Collective Classification for Packed Executable Identification

Igor Santos
isantos@deusto.es

Xabier Ugarte-Pedrero
xabier.ugarte@deusto.es

Borja Sanz
borja.sanz@deusto.es

Carlos Laorden
claorden@deusto.es

Pablo G. Bringas
pablo.garcia.bringas.es

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

ABSTRACT

Malware is any software designed to harm computers. Commercial anti-virus are based on signature scanning, which is a technique effective only when the malicious executables have been previously analysed and identified. Malware writers employ several techniques in order to hide their actual behaviour. Executable packing consists in encrypting or hiding the real payload of the executable. Generic unpacking techniques do not depend on the packer used, as they execute the binary within an isolated environment (namely 'sandbox') to gather the real code of the packed executable. However, this approach is slow and, therefore, a filter step is required to determine when an executable has been packed. To this end, supervised machine learning approaches trained with static features from the executables have been proposed. Notwithstanding, supervised learning 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 a collective learning approach to reduce the labelling requirements of completely supervised approaches. We performed an empirical validation demonstrating that the system maintains a high accuracy rate while the labelling efforts are lower than when using supervised learning.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*invasive software*

General Terms

Security

Keywords

malware, machine learning, executable packing

1. INTRODUCTION

Malware is defined as any computer software that damages computers or networks [30, 24]. 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 [15], up to the 80% of the detected malware is packed.

Because malware creators generally use currently known packers, packer signatures have been adopted by commercial antiviruses. For instance, PEID [19] can detect a wide range of well-known packers while Faster Universal Unpacker (FUU) [7] identifies the packer to, then, apply custom unpacking routines designed and specifically coded for each different packer. However, this approach has the same shortcomings as signatures for malware detection: it is not effective for unknown obfuscation techniques or custom packers (i.e., protectors implemented for certain malware samples). This limitation is a problem because, according to Morgenstern and Pilz [16], the 35 % of malware is packed by a custom packer.

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 an emulator [1]. Then, the execution is traced and stopped when certain events occur.

Various dynamic unpackers (e.g., Universal PE Unpacker [5] and OllyBonE [29]) 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 (e.g., [22]) do not unpack the code before executing it: 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 [23] obtains an static view of the code and then compares it to the code dynamically generated during execution. If both codes differ, the executable is considered to be packed. Renovo [11] monitors memory accesses and checks whether any memory area has been written and then executed or not. OmniUnpack [14] statically analyses memory pages that have been over-written and then executed only when a dangerous system call is performed. Finally, Eureka [33] 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 [4, 2, 10].

By using the structural information of PE executables, some methods (e.g., PE-Miner [6], PE-Probe [25] and Perdisci et al. [21]) 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. These methods employ several features to train supervised learning methods.

However, these supervised machine-learning classifiers require a high number of labelled executables for 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 [3]. In particular, collective classification [18] is an 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.

Against 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 [31], which presented an anomaly detector for packed executable detection.

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 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 the structural features employed for packed executable identification. Section 3 describes different collective classification methods and how they can be adopted for packed executable detection. Section 4 describes the experiments and presents results. Finally, Section 5 concludes the paper and outlines avenues for future work.

2. STRUCTURAL FEATURES OF THE PE FILES

Given the conclusions obtained in previous work [6, 25, 21], 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 community.

We consider that one of the main requisites of our detection system is 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 too much processing time, and avoided techniques such as code disassembly, string extraction or n-gram analysis [21], which could slow down sample comparison.

Features can be divided into four main groups: 125 raw header characteristics [25], 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) [12]. IG provides a ratio for each characteristic that measures its importance to consider if a sample is packed or not. These weights were calculated from a dataset composed of 1,000 packed executables (from which 500 belong to the Zeus malware family and thus, are protected with custom packers, and the other 500 are executables packed with 10 different packers available on-line), and 1,000 not packed executables. We reduced the amount of selected characteristics to obtain a more efficient classification, given that only 166 of them have a non-zero IG value.

2.1 DOS header characteristics

The first bytes of the PE file header correspond to the DOS executable fields. IG results showed that these characteristics are not specially relevant, having a maximum IG value of 0.13, corresponding to the field containing the address of the PE header. 19 values range from 0.05 to 0.13, and the rest present a relevance bellow 0.05.

2.2 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, 11 have an IG value greater than 0, and only 3 have an IG value greater than 0.10:

- Header characteristics field (0.51185).
- Time stamp (0.32806).
- Number of sections (0.16112).

2.3 Optional Header Block

The 71 features of this optional block, which is only part of image files, contain data about how the executable must be loaded into memory. 51 characteristics have an IG value over 0, but the most relevant ones are:

- Major linker version (0.53527).
- Import Address Table size (0.3763).
- Size of heap commit (0.34607).
- Major operating system version (0.34302).
- Size of the resource table (0.33863).
- Size of heap reverse (0.33196).
- Size of code (0.30799).
- Address of entry point (0.28229).
- Minor linker version (0.26716).
- Size of the debug data (0.2425).
- Base of data (0.21667).
- Size of initialized data (0.19152).

2.4 Section characteristics

From the 31 characteristics that conform this group, 24 have an IG value greater than 0. The most significant ones are:

- First section raw data size (0.44452).
- Number of standard sections (0.34731).
- Number of sections with virtual size greater than raw data (0.34432).
- Number of sections that are readable and writeable (0.33754).
- Maximum raw data per virtual size ratio (0.32552) ($rawSize/virtualSize$, where $rawSize$ is the section raw data size, and $virtualSize$ is the section virtual size, both expressed in bytes).
- Number of sections that contain initialized data (0.31558).
- Number of non standard sections (0.2888).
- Number of executable sections (0.27776).
- Number of readable, writeable and executable sections (0.25601).
- Number of sections that are readable and executable (0.23809).
- Number of readable sections (0.23608).
- Minimum raw data per virtual size ratio (0.22251).

2.5 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. 18 characteristics have an IG value greater than 0, from which 9 have a significant relevance:

- Size of raw data (0.42061).
- Raw data per virtual size ratio (0.37729).
- Virtual size (0.355).
- Pointer to raw data (0.32303).
- Header characteristics (0.31739).
- Writeable section (0.3132).
- Virtual address (0.28992).
- Standard section (0.19317)
- A flag that is set if the section contains initialized data (0.12998).

2.6 Entropy values

We have selected 24 entropy values, commonly used in previous works [13], and all of them have an IG value greater than 0. Concretely, 8 have a relevant IG value:

- Global file entropy (0.67246).
- Maximum entropy (0.63232).
- Mean section entropy (0.56581).
- Section of entry point entropy (0.52056).
- Mean code section entropy (0.51106).
- Mean data section entropy (0.4325).
- The number of sections with entropy in the range between 7.5 and 8 (0.32336).
- Header entropy (0.19541).
- The number of sections with entropy in the range between 7 and 7.5 (0.10074).

3. COLLECTIVE CLASSIFICATION

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 [17]. 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, $\mathcal{L} = \{l_1, ..., l_q\}$.

We use the *Waikato Environment for Knowledge Analysis* (WEKA) [8] and its Semi-Supervised Learning and Collective Classification plugin[1]. In the remainder of this section we review the collective algorithms used in the empirical evaluation.

[1]Available at: http://www.scms.waikato.ac.nz/~fracpete/projects/collectiveclassification

3.1 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.

3.2 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.

3.3 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.

3.4 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.

4. EMPIRICAL VALIDATION

The research question we seek to answer through this empirical validation is the following one:

> *What is the minimum number of labelled instances required to assure a suitable performance in packed executable detection using collective classification?*

To asses this research question and, therefore, evaluate our semi-supervised packed executable detector, we collected a dataset comprising 1,000 not packed executables and 1,000 packed executables. The first one is composed of 500 benign executables and 500 malicious executables gathered from the website VxHeavens [32]. The packed samples are divided into 500 executables manually packed and 500 variants of the malware family 'Zeus' protected by different custom packers. On the one hand, the 500 executables manually packed were not packed executables protected with 10 different packing tools with different configurations: Armadillo, ASProtect, FSG, MEW, PackMan, RLPack, SLV, Telock, Themida and UPX. On the one hand, the 500 variants of the 'Zeus' family were protected with packers that PEiD [19] (updated to the last signature database) was unable to identify.

Hereafter, we extracted the structural representation for each file in the dataset. Next, we split the dataset into different percentages of training and test instances. In other words, we changed the number of labelled instances from 10% to 90% to measure the effect of the number of previously labelled instances on the final performance of collective classification in detecting packed executables.

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 2.
2. **Feature Selection.** We reduced the number of features to only the ones that obtained an IG value greater than 0.
3. **Training and Test Generation.** We constructed an ARFF file [9] (i.e., Attribute Relation File Format) with the resultant vector representations of the executables to build the aforementioned WEKA's classifiers.

   We did not use cross-validation because in the evaluation we did not want to test the performance of the classifier when a fixed size of training instances is used iteratively. Otherwise, we employed a variable number of training instances and tried to predict the class of the remaining ones using collective classification in order to determine which is the best training set size. In this case, the training instances are the labelled ones whereas the unlabelled ones are the ones in the test dataset.

   Therefore, we split the dataset into different percentages of training and tested instances, changing the number of labelled instances from 10% to 90% to measure the effect of the number of labelled instances on the final performance of collective classification in detecting packed executables.

As aforementioned, we used the collective classification implementations provided by the *Semi-Supervised Learning and Collective Classification* package[2] for the well-known machine-learning tool WEKA [8]. All the classifiers were tested with their default parameters.

4. **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 [28]. The ROC curve is obtained by plotting the TPR against the FPR. All these measures refer to the test instances.

Figure 1 shows the obtained results in terms of accuracy, TPR, FPR and AUC. Our results outline that, obviously, the higher the number of labelled executables in the dataset the better results achieved. However, by using only the 10% of the available data, with the exception of CollectiveIBK, the collective classifiers were able to achieve TPRs higher than 99% and FPRs lower than 0.5%. In particular, CollectiveForest trained with the 10% of the data obtained 99.78% of accuracy, 99.90% of TPR, 0.30% of FPR and 99.90% of AUC. Figure 1(a) shows the accuracy results of our proposed method. All the tested classifiers, with the exception of CollectiveIBK, achieved accuracy results higher than 99%. In particular, CollectiveForest was the best, achieving an accuracy of 99.8% using only a 10% of the instances for training. Figure 1(b) shows the obtained results in terms of correctly classified packed executables. In this way, Collective Forest was also the best detecting the 99.9% of the packed executables with only a 10% of the dataset labelled. Figure 1(c) shows the FPR results. Every classifier obtained results lower than 1%. In particular, the lowest FPR achieved was of 0.01%, achieved by CollectiveIBK with the 10% of dataset. However, in order to guarantee results of TPR lower than 5%, Collective Forest only needs to be trained with, at least, 20% of the dataset. Finally, regarding AUC, shown in Figure 1(d), Collective Forest was again the best, with results higher than 99% for every configuration.

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

Note that our method is devoted to pre-filter packed executables, as an initial phase to decide whether it is necessary to analyse samples using a generic unpacker or not. Our main contribution to this topic is the adoption of collective classification employed for packed executable detection.

Table 1: Comparison with previous work in packed executable detection. The abbreviation *Acc.* stands for Accuracy. Our method outperforms previous work in packed executable detection using less instances to train models.

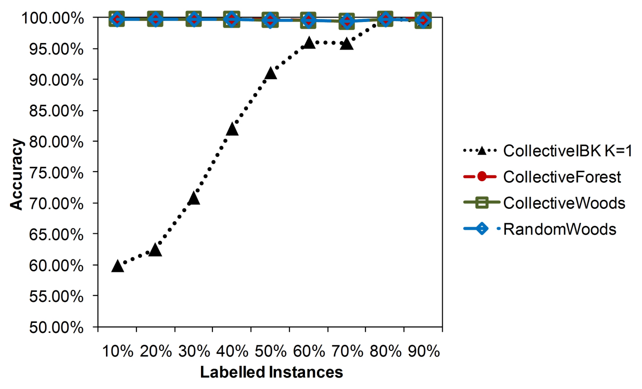
Approach	Acc.	TPR	FPR	AUC
Perdisci et al. [21]	0.994	0.996	0.008	0.997
CollectiveForest (10% of the dataset)	0.998	0.999	0.003	0.999

If compared with previous packed executable identification methods, our technique does not require as much previously labelled packed and not packed executables as the supervised approaches do. The accuracy our system raises is also higher than the obtained with fully supervised approaches. Table 1 compares the work of Perdisci et al. [21, 20] with our semi-supervised approach. Albeit the approach of Perdisci et al., used fully supervised methods our method outperforms it. Besides, our collective learning based method requires less executables to be labelled: a 10% of the whole dataset in order to guarantee an accuracy higher than 99%. Note that we did not use the same dataset employed by Perdisci et al., which may alter the results. However, we did use different custom packers in our evaluation (a 50% of the packed samples were packed by different custom packers), which validates our method against custom protectors.

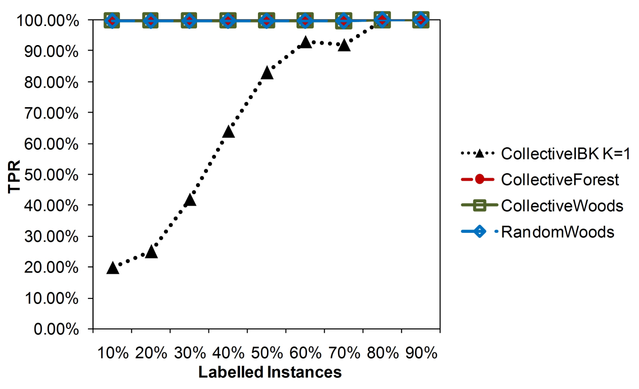
Our main objective was to achieve high accuracy rates while reducing the required number of labelled executables and we consider that our method has accomplished it. However, there are several shortcomings that should be discussed:

- 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.

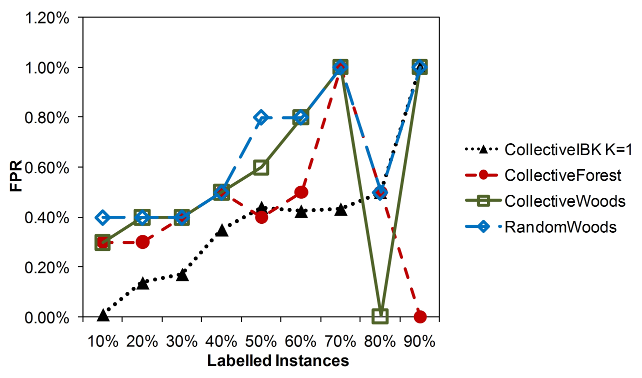
- The features extracted can be modified by malware writers to bypass the filter. In the case of structural features, packers can build executables using the same flags and patterns as common compilers, for instance importing common DLL files or creating the same number of sections. Heuristic analysis, 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



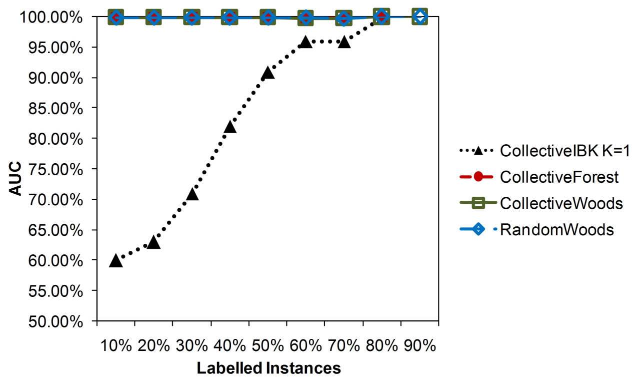
(a) **Accuracy results.** The accuracy axis (Y axis) has been scaled from 50% to 100% in order to appreciate better the evolution of CollectiveIBK. This classifier was the only one that was highly sensitive to the increase of the training dataset. The rest of the classifiers obtained accuracies higher than 99.5% using only a 10% of labelled instances.



(b) **TPR results.** CollectiveIBK was the only classifier that obtained low results using a low amount of labelled instances. In particular, this classifier only achieves a TPR higher than 80% using a labelled dataset size of 60% or higher. The remainder of the classifiers obtained a detection rate of more than 99% with a 10% of training size.



(c) **FPR results.** The FPR is scaled from 0.00% to 1.20% in order to magnify the differences among the configurations. In general, the higher the amount of labelled instances, the higher the FPR. However, the FPR is always lower than 1.20%, which is a sound result. Indeed, using only a 10% of the instances, we can achieve a FPR lower than 0.4% using CollectiveForest.



(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 the only one that was highly sensitive to the increase of the training dataset. Anyhow, the rest of the classifiers obtained AUC values higher than 99.9% 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.

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 [6, 25, 21] and we add several of our own, like different entropy values, ratio between data and code sections and so on. Nevertheless, these features are heuristics employed by common packers. Anyhow, new packing techniques like virtualization [26, 27, 22], 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: 500 variants of the Zeus family, gathered from 2009 to 2011, which PEiD was not able to detect as packed as they were protected with custom packers. Our approach was able to detect the majority of these custom packers. Our results show that, although some of the features mentioned in section 2 may be evaded by different techniques, it is not trivial to evade all of them while maintaining the whole functionality.

5. CONCLUDING REMARKS

Malware detection is a critical topic of research due to its increasing ability to hide itself. Current malware usually employs 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 that appear 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, with both known packers and custom packers, and not packed executables, showing that our technique, despite having less labelling requirements, outperforms previous work.

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 compare the results obtained by collective approaches. Finally, we plan to study the attacks that malicious software filtering systems can suffer.

6. 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.

7. REFERENCES

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

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

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

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

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

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

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

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

[9] G. Holmes, A. Donkin, and I. H. Witten. Weka: a machine learning workbench. pages 357–361, August 1994.

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

[11] M. 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. ACM, 2007.

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

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

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

[15] 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`.

[16] M. Morgenstern and H. Pilz. 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`.

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

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

[19] PEiD. PEiD webpage, 2010. Available online: `http://www.peid.info/`.

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

[21] R. Perdisci, A. Lanzi, and W. Lee. 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.

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

[23] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. 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.

[24] I. Santos, F. Brezo, J. Nieves, Y. Penya, B. Sanz, C. Laorden, and P. Bringas. 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.

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

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

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

[28] Y. Singh, A. Kaur, and R. Malhotra. 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.

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

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

[31] X. Ugarte-Pedrero, I. Santos, and P. G. Bringas. Structural feature based anomaly detection for packed executable identification. In *Proceedings of the 4th International Conference on Computational Intelligence in Security for Information Systems (CISIS)*, pages 50–57, 2011.

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

[33] V. Yegneswaran, H. Saidi, P. Porras, M. Sharif, and W. Mark. Eureka: A framework for enabling static analysis on malware. Technical report, Technical Report SRI-CSL-08-01, 2008.