Analysis of Machine Learning Approaches to Packing Detection

Charles-Henry Bertrand Van Ouytsel[a,*], Thomas Given-Wilson[a], Jeremy Minet, Julian Roussieau, 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 an obfuscation technique widely used by malware to hide the content and behavior of a program. Much prior research has explored how to detect whether a program is packed. This research includes a broad variety of approaches such as entropy analysis, syntactic signatures and more recently machine learning classifiers using various features. However, no robust results have indicated which algorithms perform best, or which features are most significant. This is complicated by considering how to evaluate the results since accuracy, cost, generalization capabilities, and other measures are all reasonable. This work explores eleven different machine learning approaches using 119 features to understand: 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

1. Introduction

Malware detection represents a significant and expensive problem for current computer security. Since the vast majority of malware detection programs are based on signatures, this means that they are reactive in nature and so the malware writers are typically one step ahead.Further, new malware are constantly being created, the AV-Test Institute [4] registers an average 350,000 new malware (malicious programs) every day.

Malware analysis techniques used to identify, understand, and detect malware are typically divided into two approaches: static analysis and dynamic analysis. Static analysis operates by examining the sample program without executing the program. This can range from very simple analysis of strings or bytes or the code, to complex disassembly and reconstruction of key program behaviours and features. Most malware detection programs use static analysis on simple features such as strings. Dynamic analysis operates by executing the sample program in a protected environment, e.g. a sandbox, and observing the program's behavior such as recording API calls, network activity, process creation, etc. While dynamic analysis allows more in-depth inspection, the cost of starting and running a sandbox is significant. Most detection engines cannot afford to perform dynamic analysis, and so dynamic analysis is usually only done for signature creation or by malware analysis teams to understand new samples.

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

Packing is a widely used technique strategy that is able to evade or disrupt many malware analysis techniques. Packing operates using various methods such as: compressing or encrypting sections of the program; encoding the program in a virtual machine; or fragmenting program behavior. All of these make both static and dynamic analysis much more difficult, since they obfuscate the static features and program behavior, and also make the dynamic execution behavior more complex and harder to observe. Packing techniques are typically implemented with a small stub section of the program that performs the initial decompression/decryption or runs the virtual machine and thus allows the program to execute its behavior. While packing is used by benign programs (typically for compression or to protect intellectual property) WildList [8] states that 92% of packed programs hide harmful behavior.

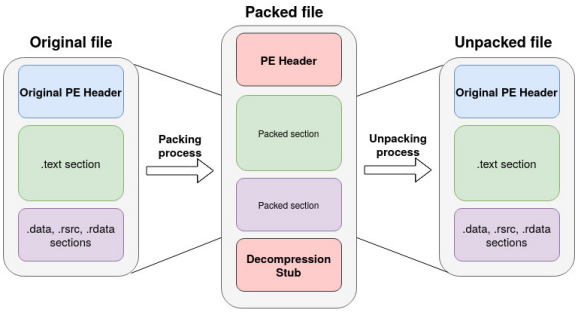

Figure 1: Life-cycle of a packed program

Figure 1 illustrates the life-cycle of a typical packed program. The original program is compressed and the compressed program is stored in the "Packed section". Another section is also added containing the decompression stub routine. When the program executed, the decompression stub decompresses the packed section and then executes the decompressed program. .

To create an effective malware detection or classification engine, it is critical to be able to detect whether a program being analysed is packed. A packed program can be unpacked or treated with greater suspicion to improve the analysis and response. This approach was developed by Perdisci et al. [32] to efficiently distinguish if a program is packed and then deliver the packed programs to a universal unpacker (a program to reverse the packing). Combined with a fast and efficient static malware detector such as the one proposed by Baldangombo et al. [13], a complete malware detection engine can offer high-end performance malware detection and classification.

Packing detection and classification have been widely studied in the literature [27, 27, 22, 24, 34, 37, 39, 38, 32, 22, 24, 37, 38, 34, 41, 19, 10, 14, 18]. These works consider packing detection through a wide variety of approaches to detection and classification such as using rules, heuristics, or machine learning. There is also a wide variety of potential static (and dynamic) features considered for which may be most significant for accurate (and efficient) packer detection and classification. Static features are preferred as the basis for detection and classification since they can be easily extracted with low cost, and thus are widely used as the basis for popular antivirus such as ClamAV [1] or tools such as PEiD [5].

This work addresses the challenges with the current state-of-the-art on packing detection that various ML approaches on various features appear effective, but there is a lack of clarity on which ML techniques and features are the most effective. To address this challenge, this work explores 11 ML algorithms over a variety of data sets with various forms of ground truth, samples, and

to measure different kinds of performance. In particular this works considers 11 different ML algorithms that have been used for packing detection. The ground truth generation for data sets is performed using 5 different labeling approaches and a majority vote among them [18]. The features are also processed in different forms since data processing is a significant factor for the effectiveness of ML algorithms. The 11 ML algorithms are then each tuned according to their own (hyper) parameters to gather information on which parameters are most significant to each algorithm. The 11 tuned ML algorithms are then used to explore which features of 119 static features are the most significant for packer detection. The 11 ML algorithms are then applied to larger data sets to explore their effectiveness considering different metrics. Their resilience against different known packers is also assessed by packing malware ourself. Finally, an economical analysis (how accuracy degrades over time and on new samples) is considered to explore the effectiveness of the 11 ML algorithms over time and as an tool.

The challenges explored in this work can be focused into the following research questions:

RQ1 **Which features are most significant for packing detection?** We explore relative relevance of features present in literature into the classification decision process.

RQ2 **Which ML classifiers are effective for packing detection?** We study performances of our classifiers over large dataset and against specific packers (present or not in the dataset) to discover their strengths and weakness.

RQ3 **Which ML algorithms perform well in long term for packing detection?** We train our classifiers on a huge dataset and evaluate them over different samples recorded later. Observing evolution of their effectiveness reveals us how accuracies of classifiers change over time and which classifiers are more economical in term of training.

1.1. Related Work

This section briefly overviews popular approaches to packing detection.

Syntactic signatures are a popular approach to packer detection that can be extremely cheap to compute and still achieve high accuracy for known packers. Signatures such as those used by YARA [40] typically consider simple static features such as byte sequences, strings, and file features such as size or hash. Although such signatures are effective for exactly matching known samples, they perform less effectively against even minor permutations and can be easily evaded [19].

Entropy metrics have been shown effective in detecting packing since packed files often have much higher entropy than unpacked files. Such approaches were considered in [27, 22, 24, 34, 37, 39, 38]. Lyda et al. [27] were first to address the problem of packing detection using entropy in an academic publication. Their approach was focused on global entropy (computed over the whole program) used as a proxy to detect packing. Their dataset was cleanware for non-packed samples while packed samples where obtained by employing well-known packers on each cleanware sample. Although not verified experimentally, their approach was calculated to target a 3.8% false positive rate and 0.5% false negative rate. The concept was then improved by evaluating entropy for each section of the sample in the paper of Han and Lee [23]. They use 200 cleanware for non-packed samples and 200 malware from a honeypot for packed samples, obtaining a detection rate of 97.5% with false positive of 2.5%. However, methods focusing exclusively on entropy are vulnerable to adversaries. For example, inserting selected set of bytes in an executable in order to keep the entropy of the file low such as in the Zeus malware family [39].

Machine learning (ML) techniques using supervised learning on variety of features extracted statically have been investigated in different works [27, 22, 24, 37, 38, 34, 41, 10, 14, 18]. Perdisci et al. investigated different ML methods in packing detection : Naive Bayes Classifier, decision tree, ensemble of decision trees, k-nearest neighbors and Multi-layer perceptrons. They consider 9 features (also considered in this work) related to section names and properties, import access table (IAT) entries and several entropy metrics in [32] while using a dataset of 5,498 PE files (benign and malicious PE). Multi-layer perceptrons performed well in their work (98.91% test accuracy) compared to other investigated methods. That is why, multi-layer perceptron algorithm is also investigated in our work. We will go deeper by presenting more insights on features relevance, tuning and performing an economical analysis. Our bigger data set will additionally allow more meaningful measure of accuracies. In [33], they add N-grams features to their work and include the packing detection process into a workflow to analyse and detect malwares.

Diverse new features were proposed by malware researchers, for instance in [12, 35, 38] like structural PE attributes, heuristic values, entropy-based metrics or different ratio (raw data per virtual size ratio or the ratio of the sections with virtual size higher than raw data for instance). Although more expensive to extract, some dynamic features have also been studied in the literature, such as using evolution of the import address table (IAT) table during the unpacking process [20] or basing analysis on 'write-then-execute' instructions typical in packed samples [21, 25].

Recent work [18] has surveyed the literature, explored the feasibility of using machine learning techniques for packing detection and produced a list of 119 commonly used features (the same used in this work). Most of this work focus on defining the methodology and on developing efficient techniques to extract interesting features statically. In their paper, Biondi et al. divide them into six categories: Byte entropy, Entry bytes, Import functions, Metadata, Resource and Sections. Moreover, they focus on impact of each categories. Our work investigates this set of 119 features (See Appendix A) by considering all of them as independent. Their paper also discuss few machine learning algorithms based on those features to show the feasibility of the approach (i.e. Bayes classifier, Decision trees and Random Forest).They studied these algorithms over a dataset of 280,000 samples in terms of effectiveness, robustness and efficiency (related to computational cost). They conclude their study by showing that accepting a small decrease in effectiveness can increase efficiency by many orders of magnitude. We expand their set of studied ML classifiers and apply similar analysis on their performance.

The problem of packing detection has also been extended to packing classification [36]. Different classifier have been compared in this perspective using pattern recognition techniques on randomness profiles of packers. In [15, 17, 24], entropy patterns extracted dynamically (i.e., by executing the binary) are considered as an efficient way to classify packers. Knowing the file is packed, the entropy of the whole file is recorded after each JMP instruction until entropy ceases to change. In [16], the authors use similar techniques to determine if multiple packers have been used on the same program. The problem of known packer classification has been demonstrated as a trivial task in [9]. Using samples from each packer (class), their classifier achieved precision and recall greater than 99.99% for each class. However, the authors have highlighted challenges related to problems of packing detection.

Unsupervised learning has recently also been investigated [30] to build clusters related to packers and identify variants of known packers or new packers. However, this is beyond the scope of this work, as the focus here is on exploring supervised learning algorithms.

The structure of the paper is presented hereafter. Section 2 recalls background on the machine learning algorithms used in this work. Section 3 presents the construction of datasets used in this

work, the methodology applied to answer our research questions and configurations used for our ML algorithms (i.e.: pre-processing and hyper-parameter tuning). Section 4 explores the relevance of different feature in the decision process of classifiers, including feature selection methods and principal component analysis. Section 5 evaluates the effectiveness of the ML algorithms on a large data set. Section 6 explores the economics of the trained packing detectors over a chronological data set. Section 7 discusses the validity of the results and some observations on dynamic features. Section 8 concludes.

2. Background on Machine Learning algorithm

This section recalls useful background information about eleven machine learning algorithm used in this paper.

*2.1. K-Nearest Neighbors (*KNN*)*

The K-Nearest Neighbors (KNN) ML algorithm considers each input data as a point in a feature space with a particular label. When given a new data point to classify KNN will find the K nearest points in the feature space in terms of a defined distance (e.g. Euclidean distance) and classify the new data point as a plurality vote of its neighbors within the defined distance. In the case of $k = 1$, the input is simply assigned the label of its nearest neighbor.

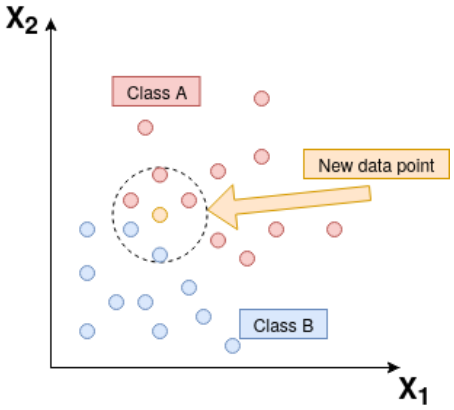

Figure 2: Binary classification involving two features (X_1 and X_2) and using a 5-Nearest Neighbors classifier

An example is shown in Figure 2 with two labels (red and blue) where the new data point (orange) will be assigned the color red according to the majority vote. This classification algorithm requires complete information about the whole training data set to classify each new data point and thus requires significant memory use in practice for larger data sets.

*2.2. Naive Bayes Classifiers (*GNBC *&* BNBC*)*

Naive Bayes classifiers (NBC) are probabilistic models based on Bayes theorem:

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

where C_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 classification use Equation 1 to evaluate probability to observe label C_k for a sample according to value of feature vector x_j. These estimated probabilities are then combined to construct the classifier as in Equation 2 where $\hat{y}$ is the estimated label of the sample:

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

The naive adjective of the classifier comes from the assumption that each data point's feature is independent from each other feature. Although this assumption does not usually hold in general NBC is known to work well in practice.

Two variant of the NBC are investigated in this paper: Gaussian (GNBC) based on normal distribution and Bernouilli (BNBC) where features are considered as independent booleans. Although NBC are among the simplest ML algorithms, they offer good accuracy in some areas and have low performance costs.

*2.2.1. Linear Models (*LR *&* LSVM*)*

In linear models (Lin) ML algorithms a model is built from training data as a linear combination of features. The model has the form given in Equation 3 below:

$$y = x[1] \cdot w[1] + x[2] \cdot w[2] + ... + x[p] \cdot w[p] + b \tag{3}$$

where y is the prediction calculated from $x[i]$ is each feature indexed by i in the feature vector x and $w[i]$ is the learned weight for each i in the length of the feature vector and b is the offset.

Since our focus is binary classification, the predicted value is set to a threshold at zero. Thus, a y value greater than zero corresponds to label 0 while a value less than zero corresponds to label 1. Therefore, the decision boundary can be represented as a line or a plane that separates the data in two classes. Since Lin are a very large family of algorithms this paper considers two forms of Lin: logistic regression and linear support vector machines.

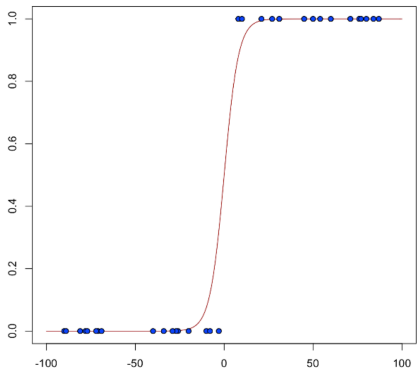

Figure 3: Logistic regression [3]

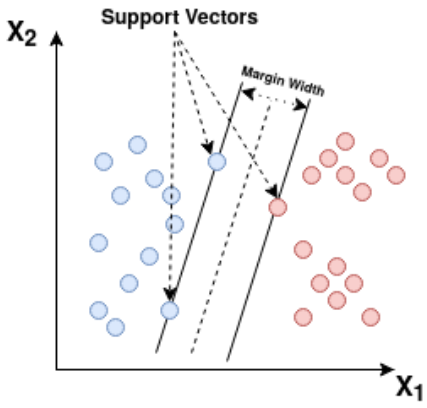

Figure 4: Linear support vector machine.

Logistic regression (LR) is based on the logistic function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \ . \tag{4}$$

that is also illustrated by the red curve in Figure 3. To obtain optimal weights w for LR, maximum likelihood estimation is used. This involves finding weights $w[i]$ that maximize the log-likelihood function defined as follows:

$$L(w) = \prod_{i=1}^{n} \sigma(w^T x_(i))^{y_i} \cdot [1 - \sigma(w^T x_(i))]^{1-y_i} \ .$$

A Linear support vector machine (LSVM) is based on parallel hyperplanes that separate the classes of data, so that the distance (or margin) between them is maximised. Samples used to construct the two planes are called support vectors. The equation used to define these is as follows:

$$w^T x - b = 1 \quad \text{and} \quad w^T x - b = -1 \ . \tag{5}$$

Figure 4 shows the two planes (here two lines), support vectors used to build the planes, and the margin width that we try to maximize here.

*2.3. Tree-based Models (*DT, GBDT, RF, *&* DL*8.5)*

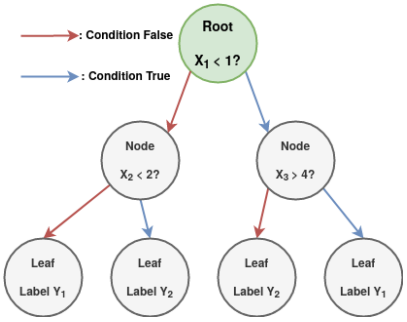

Figure 5: Example of a simple decision tree

Decision trees (DT) are classification models based on the sequential application of simple binary rules. They are structured as a directed tree as illustrated in Figure 5. Starting at the root (in green), the tree is traversed towards a leaf (containing a label) by selecting an edge at each node according to a simple rule (e.g. in Figure 5, the rule at the root is "if X_1 of the sample is smaller than 1 go to right branch, else go to left branch"). In the building of a decision tree, all features are considered. Different combinations of splits are considered and the best one is selected according to a cost function (such as gini impurity or entropy). The choice of effective split being of first importance in term of tree size and cost, although features relevance and features selection is an important concern which can assist with pruning or depth limitation.

Simple DT models can be combined to construct more complex classifiers solving variance and bias that single DT can suffer from. The concept is to combine many weak learners to shape a stronger one at the cost of more resource consumption. Two of these classifiers, known to have

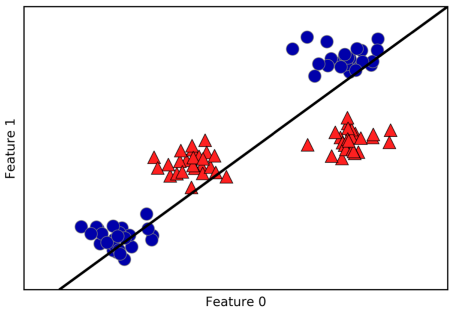

Figure 6: Decision boundary found by Lin [28].

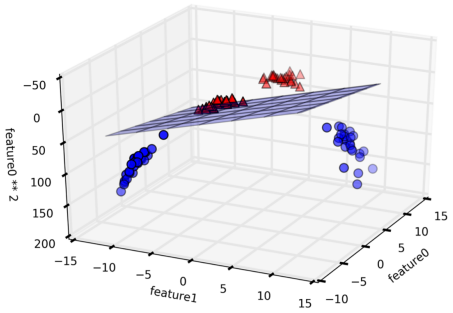

Figure 7: Decision boundary found by KSVM [28].

demonstrated good performance on a diversity of data sets are considered in this work: Random Forests and Gradient Boosted Decision Trees.

Random Forests (RF) represent the combination of slightly different DT. While a single DT may be very effective there is a risk be overfitting the training data. RF therefore address this risk by considering outcomes from several DT, resulting in a reduced overfitting while maintaining the predictive power of each tree. The random nature of this algorithm resides in the creation of the DT, which are built upon a randomly generated set of samples (this process is called bagging). Each tree is then trained individually and their outputs are aggregated with each other in a majority vote.

Gradient Boosted Decision Trees (GBDT) is another approach to address overfitting that works by using a technique called boosting (while RF uses bagging). In this process, each tree is constructed sequentially with each subsequent tree trying to correct the errors of the previous trees. Typically GBDT start with extremely shallow trees, often called weak learners, which provide good prediction only on some parts of the data. Increasingly trees are then grown iteratively to increase the global performance by reducing the loss margin. In comparison to RF, GBDT produces shallower trees. Hence, the model is more economical in terms of memory but harder to tune.

In addition to the traditional approach, a recent version of DT is also considered in this work, namely DL8.5 [29], working only with binary data (implementation available at [6]). DL8.5 differs from classical approaches by using a cache of item sets combined with branch-and-bound search in order to create less complex trees and avoid unnecessary computations.

*2.4. Kernelized Support Vector Machines (*KSVM*)*

Kernelized Support Vector Machines (KSVM) are an extension of Lin which allows separation of data not linearly separable by using hyper-planes. In KSVM low dimensional data are converted to higher dimensional data using a kernal function (e.g. polynomial, Gaussian, Sigmoid, etc.) and the planes used to separate the data are now higher dimensional hyperplanes. For example, on Figure 6 is shown how Lin would try to find planes to classify the red and blue data points. On Figure 7 the data points are converted to a higher dimension and a higher dimensional hyperplane is used to separate the points, leading to a much more effective separation.

Although KSVM work well on a variety of datasets, they are known to suffer from performance degradation when scaled to a large number of samples.

2.5. Neural Networks (MLP)

Neural Networks are mainly used in deep learning and have been employed in thousands of different classifiers for specific purposes. This work uses the most popular variation of NN: Multi-Layer Perceptrons (MLP) algorithm. MLP can be viewed as a generalization of Lin that is constituted with various layers of neurons. Each neuron is a unit taking its input applying a weighted sum of its features and passing it through a non-linear function before forwarding it to the subsequent layer as illustrated in Figure 8.

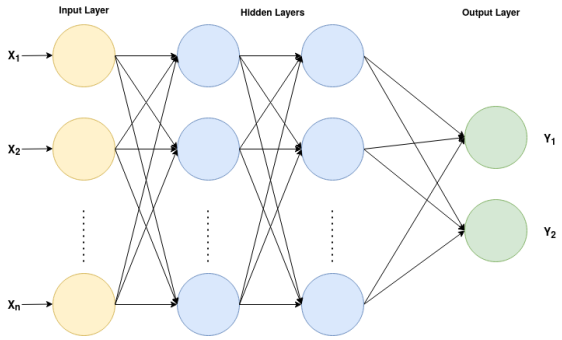

Figure 8: Multi-Layer Perceptron with two hidden layers.

Each neuron of each layer with input features X will apply the following equation to calculate its output y:

$$y = f(x[0] \cdot w[0] + w[1] \cdot x[1] + ... + w[p] \cdot x[p] + b) \tag{6}$$

where f is a non-linear function (e.g. logistic function like Equation 4 or 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, then there may be one or more hidden layers representing intermediate processing steps, and eventually predictions are made on the output layer.

The main advantage of using NN is that they have demonstrated their efficiency to deal with large data sets and come up with incredibly complex models at the cost of training time, wise parameter tuning and accurate pre-processing.

3. Experimental Setup

This section details the experimental setup used in this work. This is divided into three areas: construction of data sets; experimental methodologies; and pre-processing and hyper-parameter tuning of the ML algorithms.

3.1. Data Sets and Labeling Tools

All data sets are built from a feed of malware provided by Cisco that offers 1,000 new (assumed) malware samples per day [7]. The construction of data sets begins by taking samples of malware from the feed that are collected over a given time period. For each sample, 119 static features are

(attempted to be) extracted[1]. These features are a collection of all those used in other works and identified in the work of Biondi et al. [18].

Once gathered, these samples need to be labeled as packed or not packed to obtain a ground truth for training ML classifiers. In this work a similar approach to the one of VirusTotal was taken; to employ multiple detector engines for classification and then build on these results. There were five classification engines used, described below.

- **PEframe** [11] This is an open source tool written in Python and freely available to perform static analysis of portable executable (PE) files, object code, DLLs and others. Its utilization will be restricted to packer detection but the tool can also be used to obtain other digital forensics information such as macros, anti-forensics techniques, etc.
- **Manalyze** [26] This is a static analyzer mainly used by large enterprises and composed of several plugins. Its packer detection plugin is based on signatures and custom heuristics. Signatures are generally related to names of PE sections (e.g. UPX renames some of the section after packing as UPX0, UPX1, etc.) while heuristics are more related to anomalies in the PE structure (unusual section names, sections both writable and executable, etc.) or entropy-based features.
- **PEiD** [5] This is a widely known detector, freely available and signature based. Its signatures only contains low-level byte patterns which can match either at the entry point or more generally anywhere within a PE file.
- **DIE** Detect-it-Easy [2] is an open source architecture of signatures written in javascript which allows more complex and fine tuned signatures.
- **Cisco** This was an in-house engine that Cisco used on some samples. This is not publicly available and was only used for some of experiments due to limited availability.

A sample is considered as packed if a majority of detectors label the sample as packed, in this case three out of five detectors. When the Cisco detector was unavailable, a threshold of two out of four was used.

A first data set TUNE of 43,092 samples is used for parameters tuning and finding optimal representation of the dataset (by feature selection and Boolean conversion). This data set consists of samples collected from 2019-06-15 to 2019-07-28 and according to the labeling has 10,009 packed samples and 33,083 not packed samples. Labels for all five label engines were available for this data set.

A second much larger data set BIG of 95,876 samples is used for the broader experimental results in this work. This data set consists of samples collected from 2019-10-01 to 2020-02-28 and according to the labeling has 23,894 packed samples and 71,982 not packed samples. Since labels from the Cisco engine were exclusively available for the samples in TUNE, this data set was generated to carry out larger experiments on the machine learning and features, but had reduced label information. An extended data set denoted BIG$+_f$ is the data set BIG with 500 samples taken

[1]Occasional errors in the extraction software caused a few samples to be discarded, although this is an insignificant number and so ignored for the rest of this work.

at random from TUNE and packed with the packer f added to BIG. The label information is the same for samples originally from BIG, the extended samples have known packer.

Specific data sets SELFPACK $_f$ of approximately 12,036 samples are samples collected from 2020-06-15 to 2020-06-30 and packed ourself with known packer f .

A third data set CHRONO of 37,794 samples is used for the economical analysis results in this work. This data set consists of samples collected from 2020-04-01 to 2020-05-26 and according to the labeling has 8745 packed samples and 29,049 not packed samples. This data set has also reduced label information as BIG.

3.2. Methodology

To address the research questions effectively it is necessary to properly pre-process the data sets and tune the ML algorithms appropriately. This is detailed in Section 3.3, however an overview is as follows. The pre-processing includes Boolean conversion of non-Boolean data, bucketing, and normalization of feature values. Once the data sets have been pre-processed, the hyper-parameters of all the ML algorithms are tuned using TUNE to find the best configurations for later use.

The methodology used to address the various research questions in this paper is split into several parts based upon the research question being considered.

- **RQ1** is addressed by exploring which features are the more significant in different ML algorithms. Due to there being many different ML algorithms several approaches are used for reducing the number of features including: feature coefficients, K-best features, and principle components analysis. These approaches are applied to all 11 ML algorithms using TUNE and experimentally evaluated with 10-fold cross-validation and 90% of the data for training and 10% for testing. Details on the approach and results are presented in Section 4.

- **RQ2** is addressed by taking all the ML algorithms along with all the lessons learned on data processing, ML tuning, and feature selection to evaluate their overall effectiveness on the larger data set BIG. The analysis is performed by using 10-fold cross-validation on BIG and taking 90% of the data for training and 10% for testing. Further, the adaptability of the 11 ML algorithms is evaluated by comparing the accuracy when the data set is extended with more samples of a specific packer. This evaluates the adaptability of the algorithms as their training data changes, and to evolutions in packer families. Details of the results and experiments are presented in Section 5.

- **RQ3** is addressed by training the ML algorithms (developed and tuned as above) on BIG and then evaluating them on CHRONO. Observe that CHRONO is chronologically after BIG and thus indicates a representation of the evolution of malware over time (there is a time gap of a month between last malware of BIG and first malware of CHRONO to focus on packed samples which would be release later than the training set). To further assess the economic and chronological performance of the ML algorithms, the testing on CHRONO is considered over smaller chronological periods (four periods are considered, each one consisting of two weeks). This evaluates the effectiveness of the various ML algorithms over time and allows to calculate the effective cost of retraining to maintain a desired level of accuracy. Note that since the experiments here consider using data sets from various times, the whole early data set is used for training and not cross-validation is applied. Details of the approach and results are presented in Section 6.

All the ML algorithm implementations used are from the scikit-learn library [31] version 0.23.2. except the DL8.5 algorithm [29] where the implementation on github is used (that implement the same functions as the scikit interface). All experiments here 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.

3.3. Pre-processing and Hyper-Parameter Tuning

This section overviews the data pre-processing and ML algorithm hyper-parameter tuning used in the later experiments. Pre-processing includes the consideration and application of Boolean conversion, standardization and normalization of the data. For each algorithm, a grid search is applied to find the best combination of pre-processing and hyper parameters.

3.3.1. Data Pre-processing

- 42% have value 1024 (in blue).
- 34% have value 4096 (in orange).
- 16% have value 512 (in green).
- 7% have value 1536 (in red).
- 1% have other values (in yellow).

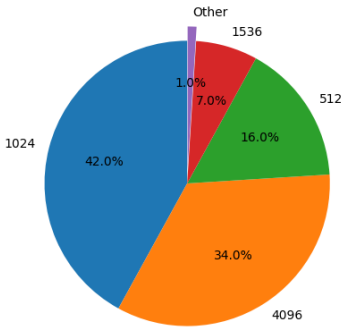

Figure 9: Distribution of values for feature 16 (total header size in bytes).

Of the 119 static features considered for each sample, only 16 of them are Boolean values. Some of the ML algorithms such as DL8.5 requires Boolean features (for details see Section 2). Therefore it is necessary to convert non-Boolean values to Boolean values for use in these algorithms.

This conversion begins by observing the value distribution of TUNE features. For example, Figure 9 shows the distribution of feature 16; namely header size in bytes. Bucketing is then applied which gathers similar values into buckets and hence reduces the range of possible values. For example: value 1 is mapped to 1024, 2 is mapped to 4096, 3 is mapped to 512, 4 is mapped to 1536 and 0 is mapped to other possible values. Then one-hot encoding is used to convert these bucket values into Boolean values. Therefore in the case of the feature 16, five Boolean features will be used in the new dataset to represent header size. This method induces an important increase in term of the number of features and it is therefore important to efficiently limit the number of buckets for each feature. Details on the results of this pre-processing via bucketing and conversion to Boolean values is given in Appendix B.

Initially Boolean conversion is used only in the BNBC and DL8.5 ML algorithms which requires Boolean data. However, since other algorithms could benefit from this conversion, Boolean features conversion is also tested on all the ML algorithms to observe the impact. Final choice for each algorithm is investigated in Section 3.3.2.

To work most efficiently, some ML algorithms require their data to be normalized (values $\in [0;1]$) or standardized (considered as a Gaussian distribution $\in [-1;1]$). For example, GNBC requires normalization to produce an efficient and coherent model. Like for the Boolean conversion, although many ML algorithms do not require normalization or standardization, they may be able to benefit from this data pre-processing. This is investigated below.

3.3.2. Hyper-Parameters Tuning

Each of the 11 ML algorithms has their own hyper-parameters that can be tuned to improve their performance on different kinds of data sets and classification problems. In this section a grid search is applied to each ML algorithm to obtain the best combination of pre-processing and hyper-parameters. The best combination is chosen by accuracy, with equal accuracy outcomes decided by training time. Each combination has been tested with 10-fold cross-validation on Tune (90% for training and 10% for testing).

Together with the three types of pre-processing (Boolean conversion, normalization, and standardization) the hyper parameters investigated for each ML algorithm are listed below.

- KNN: Number of neighbors $[1;30]$.
- LR and LSVM: Loss function (hinge loss or squared hinge loss).
- 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 $[1;12]$.
- DL8.5: Maximal depth of the tree $[1;10]$ (limited for important training time purpose).
- RF and GBDT: Number of estimators $[2;150]$, best parameters of DT are also used for individual tree.
- 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).
- KSVM: Kernel used (linear, polynomial, rbf or sigmoid).

Note that NBC and BNBC do not have hyper-parameters of interest and are only tested for appropriate data pre-processing.

Figure 10 shows a summary of different trained classifiers with their best accuracy (according to all the possible pre-processing and hyper-paramater tuning) on Tune. This provides a baseline for later comparison as well as an overview of the potential accuracy achievable for each of the 11 ML trained classifiers. The best choices of data pre-processing and hyper-parameter selection for each ML algorithm as listed below.

- KNN: Boolean conversion, 16 neighbors.
- BNBC: Boolean conversion.
- GNBC: Standardization.
- LR: Standardization and square hinge for loss function
- LSVM: Boolean conversion and square hinge for loss function

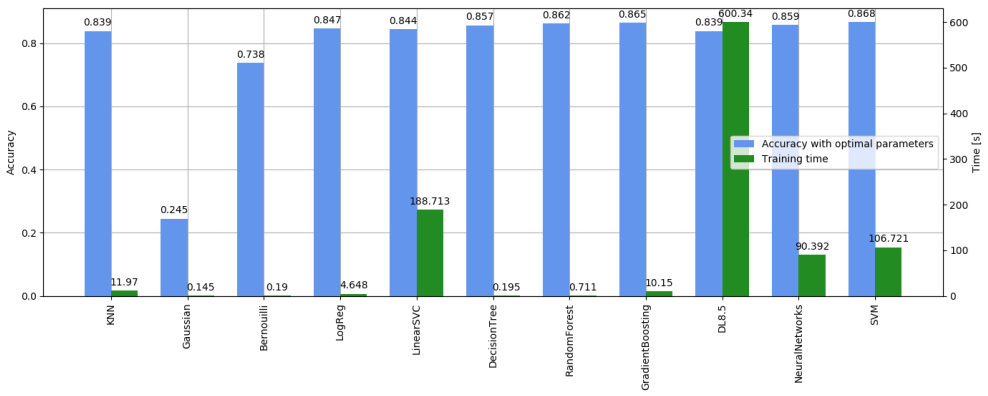

Figure 10: Accuracy and training time for all 11 ML trained classifiers with optimal pre-processing and hyper parameters. 10-fold cross validation has been used with 90%-training and 10%-testing using the TUNE data set.

- DT: No pre-processing, entropy as criterion to measure quality of split, 10 as minimal number of samples required for a leaf and 6 as the maximal depth of the tree.
- DL8.5: Boolean conversion, maximal depth of 10.
- RF and GBDT: No pre-processing, 20 trees as number of estimators.
- MLP: Boolean conversion, 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: Standardization, Gaussian radial basis functions (rbf) as kernel.

The above pre-processing and hyper-parameter configurations are used for all later experiments in this paper.

4. Features Analysis

This section addresses **RQ1** (Which features are most significant for packing detection?) by considering the relevance and significance of the 119 different features explored in the literature. In Biondi et al. [18] these features are grouped into six categories: Metadata, Section, Entropy, Entry bytes, import functions and Resource features. See Appendix A for the full list of features and their categories). The relevance and significance of the features is considered using three different approaches: analyzing the weights and choices from the classifiers produced with the ML algorithms (LR, LSVM, DT, GBDT, & RF); using iterative K-best feature selection; and principal component analysis. The accuracy of the classifier is used to measure the relevance and significance of features, and all analysis in this section is performed on TUNE using 10-fold cross-validation with 90% of TUNE used for training and 10% for testing.

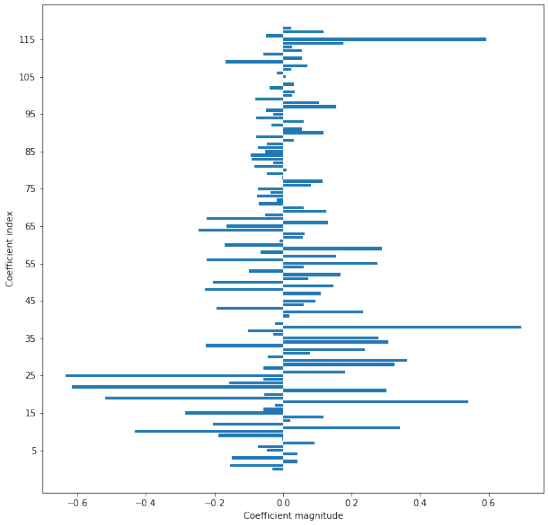

Figure 11: Coefficient magnitudes for the LR classifier. 10 most important features by order of relevance are: 39, 116, 19, 30, 12, 29, 35, 22, 60, & 36.

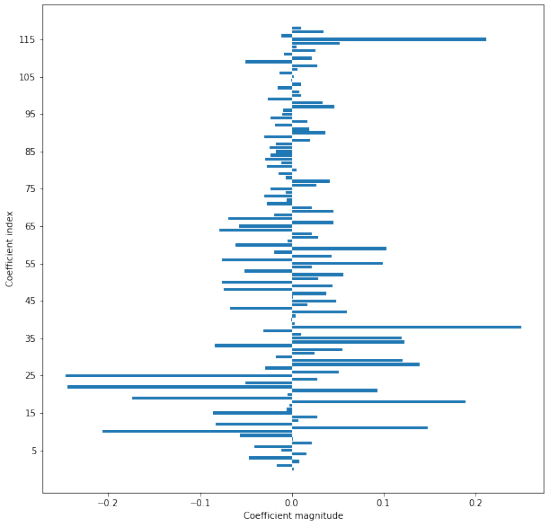

Figure 12: Coefficient magnitudes for the LSVM classifier. 10 most important features by order of relevance are: 39, 116, 19, 12, 29, 35, 30, 36, 60, & 56.

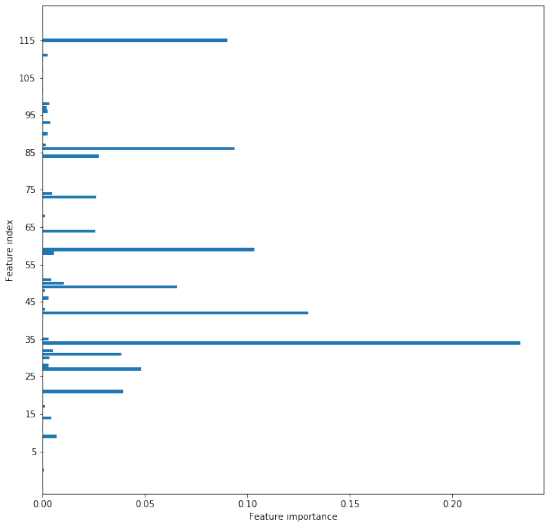

Figure 13: Feature importance for the DT classifier. 10 most important features by order of relevance are: 35, 43, 60, 87, 116, 50, 28, 22, 32, 85, & 74.

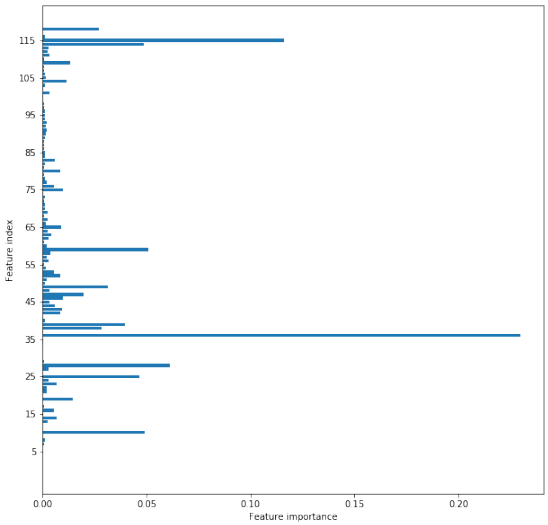

Figure 14: Feature importance for the GBDT classifier. 10 most important features by order of relevance are: 37, 116, 29, 60, 11, 115, 26, 40, 50, 39, & 119.

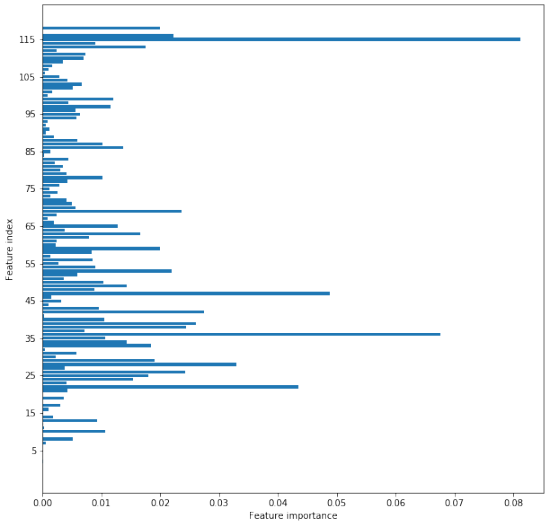

Figure 15: Feature importance for the RF classifier. 10 most important features by order of relevance are: 116, 37, 48, 23, 29, 43, 40, 39, 27, 70, & 117.

4.1. Overview of Feature Relevance

One approach to understanding which features are most relevant and significant is to examine the classifiers produced by different ML algorithms. For LR, and LSVM trained classifiers the coefficients magnitude is one way to examine which features are most significant. For DT, RF, and GBDT the position in the decision tree(s) also implies relevance and significance of features and provides another insight. Observe that since these two different groups of ML algorithms (Lin-based Lin, LR, & LSVM, and DT-based DT, RF, & GBDT) operate in different ways, they provide different kinds of insights.

The following features appear relevant and significant to ML classifiers based on this analysis approach (presented in feature index order). Note that an overview of feature significance for each of the classifiers considered here is presented in Figures 11 to 15.

- Feature 19 corresponds to the size of the stack to reserve. Since unpacking implies in-memory operations and execution, this feature appears to be intrinsically linked to packing behavior. This feature is mainly significant in Lin-based models.
- Feature 29 corresponds to the number of readable and writable sections the PE holds. Typically, a packer will unpack its code in specific sections which means it will re-write some of its previously encrypted sections. Therefore it is expected that this feature is in the top ten used features of LR, LSVM, RF and GBDT.
- Feature 35 corresponds to the entry point not being in a standard section. This can could happen when the entry point which corresponds to the unpacking stub is in a dedicated section. This feature is in the top ten features of LR, LSVM and DT.
- Feature 37 corresponds to the ratio between raw data and virtual size for the section of the entry point. A huge ratio can be link to a section that will be modified during execution,

Classifier	Selection	Best # features	Old accuracy	New Accuracy	Old time	New time	Ratio
LR	Iterative	103	0.8476	0.8480	4.385	3.462	(-)446
LSVM	Iterative	84	0.8440	0.8470	281.243	128.804	(-)152
DT	K best	16	0.8572	0.8556	0.1797	0.0218	470
RF	Iterative	24	0.8627	0.8622	0.6423	0.2843	961.68
GBDT	K best	28	0.8657	0.8656	9.6522	2.6470	6283

Table 1: Best results for feature selection based on ratio value.

in the case of the entry point section this is quite suspicious. This feature is one of the top features of RF and GBDT.

- Feature 39 corresponds to the number of sections having their virtual size greater than their raw data size. A packed executable will typically create this change in section size due to modifying data inside a section. This is a top ten features of LR, LSVM, RF and GBDT.
- Feature 43 corresponds to the byte entropy of code (.text) sections. Since packing implies generally compression/encryption, entropies of code sections is generally high. This explains why this feature is often used in literature [27, 22, 24, 34, 38].
- Feature 60 represent the 12th-byte value following the entry point. The significance of this byte specifically is interesting to observe and is in the decision process for LR, LSVM and DT.
- Features 115-116 are related to the number of API calls imported by the binary and often used in malicious software according to [41]. Feature 115 corresponds directly to the number of API calls imported while Feature 116 is related to the ratio between malicious API calls imported and the total API calls imported. These API calls are not intrinsically malicious but allow easily alteration or obfuscation of control flow of a running program. This can be by allocating memory to store unpacked files, getting address of specific API function call after unpacking, etc. Feature 116 is part of top ten used features of all explored methods while Feature 115 is also utilized in GBDT.

4.2. Selecting Features

Another approach to better understand the relevance and significance of features is to explore how to reduce the features used by some of the ML algorithms. Here this is done in two ways: the first is by exploring the K-best features extracted from the initial models; and the second is by iterative increase where (increasing iteratively) K features are chosen and the intersection of the best features is kept over iterations.

To select the K-best features for models, the scikit implementation (namely SelectFromModel function) was used. A threshold is used to decide if a feature is sufficiently important to be kept, here the threshold has been incremented progressively, and all feature combinations tested with combinations that did not decrease accuracy of the classifier by more than 5% kept. For the list of combinations of features satisfying this requirement, an indicative time-to-accuracy ratio has been computed which represents the ratio between the percentage of time saved and the percentage of accuracy lost. Observe that this process of selection is only possible on estimators making direct use of coefficient or feature importance namely: LR, LSVM, DT, RF and GBDT.

Classifier	Best # components	Old accuracy	New accuracy	Old time	New time (s)	Ratio
KNN	71	0.8391	0.8411	11.97	**0.634462**	(-)397
GNBC	14	0.2458	**0.8086**	0.145	0.0201	(-)0.37
LR	99	0.8476	0.8513	4.648	3.808	(-)41
LSVM	101	0.844	0.8458	188.713	153.405	(-)87.72
DT	58	0.8572	0.8464	0.195	0.0829	46
RF	11	0.8627	0.8589	0.711	0.6717	13
GBDT	18	0.8657	0.8622	10.15	5.13846	122.33
MLP	57	0.8592	0.8628	90.392	**85.882**	(-)12
KSVM	101	0.8682	0.8682	106.72	**92.990**	1117

Table 2: Top result from applying PCA and keeping only iterations improving time and finally sorting them by accuracy.

A summary of the feature relevance process outcomes can be seen in Table 1. These results indicate that feature selection is in general beneficial for every tested ML classifier measured by the high value of time-to-accuracy ratios For ML approaches such as LSVM which take a significant training time, feature selection not only reduces training time but also improves classifier accuracy. A small improvement in performance is also observed for LR. Generally the results indicate that Lin-based models tend to use more features than decision tree based models. This confirms observations in Section 4.1.

4.3. Principal Component Analysis

Principal component analysis (PCA) aims to reduce the dimension of a feature space by grouping features together to constitute new components. Each new component is an independent novel linear combination of some prior features. These are ranked by importance, the first component trying to capture as much information (i.e. variance of the data) as possible.

Using different number of principal components, best sets of these new features in terms of accuracy were selected. Table 2 summarizes best results from applying PCA over Tune before training of our ML classifiers. Note that since PCA expects standardization as a part of the process, DL8.5 and BNBC are ineligible since they require binary data.

An overview of the results of PCA can be seen in Table 2. Although computation times generally improve in every case, three types of results could be distinguished. First, for KNN and GNBC the number of features is substantially decreased and a significant improvement can be observed in accuracy. These methods being more based on distance and statistical distribution, it appears intuitive creating principal components will benefit to them.
Second, the impact of PCA on algorithms like LR, LSVM, MLP and KSVM seems less significant: the number of features didn't decrease significantly, and accuracy performance did not reveal substantial improvement. These algorithms already attach importance to different feature and since PCA reduces dimensionality by employing linear combinations over data, these are somewhat redundant in their effect. Although this limited the improvements in accuracy, improvement in training time can be observed for MLP and KSVM. Third, tree-based ML algorithms such as DT, RF and GBDT dramatically decreases the number of features without impacting accuracy. Since trees-based algorithms derive their cogency from the diversity of features to perform their split, applying PCA reduces dimensionality and creates more complex feature to split upon for tree-based models.

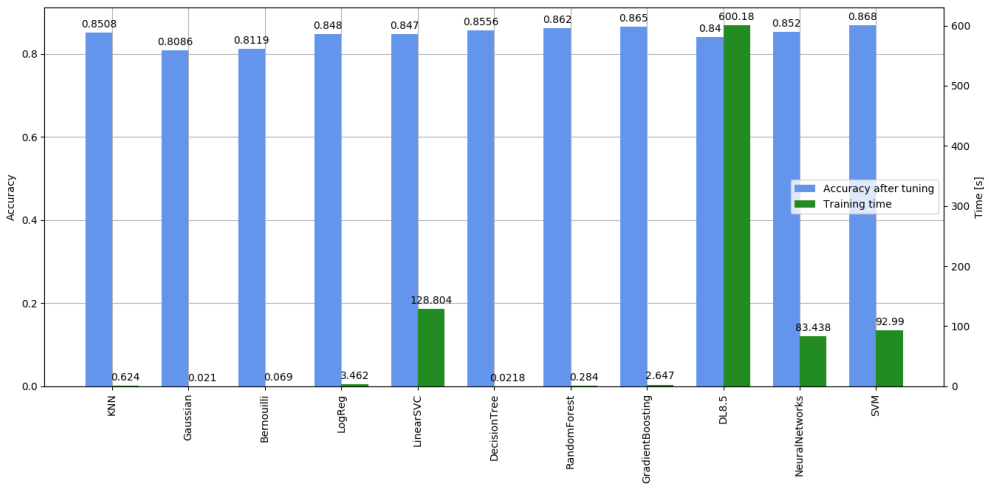

Figure 16: Accuracy and training time for all ML classifiers after processing of features.

Ultimately, PCA will be kept as a step for later application of KNN, GNBC, MLP and KSVM since improvements in accuracy or training time justifies always using PCA.

4.4. Accuracy with Modified Features

Figure 16 illustrates the accuracy and training time of all ML classifiers after applying the best choices from the above analysis. Feature selection is applied to LR, LSVM, DT, RF and GBDT, and PCA is applied to KNN, GNBC, MLP and KSVM. Using feature selection/PCA reduces the number of necessary features stored and used for training. Most ML algorithms can significantly reduce their number of features while maintaining their accuracy.

Regarding **RQ1**, we observe that a small number of features are clearly more influential than others. These are: the stack reserve (Feature 19), number of readable and writable sections (Feature 29), non-standard entry point section (Feature 35), raw and virtual data size of sections (Features 37 & 39), byte entropy of `.text` section (Feature 43), 12th entry point byte (Feature 60), and malicious API calls (Features 115-116). In Lin classifiers, all features have weighted contributions to the classification process even if some of them are significantly lower weight than others. By contrast, in tree-based models, classifiers take advantage of a small amount of features to classify a sample. These properties of the classifiers and the diversity of the features and their provenance correlates with prior results showing different features and approaches can be effective and play a role in classification even if many are not significant independently.

5. Classifier Effectiveness

This section addresses **RQ2** (Which ML classifiers are effective for packing detection?) by examining various metrics for classification over a large data set. The adaptability of the classifiers is also evaluated by testing the trained classifiers on specific packers.

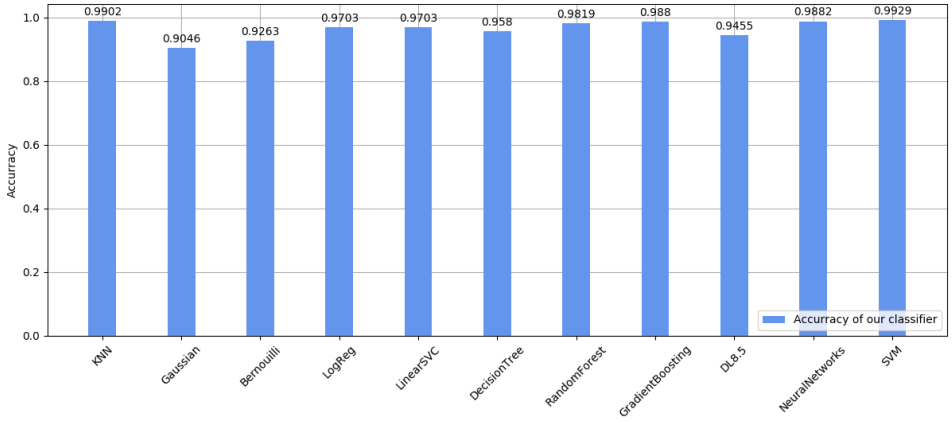

Figure 17: Accuracy for all 11 ML trained classifiers after tuning tested on BIG.

Classifier	precision_p	precision_{np}	recall_p	recall_{np}	$F\text{-score}_p$	$F\text{-score}_{np}$	precision_{wa}	recall_{wa}	$F\text{-score}_{wa}$
KNN	0.9746	**0.9955**	**0.9867**	0.9913	0.9806	0.9934	0.9903	0.9902	0.9902
GNBC	0.7882	0.9478	0.8487	0.9234	0.8173	0.9354	0.9077	0.9046	0.9057
BNBC	0.8091	0.9728	0.9219	0.9278	0.8619	0.9498	0.9321	0.9264	0.9279
LR	0.9415	0.9801	0.9407	0.9803	0.9411	0.9802	0.9704	0.9703	0.9703
LSVM	0.9393	0.9809	0.9432	0.9795	0.9412	0.9802	0.9704	0.9704	0.9704
DT	0.9119	0.9738	0.9225	0.9700	0.9172	0.9719	0.9586	0.9581	0.9582
RF	0.9733	0.9847	0.9543	0.9912	0.9637	0.9880	0.9819	0.9820	0.9819
GBDT	0.9844	0.9892	0.9676	0.9948	0.9759	0.9920	0.9880	0.9880	0.9880
DL8.5	0.9473	0.9451	0.8276	0.9847	0.8834	0.9645	0.9456	0.9456	0.9443
MLP	0.9729	0.9934	0.9805	0.9908	0.9767	0.9921	0.9882	0.9882	0.9882
KSVM	**0.9875**	0.9947	0.9843	**0.9958**	**0.9859**	**0.9952**	**0.9929**	**0.9929**	**0.9929**

Table 3: Detailed metrics for all 11 ML trained classifiers after tuning tested on BIG.

5.1. Evaluation of Classification Metrics

To assess the performance of the 11 ML trained classifiers an analysis is performed over the larger BIG. For the results in this section 10-fold cross-validation is performed with 90% of BIG used for training and 10% for validation.

An overview of the average accuracy for each classifier can be seen in Figure 17. Observe that in all cases except for BNBC the ML classifier accuracy is higher than when performing the parameter tuning (see Section 4.4 and Figure 16 for details). These results indicate that the choice of parameters appears reasonable and that they are not over-fitted to the data set used for the parameter tuning (TUNE).

A more detailed exploration of the metrics for evaluation the different classifiers is presented in Table 3. For each classifier the average of the following metrics is presented.

- The precision of a class A is defined as the ratio of samples correctly labeled A over all samples labeled A.
- The recall of a class A is defined as the ratio of samples correctly classified as A over all samples belonging to class A.

Classifier	UPX		kkrunchy		MPress		TElock		PEtite	
	BIG	BIG+$_{UPX}$	BIG	BIG+$_{kkrunchy}$	BIG	BIG+$_{MPress}$	BIG	BIG+$_{TElock}$	BIG	BIG+$_{PEtite}$
KNN	0.9873	0.9756	0.7682	0.968	0.9264	0.9728	0.7606	0.8359	0.016	0.9277
GNBC	0.9873	0.9955	0.5133	1.0	0.3676	0.9689	0.3204	0.3735	0.0115	0.0246
BNBC	0.9924	0.9822	0.7433	0.972	0.875	0.9689	0.5922	0.9486	0.1598	0.7701
LR	0.9600	0.9977	0.792	0.988	0.5755	0.9844	0.1837	0.4940	0.0098	0.8998
LSVM	0.9623	0.9977	0.816	0.982	0.6337	0.982	0.1739	0.8675	0.009	0.8456
DT	0.9911	0.9955	0.01	0.12	0.9709	0.9903	0.426	0.4466	0.003	0.08
RF	0.9955	1.0	0.34	1.0	0.9709	0.9903	0.1482	0.7035	0.0	0.0476
GBDT	0.9955	1.0	0.002	0.998	0.9709	1.0	0.3913	0.91	0.0	1.0
DL8.5	0.75	0.8824	0.03	0.836	0.93	0.9573	0.6267	0.1660	0.0268	0.267
MLP	0.9848	1.0	0.5989	1.0	0.9166	0.9883	0.5415	0.7509	0.0089	0.9688
KSVM	0.9671	1.0	0.037	0.996	0.897	1.0	0.6227	0.8992	0.0089	0.9852

Table 4: Accuracy for all 11 ML trained classifiers after tuning applied to a BIG training set and BIG+training set with added samples.

- The F-score of a class A is defined as $\frac{2 \cdot \text{precision}_A \cdot \text{recall}_A}{\text{precision}_A + \text{recall}_A}$.

The subscript p denotes the packed class, the subscript np denotes non-packed class, and the subscript wa denotes the weighted average.

Overall the precision, recall, and F-score tend to be higher for the non-packed class for almost all ML classifiers. The only exception is small improvement for DL8.5 regarding precision. Globally of the 11 ML trained classifiers KSVM scores the best in all metrics except for precision$_{np}$ and recall$_p$ which are both highest with KNN.

Overall the Bayes-based classifiers (GNBC & BNBC) perform relatively poorly compared to the other ML trained classifiers. The significant weakness of these classifiers appears to be in their ability to detect packed samples, with all metrics much lower than non-packed metrics. GNBC performs worse than BNBC, which is in line with the prior tuning results.

The linear models (LR & LSVM) performed in the middle compared with the other algorithms, and very similarly to each other. There is no easy way to distinguish LR from LSVM fro these metrics, and so a decision between them may consider the training time where LR is much faster.

The decision tree based classifiers (DT, GBDT, RF, & DL8.5) mostly perform well, with GBDT performing slightly better than RF while DT and DL8.5 are less effective on all metrics.

MLP appears effective and comparable with the best decision tree based classifier (GBDT) although worse than KNN and LSVM.

Finally, KSVM appears to offer the best overall performance among the different classifiers.

5.2. Adaptability

In the prior experiments the data sets consist of collected samples labeled with majority vote (as detailed in Section 3.1). To explore issues related to poorly represented packers and adaptability of the ML algorithms, the accuracy contrasting BIG with BIG+$_f$ is here evaluated with a variety of packers f. This contrast is detailed for five packer families: UPX, kkrunchy, MPress, TElock and PEtite. In each case all 11 ML trained classifiers are trained on both BIG and BIG+$_f$ for each f of the five packers and evaluated on SELFPACK $_f$.

These five packers are chosen as they represent very different frequencies within BIG. The frequency of UPX is 11978 in BIG and 12478 in BIG+$_{UPX}$. The frequency of kkrunchy is 71 in BIG and 571 in BIG+$_{kkrunchy}$. The frequency of MPress is 344 in BIG and 844 in BIG+$_{MPress}$. The frequency of TElock is 351 in BIG and 851 in BIG+$_{TElock}$. Finally, the frequency of PEtite is 23 in

BIG and 523 in BIG+$_{PEtite}$. Note that the frequencies for all packers except f in BIG+$_f$ remains unchanged.

The experimental approach is an adaptation of the approach used in [9]. All the ML classifiers are trained using either BIG or BIG+$_f$ and then evaluated on SELFPACK $_f$. The goal of this experiment was to observe the adaptability of the ML trained classifiers to detect new samples created with known packer f in BIG+$_f$ and compare this with the ML classifier that is trained on BIG without the additional packed samples of family f. As usual all experiments were performed with 10-fold cross-validation, although here the training set and testing sets are distinct.

An overview of the results can be seen in Table 4. Observe that in almost all cases the extended training set improves the accuracy for classification on SELFPACK $_f$ for all f (the only exception is TElock using MLP). Observe that not all classifiers are equal against new packed samples and many perform poorly on unknown packers. Since KNN keeps and uses all the data set for its model, more new samples appear necessary to improve classification to a high accuracy. GNBC seems to perform great on samples sufficiently present in the data set and show difficulties facing other packers (even when extending the data set). Comparatively, BNBC performs better by directly recognizing more packers and being able to integrate new information for its model. LR and LSVM models both perform well and are able to integrate new samples to improve their model (particularly LSVM). DT and DL8.5 performs well for some packers (UPX and MPress) without additional samples but fail to integrate new packers efficiently even when extending the training set. Contrarily, GBDT adapts extremely well and RF also improves, albeit less than GBDT. DL8.5 shows improvement with new information, although like RF struggling to handle PEtite. MLP shows excellent results by detecting many packers with high accuracy and integrating new information effectively. Finally, KSVM already performed extremely like MLP and integrated new information even more effectively.

These results address **RQ2** by examining the effectiveness of all 11 ML algorithms via various metrics and experiments. There are 4 ML algorithms that appear to be most effective as briefly highlighted below.

- KNN is memory expensive (all data points are part of the model) and requires many samples to improve its model. However, KNN avoids false positives/negatives and performs well if samples are sufficiently present in the training data set.
- GBDT is light in training (compared to other methods). GBDT offers good performance and can easily extend its model in case of a new packers detected (if samples are provided). (Note RF is slightly less efficient but in many ways comparable to GBDT if a decision trees based classifier is desirable.)
- MLP is resource demanding for training but offers good performance and adaptability.
- KSVM appears to be the most effective and adaptable ML algorithm for packing detection. KSVM has a low false positive/negative rate and all packers are detected in the adaptability experiments.

6. Economical Analysis

This section addresses **RQ3** (Which ML algorithms perform well in long term for packing detection?) by exploring the effectiveness of all 11 ML algorithms over chronological data.

Classifier	Baseline	Period 1	Period 2	Period 3	Period 4
KNN	0.9902	0.9899	0.9830	0.9823	0.9696
GNBC	0.9046	0.9034	0.8940	0.8738	0.8751
BNBC	0.9263	0.9170	0.9158	0.8890	0.8545
LR	0.9703	0.9634	0.9506	0.9452	0.9402
LSVM	0.9703	0.9627	0.9509	0.9466	0.9396
DT	0.9580	0.9551	0.9438	0.9395	0.9324
RF	0.9819	0.9776	0.9751	0.9768	0.9685
GBDT	0.9880	0.9851	0.9806	0.9824	0.9701
DL8.5	0.9456	0.9383	0.9408	0.9037	0.9016
MLP	0.9882	0.9823	0.9781	0.9757	0.9669
KSVM	0.9929	0.9924	0.9922	0.9913	0.9788

Table 5: Economical analysis for all 11 ML trained classifiers after tuning trained on BIG and tested on BIG (baseline) and CHRONO split by chronological period.

Classifier	Train time [s]	Uptime 0.92 [s]	Ratio 0.92	Uptime 0.95 [s]	Ratio 0.95	Uptime 0.97 [s]	Ratio 0.97
KNN	2.1259	8,986,447	4,227,126	6,846,280	3,220,415	4,907,050	2,308,222
GNBC	0.0368	N/A	N/A	N/A	N/A	N/A	N/A
BNBC	0.1673	1,661,053	9,928,592	N/A	N/A	N/A	N/A
LR	11.0572	8,346,239	754,824	2,791,949	252,500	134,757	12,187
LSVM	316.172	9,961,820	31,507	2,865,875	9,064	90,561	286
DT	0.0607	6,874,338	113,251,038	1,680,899	27,691,918	N/A	N/A
RF	0.7492	14,776,270	19,722,731	9,715,226	12,967,467	4,812,299	6,423,250
GBDT	2.7977	10,303,293	3,682,772	7,616,469	2,722,403	5,116,627	1,828,869
DL8.5	599.31	3,310,904	5,524	N/A	N/A	N/A	N/A
MLP	317.29	12,244,014	38,589	8,004,772	25,228	4,397,608	13,859
KSVM	161.9987	9,262,621	57,177	7,473,608	46,133	5,900,674	36,424

Table 6: Economical analysis for all 11 ML trained classifiers after tuning trained on BIG including time to train, uptime while maintaining F-scores, and uptime to train time ratio for F-scores 0.92, 0.95, and 0.97.

This section evaluates the effectiveness of all 11 ML algorithms over time and the retraining required to maintain a high level of accuracy. To perform this analysis all 11 ML algorithms are trained on BIG and their effectiveness evaluated on CHRONO. BIG is used to train each classifier, thus training on samples collected from 2019-10-01 to 2020-02-28. The testing is then done on CHRONO which corresponds to malware samples obtained from 2020-04-01 to 2020-05-26. CHRONO has been divided into four subsets corresponding to separate periods of time as follows. Period 1 designates malware obtained from 2020-04-01 to 2020-04-14. Period 2 malware samples are obtained from 2020-04-15 to 2020-04-28. Period 3 malware samples are obtained from 2020-04-29 to 2020-05-12. Finally, period 4 malware samples are obtained from 2020-05-13 to 2020-05-26

An overview of the accuracy and efficiency of all 11 ML trained classifiers on these four periods from CHRONO are shown in Table 5. As expected, the effectiveness of all 11 ML trained classifiers decreases with time. This corresponds to prior research [18] and can be explained by different trends in packers used depending on the period of interest. Some packers are employed frequently by malware in the period 1 of CHRONO and are slowly replaced by other packers during later periods.

The training time and uptime can be used to predict the economics of an ML trained classifier

by calculating the retraining required to maintain a given quality. Here the quality is picked as a target F-score and then the uptime is the time after training during which the ML trained classifier has at least the chosen quality. An overview of the quality measures for three different F-scores (0.92, 0.95, and 0.97) are shown in Table 6. Using the period calculated F-scores for each period a curve for more precise estimates is obtained by a quadratic least squares regression. This curve can then be used to predict the time that a particular trained classifier will drop below a given F-score threshold. The ratio then indicates the economics of cost in training to uptime, with higher numbers being better, Here these indicate that DT and RF have the best economics due to their extremely quick training time and generally high accuracy and F-score. KNN and GBDT also perform well, although the significant increase in training time brings their economic ratio down significantly. LR, LSVM, MLP, and KSVM are all also capable of meeting the requirements, but are economically hampered by their significantly larger training times. However, observe that since the uptime for KNN, RF, GBDT, MLP, and KSVM all have uptimes $>\sim 80$ days even the ~ 5 minutes training time for the highest (MLP) is not too significant.

In machine learning, the training data set used is critical to ensure good performance (as noted also in Section 5.2). Although the performance of all classifiers decreases over time, some classifiers are more robust than others and require less training time and frequency to maintain a given level of effectiveness. This allows us to answer **RQ3**: KNN, LR, DT, RF, GBDT, MLP and KSVM are all able to maintain an F-score of 0.95. From the pure economics, DT and RF significantly out perform the others, although considering all training times as reasonable then KNN, RF, GBDT, MLP, and KSVM are all able to remain effective for long periods of time between retraining.

7. Challenges to Validity

This section briefly discusses the main potential challenges to the validity of this work and a brief examination of the significance of dynamic features in packer detection.

One challenge to the validity of many works in this area is the selection of ground truth. The approach used here has the advantage of a broad selection of samples that are believed to be malware. Although this has the potential limitation of having few or no clean samples, this choice was made since this work explores the role of packing detection within a malware detection setting. Another challenge the ground truth used here is that this relies upon a multiple approaches that in turn are subject to their own limitations. Broadly the consensus approach used here should mitigate some of the limitations (see also [18]). However, this leaves those approaches and this work relatively vulnerable to rare or unknown packers. To some extent the adaptability exploration (see Section 5.2) addresses this, by showing the effectiveness of the ML algorithms when presented with new training data. The above aside, clearly better ground truth and training data should improve the behaviour of all the ML algorithms considered here.

Another challenge is the use of only accuracy as a metric for the tuning and most analysis performed in this work. This choice was to simplify the work by having a single metric (that in turn accounts for both positive and negative outcomes) without giving too much bias or favour to any particular outcome. The exploration of different metrics (see Section 5.1) indicates that other metrics do not vary significantly from accuracy. Thus, the results here should be easily broadly applicable to other metrics, and the methodology easily adaptable should another metric be more important in another application.

A further challenge is the use of only static features for packing detection although some related works have also considered dynamic features [15, 21, 25]. The problem of packing detection as part of

Classifier	Static Features Only	Static & Dynamic Features
KNN	0.9889	0.9889
GNBC	0.939	0.943
BNBC	0.947	0.947
LR	0.9836	0.9836
LSVM	0.9833	0.9833
DT	0.9864	0.9864
RF	0.9888	0.9888
GBDT	0.9941	0.9941
DL8.5	0.9906	0.9907
MLP	0.9902	0.9915
KSVM	0.9938	0.9931

Table 7: Accuracy comparison for all 11 ML algorithms using CHRONO static features and with additional dynamic features.

a larger malware detection and defence system motivated the choice to focus only on static features in this work, as well as this being proven highly effective both here and in related works. However, a small investigation was done to explore whether dynamic features may have been significantly useful. This is included below, although due to the cost of feature extraction for dynamic features this work chose to focus on larger data sets and extensive results on cheap features to extract.

7.1. A Note on Dynamic Features

Since dynamic features have been used in the literature, and since packing could be more obvious when a program is executed, this raises a potential further research questions. **RQA Do dynamic features improve packing detection?** This section briefly presents results exploring this question.

Dynamic entropy features are often used in the literature to detect packing [15, 17, 24]. To explore whether dynamic features were significant, we considered four dynamic entropy features. The maximum increase of entropy of a section (Feature D1). The maximum decrease of entropy of a section (Feature D2). The biggest change of entropy of the whole file (Feature D3). The change of entropy of the entrypoint section (Feature D4). Since initial entropy of code section, data section, resource section, entire PE file, and entrypoint sections are already present in the static features, adding these novel features constitute an approximation of the entropy pattern generally studied in literature.

To extract these features each sample was executed once for each of their executable sections. The execution continues until control flow reached the first jump instruction into the target executable section. This stop condition reached, entropy values of all distinct sections and global entropy of the file are recorded and the process is reiterated for the subsequent executable section. This way of proceeding is less powerful than monitoring all entropy changes and stopping accordingly (e.g. a packer using multiple layers could still be engaged in the unpacking process). However, it could be applied easily to non-packed executables required in the context of packing detection.

The dynamic features were extracted for a subset of 12,036 samples from CHRONO and then all 11 ML algorithms were trained and tested using 10-fold cross-validation with 90% of the data set used for training and 10% for validation (as in all the previous experiments).

The impact of these features on accuracy results can be observed in Table 7. These results indicate that (our) dynamic features do not significantly influence the decision process. At best there is a negligible increase in accuracy (e.g. for GNBC, DL8.5, and MLP) and at worse a degradation (KSVM), although most did not benefit from the addition of dynamic features.

Due to the significant extraction cost (in time, instrumentation, execution environment, etc.) of dynamic features and the lack of impact, these results show that static features can be preferred for efficient packer classification as they already provide high accuracy for packing detection. Note that since these results indicated a negligible improvement on a restricted data set. Thus, further exploration of these features was not considered here. This result agrees with what has been reported in literature, in that dynamic features can contribute to accurate detection, but are not necessary. Particularly when handling only detection, and when the accuracy is already very high. Generally, when dynamic features are used in the literature, the objective is not just packing detection but also classification or direct unpacking by trying to find the original entrypoint. Thus the high costs of computing dynamic features need to be countered balance by stronger objectives than just malware detection. However, since IAT table features show significant influence on classification, focusing on these kinds of dynamic features could deliver more impact (similar to [21]). Even though overhead would generally stay important. We leave this question for future works.

8. Conclusion

The challenge of detecting packed programs is a key part of contemporary cyber-security defense. Many approaches have been taken to detect whether a sample is packed, applying various algorithms to many different features and measuring the effectiveness in different ways. This work addresses three main research questions about the: significant features, effective ML algorithms, and long term effectiveness for packing detection.

The feature significance shows that across 11 different ML algorithms there are a small number of static features that are most significant. These are related to: the stack reserve (Feature 19), number of readable and writable sections (Feature 29), non-standard entry point section (Feature 35), raw and virtual data size of sections (Features 37 & 39), byte entropy of .text sections (Feature 43), 12th entry point byte (Feature 60), and malicious API calls (Features 115-116). This diversity of features correlates with prior results showing that many different kinds of features can be effective, and that none is clearly dominant. Further, by gathering feature use data across many ML algorithms there is evidence that many features can play a small role in improving accuracy, even if they are not significant alone.

The effectiveness on 11 different ML algorithms was considered with various metrics and also considering permutations in the training data. Overall KNN, GBDT, MLP, and KSVM proved to be the most effective and robust. These 4 algorithms were all able to score well on: accuracy, precision, recall, and F-score taking into account packed unpacked, and averaging. There 4 algorithms also performed well in adapting to new training data effectively.

The long term effectiveness of the 11 ML algorithms was also evaluated using chronological samples. The results here showed that all classifiers had reduced accuracy over time (as expected). The uptime between retraining for the different algorithms varied substantially, with KNN, RF, GBDT, MLP, and KSVM all being able to maintain a high efficiency for $\sim$ 80 days or more. However, the extremely low training cost for DT and RF makes them also competitive if time to retrain is weighted significantly higher than time between retraining.

Overall the results here indicate that KNN, GBDT, MLP, and KSVM are all effective and economical. Of these 4 the choice for a particular application depends on which metrics are most important, with KNN and GBDT scoring higher in economic ratio due to lower training time and needing less features, but tending to be lower in effectiveness. By contrast KSVM scores higher in effectiveness metrics at the cost of using more features and taking more training time. MLP does not stand out significantly in any one area, but has a different balance that may be optimal for some applications.

Acknowledgments. Charles-Henry Bertrand Van Ouytsel is FRIA grantee of the Belgian Fund for Scientific Research (FNRS-F.R.S.). We would like to thanks Cisco for their malware feed and the availability of their packing detector engine.

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] Generalized linear model (GLM) — h2o 3.30.0.4 documentation. `https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/glm.html`.

[4] Malware statistics trends report: Av-test. `https://portal.av-atlas.org/malware`. 14 January 2021.

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

[6] pydl8.5. `https://github.com/aia-uclouvain/pydl8.5`. May 2020.

[7] The shadowserver foundation. `https://www.shadowserver.org/`. November 2019.

[8] The wildlist organization international. `http://www.wildlist.org/`.

[9] 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.

[10] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, LA, USA, March 9-11, 2016, pages 183–194, 2016.

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

[12] 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.

[13] U. Baldangombo, N. Jambaljav, and S.-J. Horng. A static malware detection system using data mining methods.

[14] R. Balestriero, H. Glotin, and R. G. Baraniuk. Semi-supervised learning enabled by multiscale deep neural network inversion. CoRR, abs/1802.10172, 2018.

[15] 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.

[16] M. Bat-Erdene, T. Kim, H. Park, and H. Lee. Packer detection for multi-layer executables using entropy analysis. Entropy, 19(3):125, 2017.

[17] M. Bat-Erdene, H. Park, H. Li, H. Lee, and M.-S. Choi. Entropy analysis to classify unknown packing algorithms for malware detection. International Journal of Information Security, 16(3):227–248, 2017.

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

[19] F. Biondi, T. Given-Wilson, A. Legay, C. Puodzius, and J. Quilbeuf. Tutorial: An overview of malware detection and evasion techniques. In T. Margaria and B. Steffen, editors, Leveraging Applications of Formal Methods, Verification and Validation. Modeling, volume 11244, pages 565–586. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[20] G. Bonfante, J. M. Fernandez, J. Marion, B. Rouxel, F. Sabatier, and A. Thierry. Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015, pages 745–756. ACM, 2015.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] A. C. Müller and S. Guido. Introduction to machine learning with Python: a guide for data scientists. O'Reilly.

[29] S. Nijssen, P. Schaus, et al. Learning optimal decision trees using caching branch-and-bound search. In Thirty-Fourth AAAI Conference on Artificial Intelligence, 2020.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] X. Ugarte-Pedrero, I. Santos, and P. G. Bringas. Structural feature based anomaly detection for packed executable identification. In Computational intelligence in security for information systems, pages 230–237. Springer, 2011.

[38] 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.

[39] 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.

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

[41] 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. Extracted Features

This section includes a full table (A.8) of all the 119 static features considered in this work and their descriptions. They are also grouped according to their commonality as in Biondi et al. [18].

ID	Type	Description
		Metadata features
1	*Boolean*	DLL can be relocated at load time. *Extracted from DLLs characteristics*
2	*Boolean*	Code Integrity checks are enforced. *Extracted from DLLs characteristics*
3	*Boolean*	Image is NX compatible. *Extracted from DLLs characteristics*
4	*Boolean*	Isolation aware, but do not isolate the image. *Extracted from DLLs characteristics*
5	*Boolean*	Does not use structured exception (SE) handling. No SE handler may be called in this image. *Extracted from DLLs characteristics*
6	*Boolean*	Do not bind the image. *Extracted from DLLs characteristics*
7	*Boolean*	A WDM driver. *Extracted from DLLs characteristics*
8	*Boolean*	Terminal Server aware. *Extracted from DLLs characteristics*
9	*Integer*	The image file checksum.
10	*Integer*	The preferred address of the first byte of image when loaded into memory
11	*Integer*	The address that is relative to the image base of the beginning-of-code section when it is loaded into memory.
12	*Integer*	The major version number of the required operating system.
13	*Integer*	The minor version number of the required operating system.
14	*Integer*	The size (in bytes) of the image, including all headers, as the image is loaded in memory.
15	*Integer*	The size of the code (.text) section, or the sum of all code sections if there are multiple sections.
16	*Integer*	The combined size of an MS DOS stub, PE header, and section headers rounded up to a multiple of FileAlignment.
17	*Integer*	The size of the initialized data section, or the sum of all such sections if there are multiple data sections.
18	*Integer*	The size of the uninitialized data section (BSS), or the sum of all such sections if there are multiple BSS sections
19	*Integer*	The size of the stack to reserve.
20	*Integer*	The size of the stack to commit.
21	*Integer*	The alignment (in bytes) of sections when they are loaded into memory.
		Section features
22	*Integer*	The number of standard sections the PE holds
23	*Integer*	The number of non-standard sections the PE holds
24	*Float*	The ratio between the number of standard sections found and the number of all sections found in the PE under analysis
25	*Integer*	The number of Executable sections the PE holds
26	*Integer*	The number of Writable sections the PE holds
27	*Integer*	The number of Writable and Executable sections the PE holds
28	*Integer*	The number of readable and executable sections

29	*Integer*	The number of readable and writable sections
30	*Integer*	The number of Writable and Readable and Executable sections the PE holds
31	*Boolean*	The code section is not executable
32	*Boolean*	The executable section is not a code section
33	*Boolean*	The code section is not present in the PE under analysis
34	*Boolean*	The entry point is not in the code section
35	*Boolean*	The entry point is not in a standard section
36	*Boolean*	The entry point is not in an executable section
37	*Float*	The ratio between raw data and virtual size for the section of the entry point
38	*Integer*	The number of section having their raw data size zero
39	*Integer*	The number of sections having their virtual size greater than their raw data size.
40	*Float*	The maximum ratio raw data to virtual size among all sections
41	*Float*	the minimum ratio raw data to virtual size among all sections
42	*Boolean*	The address pointing to raw data on disk is not conforming with the file alignment
Entropy features		
43	*Float* $\in [0; 8]$	The byte entropy of code (.text) sections
44	*Float* $\in [0; 8]$	The byte entropy of data section
45	*Float* $\in [0; 8]$	The byte entropy of resource section
46	*Float* $\in [0; 8]$	The byte entropy of PE header
47	*Float* $\in [0; 8]$	The byte entropy of the entire PE file
48	*Float* $\in [0; 8]$	The byte entropy of the section holding the entry point of the PE under analysis
Entry byte features		
49 - 112	*Integer* $\in [0; 255]$	Values of 64 bytes following the entry point, each byte correspond to 1 feature position
Import functions features		
113	*Integer*	The number of DLLs imported
114	*Integer*	The number of functions imported found in the import table directory (IDT)
115	*Integer*	The number of malicious APIs imported (malicious as defined in [41]). This list includes: "GetProcAddress", "LoadLibraryA", "LoadLibrary", "ExitProcess", "GetModuleHandleA", "VirtualAlloc", "VirtualFree", "GetModuleFileNameA", "CreateFileA", "RegQueryValueExA", "MessageBoxA", "GetCommandLineA", "VirtualProtect", "GetStartupInfoA", "GetStdHandle", "RegOpenKeyExA".
116	*Float*	The ratio between the number of malicious APIs imported to the number of all functions imported by the PE
117	*Integer*	the number of addresses (corresponds to functions) found in the import address table (IAT)
Ressource features		
118	*Boolean*	The debug directory is present or not
119	*Integer*	The number of resources the PE holds

Table A.8: Features description.

Appendix B. Boolean Conversion Applied to Features

This section details in Table B.9 how each of the 119 features (detailed in Table A.8) is converted into Booleans via bucketing.

ID	Description
1-8	Already boolean
9	2 buckets : 0 or other values
10	2 buckets : 4194304 or other values
11	3 buckets : 4096, 8192 or other values
12	3 buckets : 4, 5 or other values
13	2 buckets : 0 or other values
14	2 buckets : < 250000 or $>= 25000$
15	2 buckets : < 50000 or $>= 50000$
16	5 buckets : 1024, 4096, 512, 1536 or other values
17	feature deleted due to high sparsity
18	2 buckets : 0 or other values
19	2 buckets : 1048576 or other values
20	5 buckets : 4096, 16384, 8192, 65536 or other values
21	3 buckets : 4096, 8192 or other values
22	10 buckets : 0, 1, 2, 3, 4, 5, 6, 7, 8 or $>= 9$
23	5 buckets : 0, 1, 2, 3 or $>= 4$
24	2 buckets : 0 or other values
25	2 buckets : < 3 or $>= 3$
26	2 buckets : 0 or other values
23	5 buckets : 0, 1, 2, 3 or $>= 4$
24	2 buckets : 0 or other values
25	2 buckets : < 3 or $>= 3$
26	6 buckets : 0, 1, 2, 3, 4 or $>= 5$
27	4 buckets : 0, 1, 2 or $>= 3$
28	4 buckets : 0, 1, 2 or $>= 3$
29	6 buckets : 0, 1, 2, 3, 4 or $>= 5$
30	4 buckets : 0, 1, 2 or $>= 3$
31-36	Already boolean
37	3 buckets : > 1, < 1 or 1
38	4 buckets : 0, 1, 2 or $>= 3$
39	5 buckets : 0, 1, 2, 3 or $>= 4$
40	3 buckets : > 1, < 1 or 1
41	2 buckets : 0 or other values
42	Already boolean
43-45	2 buckets : -1 or other values
46-117	2 buckets : 0 or other values
118	Already boolean
119	2 buckets : 0 or other values

Table B.9: Explanation of Boolean conversion with bucketing for all 119 features.