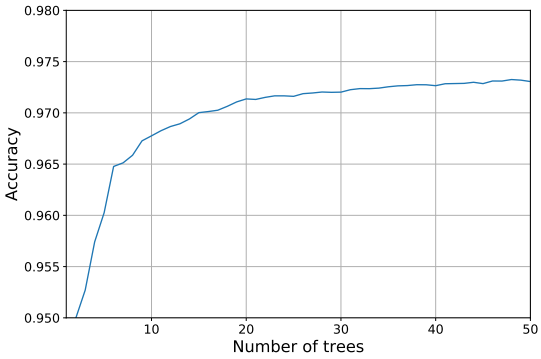


Figure B.21: Accuracy of the GBDT classifier with different number of trees.

# neurons 1st hidden layer	# neurons 2nd hidden layer	# neurons 3rd hidden layer	Accuracy
25	/	/	0.9702
50	/	/	0.9706
100	/	/	0.9720
25	50	/	0.9702
50	25	/	0.9714
25	100	/	0.9694
100	25	/	0.9725
50	100	/	0.9729
100	50	/	0.9709
25	50	100	0.9700
25	100	50	0.9717
50	25	100	0.9701
50	100	25	0.9712
100	25	50	0.9726
100	50	25	0.9717

Table B.10: Accuracy of the MLP classifier for different architectures

KSVM. Four different kernels are tested, the accuracy observed are: 0.567 for linear kernel, 0.973 for polynomial kernel, 0.974 for radial basis function kernel (rbf), and 0.827 for sigmoid kernel.

Appendix C. Extracted Features

This section includes a full table (C.11) of the 119 static features considered in this work and their descriptions. They are also grouped according to their commonality as in Biondi et al. [13].

ID	Abbreviation	Type	Description
Metadata features			
1	**DLL1**	*Boolean*	DLL can be relocated at load time. *Extracted from DLLs characteristics*
2	**DLL2**	*Boolean*	Code Integrity checks are enforced. *Extracted from DLLs characteristics*
3	**DLL3**	*Boolean*	Image is NX compatible. *Extracted from DLLs characteristics*
4	**DLL4**	*Boolean*	Isolation aware, but do not isolate the image. *Extracted from DLLs characteristics*
5	**DLL5**	*Boolean*	Does not use structured exception (SE) handling. No SE handler may be called in this image. *Extracted from DLLs characteristics*
6	**DLL6**	*Boolean*	Do not bind the image. *Extracted from DLLs characteristics*
7	**DLL7**	*Boolean*	A WDM driver. *Extracted from DLLs characteristics*
8	**DLL8**	*Boolean*	Terminal Server aware. *Extracted from DLLs characteristics*
9	**CHKSUM**	*Integer*	The image file checksum.
10	**ADDR1**	*Integer*	The preferred address of the first byte of image when loaded into memory
11	**ADDR2**	*Integer*	The address that is relative to the image base of the beginning-of-code section when it is loaded into memory.
12	**OSVER1**	*Integer*	The major version number of the required operating system.
13	**OSVER2**	*Integer*	The minor version number of the required operating system.
14	**IM_MEM_SIZE**	*Integer*	The size (in bytes) of the image, including all headers, as the image is loaded in memory.
15	**CODE_SIZE**	*Integer*	The size of the code (.text) section, or the sum of all code sections if there are multiple sections.
16	**COMB_SIZE**	*Integer*	The combined size of an MS DOS stub, PE header, and section headers rounded up to a multiple of FileAlignment.
17	**INIT_DATA_SIZE**	*Integer*	The size of the initialized data section, or the sum of all such sections if there are multiple data sections.
18	**UNINIT_DATA_SIZE**	*Integer*	The size of the uninitialized data section (BSS), or the sum of all such sections if there are multiple BSS sections

19	**STACKR_SIZE**	*Integer*	The size of the stack to reserve.
20	**STACKC_SIZE**	*Integer*	The size of the stack to commit.
21	**ALIGN**	*Integer*	The alignment (in bytes) of sections when they are loaded into memory.
Section features			
22	**STAND_SEC**	*Integer*	The number of standard sections the PE holds
23	**NON_STAND_SEC**	*Integer*	The number of non-standard sections the PE holds
24	**RATIO_STAND_SEC**	*Float*	The ratio between the number of standard sections found and the number of all sections found in the PE under analysis
25	**–X_SEC**	*Integer*	The number of Executable sections the PE holds
26	**-W-_SEC**	*Integer*	The number of Writable sections the PE holds
27	**-WX_SEC**	*Integer*	The number of Writable and Executable sections the PE holds
28	**R-X_SEC**	*Integer*	The number of readable and executable sections
29	**RW-_SEC**	*Integer*	The number of readable and writable sections
30	**RWX_SEC**	*Integer*	The number of Writable and Readable and Executable sections the PE holds
31	**CODE_NOT_X**	*Boolean*	The code section is not executable
32	**X_NOT_CODE**	*Boolean*	The executable section is not a code section
33	**NOT_CODE**	*Boolean*	The code section is not present in the PE under analysis
34	**EP_NOT_CODE**	*Boolean*	The entry point is not in the code section
35	**EP_NOT_STD**	*Boolean*	The entry point is not in a standard section
36	**EP_NOT_X**	*Boolean*	The entry point is not in an executable section
37	**RATIO_SEC_EP**	*Float*	The ratio between raw data and virtual size for the section of the entry point
38	**ZERO_RAW_SIZE**	*Integer*	The number of sections possessing raw data size zero
39	**VT_GT_RAW**	*Integer*	The number of sections with a virtual size greater than their raw data size.
40	**MAX_RATIO_RAW_VT**	*Float*	The maximum ratio raw data to virtual size among all sections
41	**MIN_RATIO_RAW_VT**	*Float*	the minimum ratio raw data to virtual size among all sections
42	**ADDR_NOT_ALIGN**	*Boolean*	The address pointing to raw data on disk is not conforming with the file alignment
Entropy features			
43	**ENT_CODE**	*Float* $\in [0; 8]$	The byte entropy of code (.text) sections
44	**ENT_DATA**	*Float* $\in [0; 8]$	The byte entropy of data section
45	**ENT_RSRC**	*Float* $\in [0; 8]$	The byte entropy of resource section
46	**ENT_HEADER**	*Float* $\in [0; 8]$	The byte entropy of PE header
47	**ENT_GLOBAL**	*Float* $\in [0; 8]$	The byte entropy of the entire PE file
48	**ENT_EP_SEC**	*Float* $\in [0; 8]$	The byte entropy of the section holding the entry point of the PE under analysis

Entry byte features			
49 - 112	**BYTE1-BYTE64**	*Integer* $\in [0; 255]$	Values of 64 bytes following the entry point, each byte correspond to 1 feature position
Import functions features			
113	**DLL_IMP**	*Integer*	The number of DLLs imported
114	**FUNC_IMP_IDT**	*Integer*	The number of functions imported found in the import directory table (IDT)
115	**MAL_API**	*Integer*	The number of malicious APIs imported (malicious as defined in [56]). This list includes: "GetProcAddress", "LoadLibraryA", "LoadLibrary", "ExitProcess", "GetModuleHandleA", "VirtualAlloc", "VirtualFree", "GetModuleFileNameA", "CreateFileA", "RegQueryValueExA", "MessageBoxA", "GetCommandLineA", "VirtualProtect", "GetStartupInfoA", "GetStdHandle", "RegOpenKeyExA".
116	**RATIO_MAL_API**	*Float*	The ratio between the number of malicious APIs imported to the number of all functions imported by the PE
117	**FUNC_IMP_IAT**	*Integer*	the number of addresses (corresponds to functions) found in the import address table (IAT)
Resource features			
118	**DBG_DIR**	*Boolean*	The debug directory is present or not
119	**NUM_RSRC**	*Integer*	The number of resources the PE holds

Table C.11: Features description.