Collective Classification for Packed Executable Identification

Xabier Ugarte-Pedrero, Igor Santos, Carlos Laorden, Borja Sanz, and Pablo G. Bringas

{xabier.ugarte, isantos, claorden, borja.sanz, pablo.garcia.bringas}@deusto.es

S^{3}Lab
DeustoTech - Computing
Deusto Institute of Technology
University of Deusto
Avenida de las Universidades 24, 48007
Bilbao, Spain

April 23, 2012

Abstract

Malware writers employ packing techniques (i.e., encrypt the real payload) to hide the actual code of their creations. Generic unpacking techniques execute the binary within an isolated environment (namely 'sandbox') to gather the real code of the packed executable. However, this approach can be very time consuming. A common approach is to apply a filtering step to avoid the execution of not packed binaries. To this end, supervised machine learning models trained with static features from the executables have been proposed. Notwithstanding, these methods need the identification and labelling of a high number of packed and not packed executables. In this paper, we propose a new method for packed executable detection that adopts collective learning approaches (a kind of semi-supervised learning) to reduce the labelling requirements of completely supervised approaches. We performed an empirical validation demonstrating that the system maintains a high accuracy rate when the number of labelled instances in the dataset is lower.

Keywords: malware, machine-learning, collective-learning, packer detection.

1 Introduction

Malware is defined as any computer software that damages computers or networks [1, 2]. Since the antivirus systems can only assure the detection of currently known threats, malware creators employ obfuscation techniques that hide the actual code of the malware. Among these techniques, one that is commonly used is executable packing. This method encrypts or compresses the actual malicious code. Packed software has a decryption routine inside its code that deciphers the malicious code which remains in a data section of the memory. According to a recent study [3], up to the 80% of the detected malware is packed.

One simple approach commonly used for packer detection is signature scanning. For instance, PEID [4] can detect a wide range of well-known packers, while Faster Universal Unpacker (FUU) [5] identifies the packer and applies custom unpacking routines designed and specifically coded for each different packer. However, signature-based detection has the same shortcomings as for malware detection: it is not effective for unknown obfuscation techniques or custom packers (i.e., protectors implemented for certain malware samples). This limitation constitutes a significant problem for malware analysis: according to Morgenstern and Pilz [6], the 35 % of malware is packed by a custom packer.

These techniques introduce the necessity of an unpacking step in the malware analysis problem. This unpacking step can be either specific or generic. Specific approaches apply unpacking routines specifically coded for certain versions of a packer. This approach is efficient and allows malware

analysts to unpack a high ammount of samples protected with common packers. Nevertheless, specific unpacking routines are useless for unknown packers. Considering the ammount of custom packers and new modified versions of existing ones, it is infeasible to create new routines for each new packing scheme: generic unpacking methods are necessary. The main drawback of generic approaches is their cost in terms of computing resources and time, that makes them infeasible to process every single sample created every day. As a consequence, sample classification and filtering becomes a very useful step to leverage the ammount of samples processed with generic and time-consuming unpacking tools. In this way, a classification system able to differentiate between packed and not packed samples can avoid the execution of not packed samples in such systems.

For years, machine-learning techniques have been proved as effective methods to face classification problems. Supervised machine-learning techniques are commonly used to train models using a set of previously labelled data. However, these supervised machine-learning classifiers require a high number of labelled executables of each of the classes. It is quite difficult to obtain this amount of labelled data for a real-world problem such as malicious code analysis. To gather these data, a time-consuming process of analysis is mandatory and, in the process, some malicious executables are able to surpass detection.

Semi-supervised learning is a type of machine-learning technique specially useful when a limited amount of labelled data exists for each class [7]. More concretely, collective classification [8] is a semi-supervised learning approach that uses the relational structure of the combined labelled and unlabelled data-sets to enhance the classification accuracy. With these relational approaches, the predicted label of an example will often be influenced by the labels of related samples.

The idea underlying collective classification is that the predicted labels of a test sample should also be influenced by the predictions made for related test samples. Sometimes, we can determine the topic of not just a single evidence but to infer it for a collection of unlabelled evidences. Collective classification tries to collectively optimise the problem taking into account the connections present among the instances. In summary, collective classification is a semi-supervised technique, i.e., uses both labelled and unlabelled data — typically a small amount of labelled data and a large amount of unlabelled data —, that reduces the labelling work. In addition, these approaches have been successfully applied to different classification problems in the area of information security such as spam filtering [9] or malware detection [10, 11]

In light of this background, we propose here the first approach that employs collective classification techniques for the detection of packed executables based on structural features and heuristics. These methods are able to learn from both labelled and unlabelled data to build accurate classifiers. We propose the adoption of collective learning for the detection of packed executables using structural information of the executable. For training the classifiers, we employ the same structural and heuristic features used in our previous work [12, 13], which presented an anomaly detector for packed executable filtering.

Summarising, our main contributions in this paper are:

- We describe how to adopt collective classification for packed executable detection.
- We empirically determine the optimal number of labelled instances and we evaluate how this parameter affects the accuracy of the model.
- We compare the results obtained for collective classification with previous semi-supervised [14], and supervised approaches.
- We demonstrate that labelling efforts can be reduced in the malware detection industry, while still maintaining a high accuracy rate.

The remainder of this paper is organised as follows. Section 2 describes related work on malware unpacking and packed software detection. Section 3 describes the structural features employed for packed executable identification. Section 4 describes previous approaches used for packed executable filtering. Section 5 describes different collective classification methods and how they can be adopted for packed executable detection. Section 6 describes the experiments and presents results. Finally, Section 7 concludes the paper and outlines avenues for future work.

2 Related Work

Dynamic unpacking approaches monitor the execution of a binary within an isolated environment to retrieve its actual code. This isolated environment can be deployed as a virtual machine or as

an emulator [15]. Then, the execution is traced and stopped when certain events occur.

Various dynamic unpackers (e.g., Universal PE Unpacker [16] and OllyBonE [17]) are based on heuristics to determine the exact point where the execution flow jumps from the unpacking routine to the original code and, once reached, they retrieve the memory content to obtain an unpacked version of the malicious code. Nevertheless, concrete heuristics are not applicable to every packer because not all of them work in the same way. For example, some packers do not unpack the code before executing it [18]: original code is transformed and stored as a custom language and, then, translated to actual code at runtime by an interpreter. Thereby, the malicious code will never be present at the same time in memory.

In contrast, other proposed techniques are not so heuristic dependant. PolyUnpack [19] obtains an static representation of the code and then compares it to the code dynamically generated during its execution. If both codes differ, the executable is considered to be packed. Renovo [20] monitors memory accesses and checks whether any memory area has been written and then executed or not. OmniUnpack [21] statically analyses memory pages that have been over-written and then executed only when a dangerous system call is performed. Finally, Eureka [22] determines the moment when an executable reaches an stable state so it can be statically analysed. However, these methods are very time consuming and cannot counter the conditional execution of unpacking routines, a technique used for anti-debugging and anti-monitoring defense [23, 24, 25].

By using the structural information of PE executables, some methods (e.g., PE-Miner [26], PE-Probe [27] and the method proposed by Perdisci et al. [28]) can determine if the sample under analysis is packed or if it is suspicious of containing malicious code, in order to act as a pre-filter to these time consuming generic unpacking techniques.

3 Structural Features of the PE Files

Given the conclusions obtained in previous work [26, 27, 28], we selected a set of 209 structural characteristics from the PE executables. Some of the characteristics were obtained directly from the PE file header, while the rest were calculated values based on certain heuristics commonly used by the scientific community. Given the success of the heuristics proposed by Perdisci et al. [28], we included them in our feature set.

We consider that one of the main requisites of our detection system has to be the speed, as it constitutes a filtering step for a heavier unpacking process. For this reason, we selected a set of characteristics whose extraction does not require large processing efforts, and avoided techniques such as code disassembly, string extraction or n-gram analysis [28, 29], which could slow down sample comparison.

Particularly, in this paper, we use an improved dataset adding a total of 2,000 binaries to the one used in previous work [30]. In this new dataset, some executables contained headers for 64-bit machines. To extract as much data as possible, these header fields where represented as 29 extra structural characteristics.

Features can be divided into four main groups: 154 raw header characteristics, 31 section characteristics (i.e., number of sections that meet certain properties), 29 characteristics of the section containing the entry point (i.e., the section which will be executed first once the executable is loaded into memory) and, finally, 24 entropy values (i.e., byte randomness in the file).

Furthermore, we have measured each characteristic relevance based on Information Gain (IG) [31] which is defined as the change in information entropy from the state of the original dataset to the state of the dataset when a certain attribute is given:

$$IG(\mathcal{T}, a) = H(\mathcal{T}) - H(\mathcal{T}|a)$$

where $\mathcal{T}$ is the training dataset, a is the attribute for which the IG is being calculated and $H(x)$ is the information entropy of a dataset. Practically, IG measures the variation in entropy when an attribute in the dataset is given. It is calculated as the difference between the entropy of the original dataset and the average entropy of the dataset subsets formed by the instances having the attribute a set to every value in $\mathcal{V}_a$ (i.e., the set of possible values for the attribute a).

IG provides a ratio for each characteristic that measures its importance to consider if a sample is packed or not. This ratio is commonly used in the classical C4.5 algorithm for decision tree generation. These values were calculated from a dataset composed of 2,000 packed executables (from which 1,000 belong to the Zeus malware family and the other 1,000 are executables packed with 10 different packers available on-line), and 2,000 not packed executables. The set of binaries

Table 1: Most relevant features extracted from the executable files. IG value measures the relevance of each feature to classify samples between packed and not packed.

Feature	**Information Gain**
File header block	
Time stamp	0.19974
Number of sections	0.14096
Optional header block	
Import Address Table (IAT) size	0.27292
Import Address Table (IAT) address	0.26561
Import Table (.idata section) size	0.24567
Size of code	0.22522
Major linker version	0.20092
Import Table (.idata section) address	0.19990
Size of image	0.19349
Base of data	0.18165
Size of initialized data	0.17564
Dll characteristics	0.15255
Address of entry point	0.15015
Checksum	0.14860
Section characteristics	
Minimum raw data per virtual size ratio [1]	0.27505
Maximum raw data per virtual size ratio	0.22478
Number of non standard sections	0.20581
Number of executable sections	0.20422
Virtual size of all the sections	0.19788
Mean virtual size of the sections	0.18014
Number of readable and executable sections	0.17102
Number of sections with virtual size greater than raw data	0.16090
Raw data size of all sections	0.16033
Number of readable, writeable and executable sections	0.14904
Mean raw size of all the sections	0.14564
Number of readable sections	0.14528
Section of entry point characteristics	
Virtual size	0.32969
Raw data per virtual size ratio	0.30717
Pointer to raw data	0.25724
Raw characteristics field	0.23985
Virtual address	0.23137
Size of raw data	0.20986
Writeable section	0.20616
Entropy values	
Global file entropy	0.57561
Maximum entropy	0.56056
Mean section entropy	0.52168
Mean code section entropy	0.41589
Section of entry point entropy	0.39560
Mean data section entropy	0.38755
The number of sections with entropy in the range between 7.5 and 8	0.29999
Header entropy	0.16333

manually packed was composed of 500 malware samples and 500 benign samples. Similarly, the set of not packed samples was composed of 1,000 malicious and 1,000 benign binaries. We reduced the amount of selected characteristics to obtain a more efficient classification, given that 38 of them have a zero IG value. Table 1 shows the most relevant features extracted and their IG values for the dataset described.

3.1 DOS header characteristics

The first bytes of the PE file header correspond to the 31 DOS executable fields. IG results showed that these characteristics are not specially relevant, having a maximum IG value of 0.10, corresponding to a reserved field. 14 values range from 0.05 to 0.10, and the rest present a relevance bellow 0.05.

3.2 File header block

This header block is present in both image files (i.e., ".exe"), and object files (i.e., ".dll"). From a total of 23 characteristics, 18 have an IG value greater than 0, and only 2 have an IG value greater than 0.10.

3.3 Optional Header Block

The 100 features of this optional block, which is only part of image files, contain data about how the executable must be loaded into memory. The 29 fields corresponding to 64-bit binary header fields are included in this group. 80 characteristics have an IG value over 0, and 12 have an IG value over 0.10.

3.4 Section characteristics

From the 31 characteristics that conform this group, 22 have an IG value greater than 0 and 12 have an IG value over 0.10.

3.5 Section of entry point characteristics

This group contains 29 characteristics relative to the section which will be executed once the executable is loaded into memory. 26 characteristics have an IG value greater than 0, from which 7 have a significant relevance (> 0.10).

3.6 Entropy values

We have selected 24 entropy values (a measure employed in previous work [32]), which have an IG value greater than 0. Concretely, 9 have a relevant IG value (> 0.10).

4 Previous Approaches

Packed executable detection is a problem that has been faced previously. Different machine learning approaches have been applied to perform this task.

Machine learning is a branch of Artificial Intelligence (AI) that is focused on the design and development of algorithms that allow computers to learn and make predictions from empirical data (i.e., computer learning) [33]. Machine Learning is commonly used for classification problems.

Machine-learning algorithms can be classified into three different subsets: supervised learning, unsupervised learning and semi-supervised learning. On the one hand, supervised algorithms need the training dataset to be labelled [34]. On the other hand, unsupervised learning approaches do no need the data to be labelled [35]: these algorithms try to determine the way data is organised into groups or *clusters*. Finally, a third type of machine-learning algorithms enables to train models when the data is partially labelled (i.e., using both labelled and unlabelled instances). These algorithms can improve the accuracy of unsupervised methods [7].

Several approaches have applied supervised machine learning algorithms to packed executable classification. In addition, we previously proposed the application of the LLGC semi-supervised

learning algorithm for the same task [14]. In the remainder of this section we describe some of the algorithms employed previously.

4.1 Bayesian Networks

Bayesian networks [36], which are based on Bayes' Theorem [37], are defined as graphical probabilistic models for multivariate analysis. Specifically, they are directed acyclic graphs that have associated probability distribution functions [38]. The nodes of the directed graph represent problem variables (either premises or conclusions), and the edges represent conditional dependencies between such variables. Moreover, the probability function illustrates the strengths of these relationships in the graph [38]. The most important capability of Bayesian networks is their ability to determine the probability that a certain hypothesis is true (e.g., the probability that a binary is packed or not) given a historical dataset.

4.2 Decision Trees

Decision tree classifiers are a type of machine-learning classifier usually graphically represented as a tree. The internal nodes represent conditions on the variables of a problem, whereas the final nodes or leaves represent the possible ultimate decisions of the algorithm [39]. Different training methods are typically used for learning the graph structure of these models from a labelled dataset. We use Random Forest, an ensemble (i.e., combination of weak classifiers) of different randomly built decision trees [40], J48, the WEKA [41] implementation of the C4.5 algorithm [39]), and Bagged J48. Bagging is a technique used to aggregate multiple versions of classifiers trained with bootstrap replicates of the original dataset in order to reduce the variance when changes in the learning set produce significant changes on the classifiers constructed[42].

4.3 K-Nearest Neighbour

The K-Nearest Neighbour (KNN) [43] classifier is one of the simplest supervised machine learning models. This method classifies an unknown specimen based on the class of the instances closest to it in the training space by measuring the distance between the training instances and the unknown instance. Although several possible methods can be used to choose the class of the unknown sample, the most common technique is simply to classify the unknown instance as the most common class amongst the K-nearest neighbours.

4.4 Support Vector Machines(SVM)

SVM algorithms divide the n-dimensional space representation of the data into two regions using a hyperplane. This hyperplane always maximises the margin between the two regions or classes. The margin is defined by the longest distance between the examples of the two classes and is computed based on the distance between the closest instances of both classes to the margin, which are called supporting vectors [44]. Instead of using linear hyperplanes, many implementations of these algorithms use so-called kernel functions. These kernel functions lead to non-linear classication surfaces, such as polynomial, radial or sigmoid surfaces [45].

4.5 Neural Networks

Artificial Neural Networks are mathematical models inspired by biological neural networks. [46]. Formally, a neural network is defined as a sorted triple (N, V, w) with two sets N, V and a function w, where N is the set of *neurons* and V a set $\{(i, j|i, j \in \mathbb{N})\}$ whose elements are called *connections* between neuron i and neuron j. The function $w : V \to \mathbb{R}$ defines the *weights* [46]. In our case, we focus on the most widely used neural networks: feedforward multilayer networks, also known as multilayer perceptrons (MLP's) [47].

4.6 A Semi-supervised Algorithm: Learning with Local and Global Consistency (LLGC)

(LLGC) [48] is a semi-supervised algorithm that provides *smooth* classification with respect to the intrinsic structure revealed by known labelled and unlabelled points. It is based in the next

assumptions: (i) nearby points are likely to have the same label and (ii) points on the same structure are likely to have the same label [48].

5 Collective Classification for Packed Executable Detection

Collective classification is a combinatorial optimisation problem, in which we are given a set of executables, or nodes, $\mathcal{E} = \{e_1, ..., e_n\}$ and a neighbourhood function N, where $N_i \subseteq \mathcal{E} \setminus \{\mathcal{E}_i\}$, which describes the underlying network structure [49]. Being $\mathcal{E}$ a random collection of executables, it is divided into two sets $\mathcal{X}$ and $\mathcal{Y}$, where $\mathcal{X}$ corresponds to the executables for which we know the correct values and $\mathcal{Y}$ are the executables whose values need to be determined. Therefore, the task is to label the nodes $\mathcal{Y}_i \in \mathcal{Y}$ with one of a small number of labels, $\mathcal{L} = \{l_1, ..., l_q\}$.

In the remainder of this section we review the collective algorithms used in the empirical evaluation.

5.1 CollectiveIBK

This model uses internally WEKA's classic IBK algorithm, an implementation of the *K-Nearest Neighbour* (KNN), to determine the best k instances on the training set and builds then, for all instances from the test set, a neighbourhood consisting of k instances from the pool of train and test set (either a naïve search over the complete set of instances or a k-dimensional tree is used to determine neighbours). All neighbours in such a neighbourhood are sorted according to their distance to the test instance they belong to. The neighbourhoods are sorted according to their 'rank', where 'rank' means the different occurrences of the two classes in the neighbourhood.

For every unlabelled test instance with the highest rank, the class label is determined by majority vote or, in case of a tie, by the first class. This is performed until no further test instances remain unlabelled. The classification terminates by returning the class label of the instance that is about to be classified.

5.2 CollectiveForest

It uses WEKA's implementation of RandomTree as base classifier to divide the test set into folds containing the same number of elements. The first iteration trains the model using the original training set and generates the distribution for all the instances in the test set. The best instances are then added to the original training set (being the number of instances chosen the same as in a fold).

The next iterations train the model with the new training set and generate then the distributions for the remaining instances in the test set.

5.3 CollectiveWoods & CollectiveTree

CollectiveWoods works like CollectiveForest using CollectiveTree algorithm instead of RandomTree.

Collective tree is similar to WEKA's original RandomTree classifier. It splits the attribute at a position that divides the current subset of instances (training and test instances) into two halves. The process finishes if one of the following conditions is met:

- Only training instances are covered (the labels for these instances are already known).
- Only test instances in the leaf, case in which distribution from the parent node is taken.
- Only training instances of one class, case in which all test instances are considered to have this class.

To calculate the class distribution of a complete set or a subset, the weights are summed up according to the weights in the training set, and then normalised. The nominal attribute distribution corresponds to the normalised sum of weights for each distinct value and, for the numeric attribute, distribution of the binary split based on median is calculated and then the weights are summed up for the two bins and finally normalised.

5.4 RandomWoods

It works like WEKA's classic RandomForest but using CollectiveBagging (classic Bagging, a machine learning ensemble meta-algorithm to improve stability and classification accuracy, extended to make it available to collective classifiers) in combination with CollectiveTree. RandomForest, in contrast, uses Bagging and RandomTree algorithms.

6 Empirical Validation

The research questions we seek to answer through this empirical validation are the following ones:

> *What is the minimum number of labelled instances required to assure a suitable performance in packed executable detection using collective classification?*
>
> *What is the effect in terms of classification accuracy when collective learning algorithms are applied, in comparison to supervised approaches?*

To asses these research questions and, therefore, evaluate our collective packed executable detector, we collected a dataset comprising 2,000 not packed executables and 2,000 packed executables. The first one is composed of 1,000 benign executables and 1,000 malicious executables gathered from the website VxHeavens [50]. The packed samples are divided into 1,000 executables manually packed and 1,000 variants of the malware family 'Zeus' protected by different custom packers. On the one hand, the 1,000 executables manually packed were executables initially not packed, which were protected with 10 different packing tools with different configurations: Armadillo, ASProtect, FSG, MEW, PackMan, RLPack, SLV, Telock, Themida and UPX. On the other hand, the 1,000 variants of the 'Zeus' family were protected with packers that PEiD [4] (updated to the last signature database) was unable to identify. The 'Zeus' family is reported to use multiple-layer packing techniques: the first layer of protection is performed by a custom packer, and a second layer is provided by a well-known packer [51]. All the samples initially considered as not packed were analysed by PEiD [4]. Similarly, all the binaries classified as goodware were obtained from a clean Windows XP installation and were checked as not infected by the ESET NOD32 antimalware tool, updated with the last signature base. It is important to highlight that the malware classification problem was stated as an undecidable problem by Cohen [52]. Likewise, Royal et al. [19] formulated the detection of the unpack-execution of a binary as the problem of determining whether a universal Turing machine simulates a Turing machine on its input tape, also proved as an undecidable problem. This fact implies that we cannot be completely sure about the labels assigned to the binaries included in the dataset. Nevertheless, we consider that the classification provided by the tools employed is the most accurate categorisation we can achieve without manually analising each sample, a task that is beyond our possibilities due to the required labelling efforts.

By means of this dataset, we conducted the following methodology to answer the research question and thus, evaluate the proposed method:

1. **Structural Feature Extraction.** We extracted the features described in Section 3.

2. **Training and Test Generation.** We constructed an ARFF file [53] (i.e., Attribute Relation File Format) with the resultant representations of the executables to build the aforementioned WEKA's classifiers.

   We employed *K-fold cross-validation* [54], a technique which consists on dividing the dataset into K folds, using the instances corresponding to $K-1$ folds for training the model, and the instances in the remaining fold for testing. K training rounds are performed using a different fold for testing each time, and thus, training and testing the model with every possible instance in the dataset.

   In the case of collective and semi-supervised approaches, it is very interesting to evaluate the performance of the algorithms when a different percentage of instances is labelled (training set), in order to evaluate how the algorithms perform when the ammount of labelled data is very low. *K-fold cross-validation* [54] allows us to test the algorithms for a 90%, 80%, 75%, 66% and 50% of labelled instances (training instances) for $k = 10, k = 5, k = 4, k = 3$ and $k = 2$ respectively. Nevertheless, the tool WEKA does not include the option to use lower rates of training instances for cross-validation. In order to solve this limitation, we

modified the *Semi-Supervised Learning and Collective Classification* package[2] for the well-known machine-learning tool WEKA [41] to enable fold inversion[3]. Fold inversion inverts the training and testing instances. In this way, $k = 10$ inverted fold validation will divide the dataset into 10 folds, using 1 fold for training and 9 folds for testing: (10% - 90%) instead of (90% - 10%). We used this modified version of WEKA to configure the experiments conducted to evaluate the algorithms described.

As aforementioned, we used this tool to test different collective algorithms configured with their default parameter. In addition, we also tested the semi-supervised LLGC algorithm and different supervised algorithms used in previous work in order to compare the results obtained and the effects of the collective approach over the performance of the classifiers. In the same way, we evaluated supervised algorithms using the same training and test percentages to evaluate their performance on the generated dataset, and to compare both approaches under the same conditions. More concretely, the different algorithms tested were:

- **Collective algorithms.** More specifically, we used the CollectiveIBK ($k = 10$), CollectiveForest (10 trees), CollectiveWoods (10 trees), CollectiveTree (10 trees) and RandomWoods (10 trees) algorithms.
- **LLGC**. We used the LLGC implementation provided by the Semi-Supervised Learning and Collective Classification package for the well-known machine-learning tool WEKA [53]. Specifically, we configured it with a transductive stochastic matrix, an stochastic matrix and a graph kernel [48] and we employed the Euclidean distance.
- **IBK**. The KNN algorithm was configured for $k = 10$.
- **J48**. This algorithm was configured with the default parameters in WEKA [41].
- **Bagged J48**. This algorithm was configured with 10 iterations and 100% of bag size.
- **RandomForest**. The algorithm was configured with 10 trees.

3. **Testing the Models.** To test the approach, we measured the *True Positive Rate* (TPR), i.e., the number of packed executables correctly detected divided by the total number of packed files:

$$TPR = \frac{TP}{TP + FN} \tag{1}$$

where TP is the number of packed instances correctly classified (true positives) and FN is the number of packed instances misclassified as legitimate software (false negatives).

We also measured the *False Positive Rate* (FPR), i.e., the number of not packed executables misclassified as packed divided by the total number of not packed files:

$$FPR = \frac{FP}{FP + TN} \tag{2}$$

where FP is the number of not packed executables incorrectly detected as packed and TN is the number of not packed executables correctly classified.

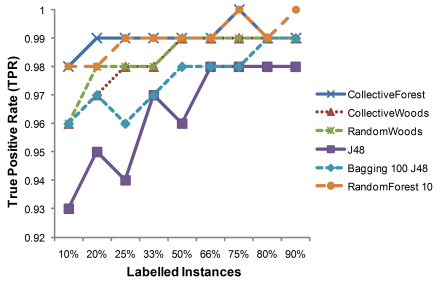
Furthermore, we measured *accuracy*, i.e., the total number of the hits of the classifiers divided by the number of instances in the whole dataset:

$$Accuracy(\%) = \frac{TP + TN}{TP + FP + TP + TN} \tag{3}$$

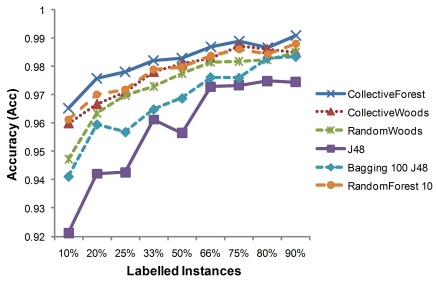
Besides, we measured the *Area Under the ROC Curve* (AUC), which establishes the relation between false negatives and false positives [55]. The ROC curve is obtained by plotting the TPR against the FPR. All these measures refer to the test instances.

[2]Available at: http://www.scms.waikato.ac.nz/~fracpete/projects/collective-classification/downloads.html

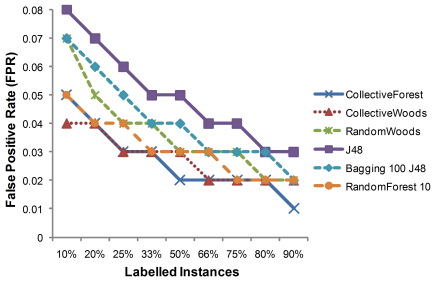
[3]The modified version of the *Semi-Supervised Learning and Collective Classification* package is available at the authors' personal webpage: https://sites.google.com/a/deusto.es/xabier-ugarte/downloads/weka-37-modification



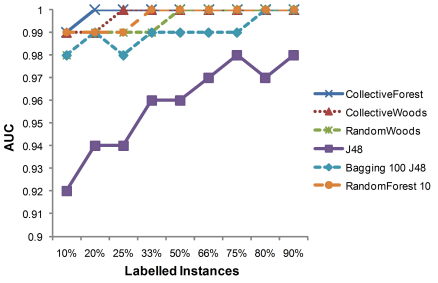
(a) **TPR results.** The TPR axis (Y axis) has been scaled from 0.92 to 1 in order to appreciate better the results obtained for the different configurations. CollectiveForest is, together with RandomForest, the best performing algorithm for all the configurations. The remainder of the classifiers obtained a detection rate of more than 0.93 with a 10% of training size.



(b) **Accuracy results.** The accuracy axis (Y axis) has been scaled from 0.92 to 1 in order to appreciate better the evolution of the algorithms. CollectiveForest presents similar results as RandomForest for all the training configurations. The rest of the classifiers obtained accuracies higher than 0.92 using only a 10% of labelled instances.



(c) **FPR results.** The FPR is scaled from 0 to 0.08 in order to magnify the differences among the configurations. In general, the higher the amount of labelled instances, the lower the FPR. However, the FPR is always lower than 0.08. CollectiveForest and RandomForest are best performing algorithms in terms of FPR, together with CollectiveWoods.



(d) **AUC results.** The AUC axis (Y axis) has been scaled from 0.5 to 1 in order to appreciate better the evolution of the algorithms. As it happened with accuracy, CollectiveForest was, together with RandomForest, the best algorithm among the ones based on decision-tree learning. Anyhow, the rest of the classifiers obtained AUC values higher than 0.9 using only a 10% of labelled instances.

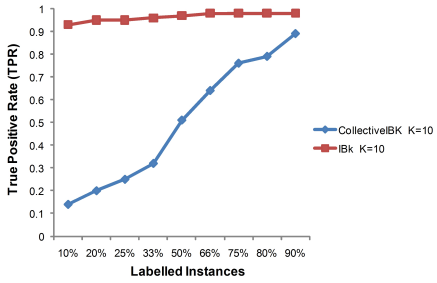
Figure 1: Results of our collective-classification-based packed executable detection method. Collective Forest was the overall classifier with the highest accuracy, TPR and AUC.

Figures 1 and 2 show the obtained results for the collective algorithms and their supervised equivalents tested in terms of accuracy, TPR, FPR and AUC. Our results show that, obviously, the higher the number of labelled executables in the dataset the better the results achieved. However, by using only the 10% of the available data, with the exception of CollectiveIBK, the collective classifiers were able to achieve accuracy rates higher than 94%, TPRs higher than 96% and FPRs lower than 7%. In particular, CollectiveForest trained with the 10% of the data obtained 96.52% of accuracy, 98% of TPR, 5% of FPR and 0.99 of AUC. With the exception of CollectiveIBK, collective algorithms present results similar to their supervised equivalents. Figure 1(b) shows the accuracy results of the proposed method. CollectiveForest presents results similar to RandomForest. CollectiveWoods and RandomWoods, in contrast, present a lower accuracy than these two algorithms, but higher than J48 and Bagged J48.

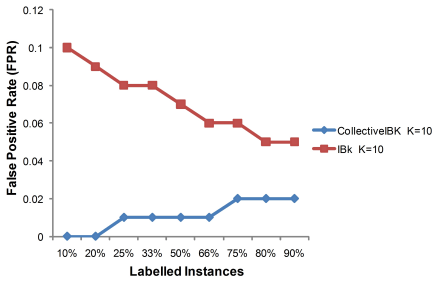
In the case of decision-tree based classifiers, the algorithm which produced the best results in general was Collective Forest, with an accuracy of 96.52% using a 10% of labelled instances for training. Figure 1(a) shows the obtained results in terms of correctly classified packed executables. In this way, Collective Forest was also the best detecting the 98% of the packed executables with only a 10% of the dataset labelled. Figure 1(c) shows the FPR results. In this case, CollectiveForest presents slightly better results than RandomForest. Regarding AUC, shown in Figure 1(d), Collective Forest was again the best, with results higher than 99% for every configuration.

In the case of CollectiveIBK and IBK, shown in Figure 2, the difference between the collective approach and the supervised approach is higher. In this case, the collective algorithm has a significant negative effect on the results in terms of accuracy (shown in Figure 2(b)), TPR (shown

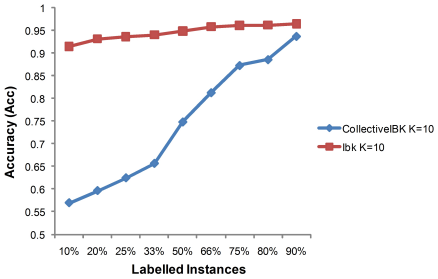
in Figure 2(a), and AUC (shown in Figure 2(d). In contrast, despite being the algorithm with worst overall results, it was the algorithm which achieved the lowest FPR, with results below 2% for all configurations.



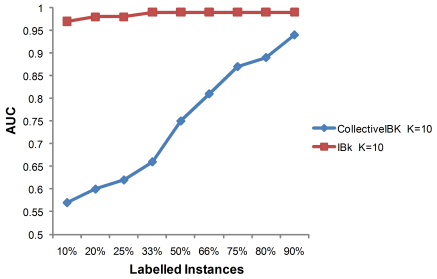
(a) **TPR results.** CollectiveIBK was the only collective classifier that obtained considerably worse results than its supervised equivalent algorithm. In particular, this classifier only achieves a TPR higher than 75% using a labelled dataset size of 75% or higher.



(c) **FPR results.** The FPR is scaled from 0.00% to 7% in order to magnify the differences among the configurations. In contrast to TPR, Collective IBK performs better than its supervised equivalent algorithm in terms of FPR.



(b) **Accuracy results.** The accuracy axis (Y axis) has been scaled from 50% to 100% in order to appreciate better the evolution of CollectiveIBK. This classifier was the only one that was highly sensitive to the increase of the training dataset.



(d) **AUC results.** The AUC axis (Y axis) has been scaled from 50% to 100% in order to appreciate better the evolution of CollectiveIBK. As it happened with accuracy, CollectiveIBK was more sensitive to the increase of the training dataset than its supervised equivalent algorithm.

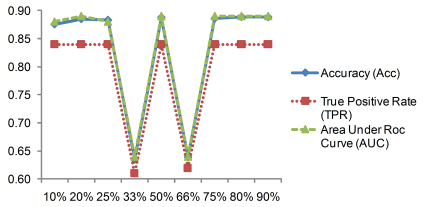
Figure 2: Results of our collective-classification-based packed executable detection method. Collective Forest was the overall classifier with the highest accuracy, TPR and AUC.

Table 2 shows the results for the experimental configurations which achieved the best possible results in terms of AUC using the lowest possible number of instances labelled. We can observe that, while Collective Forest achieved an AUC close to 1 with only a 20% of labelled instances, and Collective Woods needed a 25%, Random Woods needed a 50%, and Collective IBk needed a 90% to achieve a 0.94 AUC.

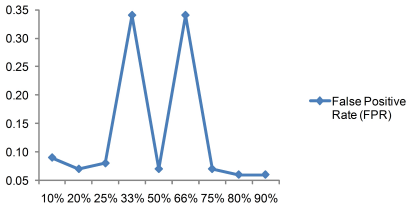
Table 2: Best results obtained for the different semi-supervised algorithms tested. In each case, it was chosen the training-set which had the least possible number of instances labelled while maintaining the best possible results in terms of AUC.

	Labelled %	**ACC.**	**TPR**	**FPR**	**AUC**
Previous semi-supervised approaches					
LLGC	50%	89%	0.84	0.07	0.89
Collective learning					
CollectiveIBK	90%	93.73%	0.89	0.02	0.94
CollectiveForest	20%	97.59%	0.99	0.04	1
CollectiveWoods	25%	97.01%	0.98	0.03	1
RandomWoods	50%	97.75%	0.99	0.03	1

In conclusion, the results indicate that it is only necessary a 10% of labelled instances in order to guarantee results higher than 96.52% in terms of accuracy, in the case of Collective Forest. Moreover, when the number of labelled instances increases, Collective Forest is still the best algorithm achieving a 96.52% of accuracy for a 90% of labelled instances.



(a) **Accuracy, TPR and AUC results.** Accuracy, TPR and AUC are scaled from 0.60 to 0.90, in order to appreciate the evolution of the results when more training instances are labelled. In general, LLGC presents better results in terms of accuracy and AUC when the number of instances labelled increases, whereas the TPR is constant for all configurations.



(b) **FPR results.** The FPR is scaled from 0.05 to 0.35 in order to magnify the differences among the configurations. The evolution of the FPR shows us that increasing the number of labelled instance produces a decrease in the FPR.

Figure 3: Results of the semi-supervised LLGC algorithm based packed executable detection method. The only configuration that achieved significant results was the transductive stochastic matrix, whose evolution is shown in this figure. The configurations corresponding to a 33% of labelled instances and 66% of labelled instances did not achieve sound results because the odd fold division affects the algorithm's performance.

In Figure 3(a) we can see the results achieved by the LLGC algorithm in terms of Accuracy, TPR, and AUC. Figure 3(b) shows the results obtained in terms of FPR. LLGC presents the same evolution of the collective algorithms: the higher the number of instances used for training, the better the results. However, the configurations corresponding to a 33% and 66% of labelled instances present significantly different results. In these cases, the accuracy achieved is between 60% and 65%. The LLGC algorithm is negatively affected by odd fold cross validation configurations. More concretely, we obtained 89% of accuracy for a 50% of labelled instances, being the highest accuracy achieved for all the configurations. In general we can state that collective learning algorithms achieve a better performance than the semi-supervised LLGC algorithm.

In reference to the first research question presented in this section, *what is the minimum number of labelled instances required to assure a suitable performance in packed executable detection using collective classification?*, the response is that our collective learning based method requires only a 10% of the whole dataset in order to guarantee an accuracy higher than 96%, and a 70% to 80% of labelled instances to achieve an accuracy similar to the best results obtained by supervised approaches.

Regarding the second research question presented: *what is the effect in terms of classification accuracy when collective learning algorithms are applied, in comparison to supervised approaches?*, we can state that, for this dataset, collective algorithms do not present a performance significantly sounder than supervised approaches. In some cases, decision-tree based collective algorithms present higher accuracy rates than supervised ones (J48 or Bagged J48). In turn, RandomForest achieves accuracy rates as high as CollectiveForest, the best performing collective algorithm. In fact, the CollectiveForest algorithm configured with a 90% of labelled instances achieved a 99.08% of accuracy, slightly higher than the 98.80% of accuracy achieved by the supervised Random Forest algorithm. The results show that the classification of the dataset proposed is a relatively easy task for supervised algorithms, and that collective and semi-supervised approaches have not demonstrated their actual capacity to outperform supervised approaches when the amount of labelled data is limited.

Finally, if we compare the results achieved with previous packed executable identification methods, we can see that collective learning algorithms and RandomForest outperform other algorithms used in previous work [28, 56]. Table 3 compares the work of Perdisci et al. [28, 56] with several supervised learning algorithms. In order to compare the results of the different approaches we tested the algorithms employed by Perdisci et al. with two different feature sets: the features extracted by Perdisci et al., and the features we describe in section 3. It is important to remark that

Table 3: Results obtained for the algorithms employed in the work of Perdisci et al. [28] extracting their 9 features from our dataset. In this way we compare the results obtained for the different feature sets under the same conditions. We applied k-fold cross validation for $k = 10$ for both feature sets. In addition to the supervised algorithms described previously, we have tested Naive Bayes and Multi-layer Perceptron (configured with their default parameters in WEKA [41]).

	ACC.	**TPR**	**FPR**	**AUC**
Method of Perdisci et al. [28] applied to our dataset				
Naive Bayes	78.19%	0.64	0.07	0.92
J48	95.69%	0.96	0.05	0.96
Bagged J48	96.37%	0.97	0.04	0.99
IBk k = 3	97.19%	0.98	0.04	0.99
MLP	94.34%	0.96	0.07	0.96
Entropy Thres.	88.62%	0.88	0.10	0.89
Supervised approaches				
NaiveBayes	67.55%	0.38	0.03	0.94
J48	97.45%	0.98	0.03	0.98
Bagged J48	98.35%	0.99	0.02	1.00
IBK - $k = 3$	97.83%	0.98	0.03	0.99
MLP	94.52%	0.96	0.07	0.95

our feature set includes the features proposed by Perdisci et al. and adds values extracted from the PE structure of the executable. We can observe that our feature set outperforms the work of Perdisci et al. for the J48, Bagged J48 and IBk with k=3. In addition, we applied more supervised machine learning algorithms to our feature set, obtaining the best results with the RandomForest algorithm configured with 10 trees, achieving a 98.80% of accuracy. In conclusion, we can state that our feature set produces sounder results for several of the algorithms proposed by Perdisci et al..

7 Discussion and Concluding Remarks

Our main objective was to pre-filter packed executables, as an initial phase to decide whether it is necessary to analyse samples using a generic unpacker or not. More concretely, the aim of this collective-learning approach was to achieve high accuracy rates while reducing the required number of labelled executables. The results obtained show that the method proposed has accomplished this goal. However, there are several shortcomings that should be discussed:

- Our approach is not able to identify neither the packer nor the family of the packer used to cypher the executable. This information can help malware analysts to unpack the executable and to create new unpacking routines. Sometimes, generic unpacking techniques are very time consuming or fail and it is easier to use specific unpacking routines, created for most widespread packers.
- The features extracted can be modified by malware writers to bypass the filter. In the case of structural features, packers can build executables using the same flags and patterns as common compilers, for instance importing common DLL files or creating the same number of sections. Heuristic analysis, in turn, can be evaded using standard sections instead of not standard ones, or filling sections with padding data to unbalance byte frequency and obtain lower entropy values. Our system is very dependant on heuristics due to the relevance values obtained from IG, making it vulnerable to such attacks.

The last limitation of our approach is also applicable to other methods for detecting packed executables. In fact, in our approach we use every feature that has been used in previous work [26, 27, 28] and we add several of our own, such as different entropy values or ratio between data and code sections. Nevertheless, these features are heuristics employed by common packers. For instance, new packing techniques like virtualisation [57, 58, 18], which consists on generating a

virtual machine to execute the malicious behaviour using an unknown set of instructions within it, do not necessarily increase the entropy of the file.

Despite the ability of malware packers to surpass these heuristics is a real problem, the majority of the packed executables are packed with known packers like UPX. Besides, there is an increasing number of malicious executables packed with custom packers. In our validation, we have included a big number of this type of malware: 1,000 variants of the Zeus family, gathered from 2009 to 2011, for which PEiD was not able to detect the packer employed. More concretely, this family is reported to be protected with custom packers [51].

In any case, our approach was able to detect the majority of these custom packers. Our results show that, although some of the features mentioned in section 3 may be evaded by different techniques, it is not trivial to evade all of them while maintaining the whole functionality.

Malware detection is a critical topic of research due to its increasing ability to hide itself. Malware writers usually employ executable packing techniques that hide the real malicious code through encryption or similar techniques. Generic unpacking techniques that use a contained environment pose as a solution in order to face these executables. However, these environments perform their task in a high-resource-consuming fashion. Therefore, approaches for executable pre-filtering have been also proposed that, based on several static features, are capable of determining whether an executable is packed or not. These approaches usually employ supervised learning approaches in order to classify executables. The problem with supervised learning is that a previous work of executable labelling is required. This process in the field of malware can introduce a high performance overhead due to the number of new threats in-the-wild. In this paper, we have proposed the first collective-learning-based packed executable detection system that based upon structural features and heuristics is able to determine when an executable is packed. We have empirically validated our method using a dataset composed of packed executables (protected with known packers and custom packers), and not packed executables, showing that our technique, despite having less labelling requirements, still presents sound results. In addition, we have compared the results obtained with several supervised machine learning algorithms. In this case, although CollectiveForest achieves slightly sounder results than any of the supervised algorithms tested, there is not a considerable difference between both approaches due to the properties of the dataset. In any way, little differences in detection accuracy in this kind of problems can make the difference between detecting a new malware binary or not detecting it, or, in other words, avoiding potential productivity losses for the companies or not.

The avenues of future work are oriented in three main ways. First, we plan to extend this approach with an specific packer detector able to discriminate between executables packed with a known packer and the ones with a custom packer in order to apply a concrete unpacking routine or a dynamic generic step. Second, we plan to test more semi-supervised techniques in order to reduce the labelling efforts required for the machine-learning classifier training process. Finally, we plan to study the attacks that malicious software filtering systems can suffer.

Acknowledgments

This research was partially supported by the Basque Government under a pre-doctoral grant given to Xabier Ugarte-Pedrero. We would also like to acknowledge S21Sec for the Zeus malware family samples provided in order to set up the experimental dataset.

References

[1] Ször, P. *The art of computer virus research and defense*. Addison-Wesley Professional, 2005.

[2] Santos, I., Brezo, F., Nieves, J., Penya, Y. et al. *Idea: Opcode-Sequence-Based Malware Detection*. In *Engineering Secure Software and Systems*, volume 5965 of *LNCS*, pages 35–43. 2010. 10.1007/978-3-642-11747-3_3.

[3] McAfee Labs. *McAfee Whitepaper: The Good, the Bad, and the Unknown*, 2011. Available online: `http://www.mcafee.com/us/resources/white-papers/wp-good-bad-unknown.pdf`.

[4] PEiD. *PEiD Webpage*, 2010. Available online: `http://www.peid.info/`.

[5] Faster Universal Unpacker, 1999. Available online: `http://code.google.com/p/fuu/`.

[6] Morgenstern, M. and Pilz, H. *Useful and useless statistics about viruses and anti-virus programs*. In *Proceedings of the CARO Workshop*. 2010. Available online: `www.f-secure.com/weblog/archives/Maik_Morgenstern_Statistics.pdf`.

[7] Chapelle, O., Schölkopf, B. and Zien, A. *Semi-supervised learning*. MIT Press, 2006.

[8] Neville, J. and Jensen, D. *Collective classification with relational dependency networks*. In *Proceedings of the Workshop on Multi-Relational Data Mining (MRDM)*. 2003.

[9] Laorden, C., Sanz, B., Santos, I., Galán-García, P. et al. *Collective Classification for Spam Filtering*. In *Proceedings of the 4^{th} International Conference on Computational Intelligence in Security for Information Systems (CISIS)*, pages 1–8. 2011.

[10] Santos, I., Sanz, B., Laorden, C., Brezo, F. et al. *Opcode-sequence-based Semi-supervised Unknown Malware Detection.*. In *Proceedings of the 4^{th} International Conference on Computational Intelligence in Security for Information Systems (CISIS)*, pages 50–57. 2011.

[11] Santos, I., Laorden, C. and Bringas, P. G. *Collective Classification for Unknown Malware Detection*. In *Proceedings of the 6^{th} International Conference on Security and Cryptography (SECRYPT)*, pages 251–256. 2011.

[12] Ugarte-Pedrero, X., Santos, I. and Bringas, P. G. *Structural Feature based Anomaly Detection for Packed Executable Identification*. In *Proceedings of the 4^{th} International Conference on Computational Intelligence in Security for Information Systems (CISIS)*, pages 50–57. 2011.

[13] Ugarte-Pedrero, X., Santos, I. and Bringas, P. *Semi-supervised Learning for Packed Executable Detection*. In *In Proceedings of the 7th International Conference on Information Security and Cryptology (INSCRYPT)*. 2011. In press.

[14] Ugarte-Pedrero, X., Santos, I., Bringas, P., Gastesi, M. et al. *Semi-supervised Learning for Packed Executable Detection*. In *In Proceedings of the 5th International Conference on Network and System Security (NSS)*, pages 342–346. 2011.

[15] Babar, K. and Khalid, F. *Generic unpacking techniques*. In *Proceedings of the 2^{nd} International Conference on Computer, Control and Communication (IC4)*, pages 1–6. IEEE, 2009.

[16] Data Rescue. *Universal PE Unpacker plug-in*. Available online: `http://www.datarescue.com/idabase/unpack_pe`.

[17] Stewart, J. *Ollybone: Semi-automatic unpacking on ia-32*. In *Proceedings of the 14^{th} DEF CON Hacking Conference*. 2006.

[18] Rolles, R. *Unpacking virtualization obfuscators*. In *Proceedings of 3^{rd} USENIX Workshop on Offensive Technologies.(WOOT)*. 2009.

[19] Royal, P., Halpin, M., Dagon, D., Edmonds, R. et al. *Polyunpack: Automating the hidden-code extraction of unpack-executing malware*. In *Proceedings of the 2006 Annual Computer Security Applications Conference (ACSAC)*, pages 289–300. 2006.

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

[21] Martignoni, L., Christodorescu, M. and Jha, S. *Omniunpack: Fast, generic, and safe unpacking of malware*. In *Proceedings of the 2007 Annual Computer Security Applications Conference (ACSAC)*, pages 431–441. 2007.

[22] Yegneswaran, V., Saidi, H., Porras, P., Sharif, M. et al. *Eureka: A framework for enabling static analysis on malware*. Technical report, Technical Report SRI-CSL-08-01, 2008.

[23] Danielescu, A. *Anti-debugging and anti-emulation techniques*. CodeBreakers Journal, 5(1), 2008. Available online: `http://www.codebreakers-journal.com/`.

[24] Cesare, S. *Linux anti-debugging techniques, fooling the debugger*, 1999. Available online: `http://vx.netlux.org/lib/vsc04.html`.

[25] Julus, L. *Anti-debugging in WIN32*, 1999. Available online: `http://vx.netlux.org/lib/vlj05.html`.

[26] Farooq, M. *PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime*. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 121–141. Springer-Verlag, 2009.

[27] Shafiq, M., Tabish, S. and Farooq, M. *PE-Probe: Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables*. In *Proceedings of the Virus Bulletin Conference (VB)*. 2009.

[28] Perdisci, R., Lanzi, A. and Lee, W. *McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables*. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC)*, pages 301–310. 2008. ISSN 1063-9527.

[29] Santos, I., Penya, Y., Devesa, J. and Bringas, P. *N-Grams-based file signatures for malware detection*. In *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS), Volume AIDSS*, pages 317–320. 2009.

[30] Santos, I., Ugarte-Pedrero, X., Sanz, B., Laorden, C. et al. *Collective Classification for Packed Executable Identification*. In *Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)*, pages 23–30. 2011.

[31] Kent, J. *Information gain and a general measure of correlation*. Biometrika, 70(1), 163–173, 1983.

[32] Lyda, R. and Hamrock, J. *Using entropy analysis to find encrypted and packed malware*. IEEE Security & Privacy, 5(2), 40–45, 2007. ISSN 1540-7993.

[33] Bishop, C. *Pattern recognition and machine learning*. Springer New York., 2006.

[34] Kotsiantis, S. *Supervised Machine Learning: A Review of Classification Techniques*. In *Proceeding of the 2007 conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, pages 3–24. 2007.

[35] Kotsiantis, S. and Pintelas, P. *Recent advances in clustering: A brief survey*. WSEAS Transactions on Information Science and Applications, 1(1), 73–81, 2004.

[36] Pearl, J. *Reverend bayes on inference engines: a distributed hierarchical approach*. In *Proceedings of the National Conference on Artificial Intelligence*, pages 133–136. 1982.

[37] Bayes, T. *An essay towards solving a problem in the doctrine of chances*. Philosophical Transactions of the Royal Society, 53, 370–418, 1763.

[38] Castillo, E., Gutiérrez, J. M. and Hadi, A. S. *Expert Systems and Probabilistic Network Models*. Springer, New York, NY, USA, erste edition, 1996. ISBN 0387948589.

[39] Quinlan, J. *C4. 5 programs for machine learning*. Morgan Kaufmann Publishers, 1993.

[40] Breiman, L. *Random forests*. Machine learning, 45(1), 5–32, 2001.

[41] Garner, S. *Weka: The Waikato environment for knowledge analysis*. In *Proceedings of the New Zealand Computer Science Research Students Conference*, pages 57–64. 1995.

[42] Breiman, L. *Bagging predictors*. Machine Learning, 24, 123–140, 1996. ISSN 0885-6125. 10.1007/BF00058655.

[43] Fix, E. and Hodges, J. L. *Discriminatory analysis: Nonparametric discrimination: Small sample performance*. Technical Report Project 21-49-004, Report Number 11, 1952.

[44] Vapnik, V. *The nature of statistical learning theory*. Springer, 2000.

[45] Amari, S. and Wu, S. *Improving support vector machine classifiers by modifying kernel functions*. Neural Networks, 12(6), 783–789, 1999.

[46] Kriesel, D. *A Brief Introduction to Neural Networks, Zeta version*. 2007. Available online: `http://www.dkriesel.com`.

[47] Zhang, G. P. *Neural networks for classification: a survey*. IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews), 30(4), 451–462, 2000. ISSN 10946977. doi:10.1109/5326.897072.

[48] Zhou, D., Bousquet, O., Lal, T., Weston, J. et al. *Learning with local and global consistency*. In *Advances in Neural Information Processing Systems 16: Proceedings of the 2003 Conference*, pages 595–602. 2004.

[49] Namata, G., Sen, P., Bilgic, M. and Getoor, L. *Collective classification for text classification*. Text Mining, pages 51–69, 2009.

[50] VX Heavens. Available online: `http://vx.netlux.org/`.

[51] Wyke J. *What is Zeus? Technical paper.*.

[52] Cohen, F. *Computer viruses:: Theory and experiments*. Computers & security, 6(1), 22–35, 1987.

[53] Holmes, G., Donkin, A. and Witten, I. H. *WEKA: a machine learning workbench*. pages 357–361. 1994.

[54] Kohavi, R. *A study of cross-validation and bootstrap for accuracy estimation and model selection*. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 14, pages 1137–1145. 1995.

[55] Singh, Y., Kaur, A. and Malhotra, R. *Comparative analysis of regression and machine learning methods for predicting fault proneness models*. International Journal of Computer Applications in Technology, 35(2), 183–193, 2009.

[56] Perdisci, R., Lanzi, A. and Lee, W. *Classification of packed executables for accurate computer virus detection*. Pattern Recognition Letters, 29(14), 1941–1946, 2008. ISSN 0167-8655.

[57] Sharif, M., Lanzi, A., Giffin, J. and Lee, W. *Automatic reverse engineering of malware emulators*. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pages 94–109. 2009.

[58] Sharif, M., Lanzi, A., Giffin, J. and Lee, W. *Rotalumè: A Tool for Automatic Reverse Engineering of Malware Emulators*. 2009.