Structural Feature based Anomaly Detection for Packed Executable Identification

Xabier Ugarte-Pedrero, Igor Santos, and Pablo G. Bringas

S^3Lab, DeustoTech - Computing, Deusto Institute of Technology
University of Deusto,
Avenida de las Universidades 24, 48007
Bilbao, Spain
{xabier.ugarte,isantos,pablo.garcia.bringas}@deusto.es

Abstract. Malware is any software with malicious intentions. Commercial anti-malware software relies on signature databases. This approach has proven to be effective when the threats are already known. However, malware writers employ software encryption tools and code obfuscation techniques to hide the actual behaviour of their malicious programs. One of these techniques is executable packing, which consists of encrypting the real code of the executable so that it is decrypted in its execution. Commercial solutions to this problem try to identify the packer and then apply the corresponding unpacking routine for each packing algorithm. Nevertheless, this approach fails to detect new and custom packers. Therefore, generic unpacking methods have been proposed which execute the binary in a contained environment and gather its actual code. However, these approaches are very time-consuming and, therefore, a filter step is required that identifies whether an executable is packed or not. In this paper, we present the first packed executable detector based on anomaly detection. This approach represents not packed executables as feature vectors of structural information and heuristic values. Thereby, an executable is classified as packed or not packed by measuring its deviation to the representation of normality (not packed executables). We show that this method achieves high accuracy rates detecting packed executables while maintaining a low false positive rate.

Key words: malware, anomaly detection, computer security, packer

1 Introduction

Malware (or malicious software) is defined as computer software that has been explicitly designed to harm computers or networks. The amount, power and variety of malware programs increases every year, as does its ability to overcome all kinds of security barriers [1]. Currently, malware writers use executable packing techniques (cyphering or compressing the actual malicious code) to hide the actual behaviour of their creations. Packed programs have a decryption routine that extracts the real payload from memory and then executes it. Currently, up to the 80 % of the detected malware is packed [2].

Signatures have been applied for the detection of packed executables (e.g., PEID [3] and Faster Universal Unpacker (FUU) [4]). However, this approach has the same shortcoming as signatures for malware detection: it is not effective with unknown obfuscation techniques or custom packers. Indeed, according to Morgenstern and Pilz [5], the 35 % of malware is packed by a custom packer.

Dynamic unpacking approaches monitor the execution of a binary in order to extract its actual code. These methods execute the samples inside an isolated environment that can be deployed as a virtual machine or an emulator [6]. Several dynamic unpackers use heuristics to determine the exact point where the execution jumps from the unpacking routine to the original code and once reached, bulk the memory content to obtain an unpacked version of the malicious code (e.g., Universal PE Unpacker [7] and OllyBonE [8]). Notwithstanding, concrete heuristics cannot be applied to all the packers in the wild, as all of them work in very different manners. In contrast, not so highly heuristic-dependant approaches have been proposed for generic dynamic unpacking (e.g., PolyUnpack [9], Renovo [10], OmniUnpack [11] and Eureka [12]). Nonetheless, these methods are very tedious and time-consuming, and cannot counter conditional execution of unpacking routines, a technique used for anti-debugging and anti-monitoring defense [13–15]. Another common approach is using the structural information of the PE executables to determine if the sample under analysis is packed or if it is suspicious of containing malicious code (e.g., PE-Miner [16], PE-Probe [17] and Perdisci et al. [18]).

In light of this background, we propose here the first method that applies anomaly detection to packed executable filtering as a previous phase to dynamic and generic unpacking. This approach is able to determine whether an executable is packed or not by comparing some structural features with a dataset composed only of not packed executables. If the executable under inspection presents a considerable deviation to what it is considered as usual (not packed executables), it is considered suspicious and is isolated for a further analysis. This method does not need updated data about packed executables, and thus, it reduces the efforts of labelling executables manually.

Summarising, our main contributions are: (i) we select a set of structural characteristics extracted from PE executables to determine whether a sample is packed or not and provide a relevance measure for each characteristic based on information gain, (ii) we propose an anomaly-detection-based architecture for packed executable filtering, by means of weighted comparison against a dataset composed of only not packed executables and (iii) we evaluate the method using three different deviation measures.

2 Structural Features of the Portable Executable Files

Given the conclusions obtained in previous work [16–18], we selected a set of 211 structural features from the PE executables. Some of the features were obtained directly from the PE file header while the rest are calculated values based on certain heuristics commonly used by the research community. Shafiq et al. [17]

used PE executable structural features were used to determine if an executable is benign or malicious but it was not considered if the executable was packed or not. Perdisci et al. [18] and later Farooq et al. [16] used some heuristics like entropy, or certain section characteristics to determine whether an executable is packed or not, as a previous step to a deeper analysis. In this paper, we combine both points of view, structural characteristics and heuristics, providing a statistical analysis to determine their true relevance for determining the packed state of an executable.

We consider that one of the main requisites of our anomaly detection system is speed, as it constitutes a filtering step for a heavier unpacking process. Therefore, we selected a set of features whose extraction does not require a significant processing time, and avoided techniques such as code disassembly, string extraction or n-gram analysis [18], which slow down the sample comparison.

Features can be divided into four main groups: 125 raw header characteristics [17], 33 section characteristics (i.e., number of sections that meet certain properties), 29 characteristics of the section containing the entry point (the section which will be executed first once the executable is loaded into memory) and, finally, 24 entropy values. We apply relevance weights to each feature based on Information Gain (IG) [19]. IG provides a ratio for each feature 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 and 1,000 not packed executables, and are useful not only to obtain a better distance rating among samples, but also to reduce the amount of selected features, given that only 151 of them have a non-zero IG value.

- **DOS header characteristics (31).** The first bytes of the PE file header correspond to the DOS executable header fields. IG results showed that these characteristics are not specially relevant, having a maximum IG value of 0.23, corresponding to a reserved field, which intuitively may not be a relevant field. 15 values range from 0.10 to 0.16, and the rest present a relevance bellow 0.10.
- **File header block (23).** This header block is present in both image files (.EXE) and object files. From a total of 23 characteristics, 14 have an IG value greater than 0, and only 2 of them have an IG value greater than 0.01: the number of sections (0.3112) and the time stamp (0.1618).
- **Optional Header Block (71).** This optional block is only present in image files and contains data about how the executable must be loaded into memory. 37 features have an IG value over 0, but the most relevant ones are: the address of entry point (0.5111), the Import Address Table (IAT) size (0.3832) and address (0.3733) (relative to the number of imported DLLs), the size of the code (0.3011), the base of the data (0.2817), the base of the code (0.2213),the major linker version (0.1996), checksum (0.1736), the size of initialized data (0.1661), the size of headers (0.1600), the size of relocation table (0.1283) and the size of image (0.1243).
- **Section characteristics (33).** From the 33 characteristics that conform this group, 22 have an IG value greater than 0. The most significant ones

are: the number of non-standard sections (0.7606), the number of executable sections (0.7127); the maximum raw data per virtual size ratio (0.5755) ($rawSize/virtualSize$, where $rawSize$ is defined as the section raw data size and $virtualSize$ is the section virtual size, both expressed in bytes), the number of readable and executable sections (0.5725) and the number of sections with a raw data per virtual size ratio lower than 1 (0.4842).
- **Section of entry point characteristics (29).** This group contains 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 11 have a significant relevance: the characteristics field in its raw state (0.9757), its availability to be written (0.9715), the raw data per virtual size ratio (0.9244), the virtual address (0.7386), whether is a pointer to raw data or not (0.6064), whether is a standard section or not (0.5203), the virtual size (0.4056), whether it contains initialized data (0.3721), the size of raw data (0.2958) and its availability to be executed (0.1575).
- **Entropy values (24).** We have selected 24 entropy values, commonly used in previous works [18], from which 22 have an IG value greater than 0, and 9 have a relevant IG value: max section entropy (0.8375), mean code section entropy (0.7656), mean section entropy (0.7359), file entropy (0.6955), entropy of the section of entry point (0.6756), mean data section entropy (0.5637), header entropy (0.1680), number of sections with an entropy value greater than 7.5 (0.7445), and number of sections with an entropy value between 7 and 7.5 (0.1059).

In this way, every feature is represented as a decimal value and then normalized, dividing each value by the maximum value for that feature in the whole dataset. This way, we can represent each executable as a vector of decimal values that range from 0 to 1. The final step is to apply the relevance obtained from IG, and it consists of multiplying each value in the normalized vector by its relevance.

3 Anomaly Detection

Through the features described in the previous section, our method represents unpacked executables as points in the feature space. When an executable is being inspected our method starts by computing the values of the point in the feature space. This point is then compared with the previously calculated points of the unpacked executables.

To this end, distance measures are required. In this study, we have used the following distance measures:

- **Manhattan Distance.** This distance between two points v and u is the sum of the lengths of the projections of the line segment between the points onto the coordinate axes: $d(x,i) = \sum_{i=0}^{n} |x_i - y_i|$ where x is the first point; y is the second point; and x_i and y_i are the ith component of first and second point, respectively.

– **Euclidean Distance.** This distance is the length of the line segment connecting two points. It is calculated as: $d(x, y) = \sum_{i=0}^{n} \sqrt{v_i^2 - u_i^2}$ where x is the first point; y is the second point; and x_i and y_i are the ith component of first and second point, respectively.
– **Cosine Similarity.** It is a measure of similarity between two vectors by finding the cosine of the angle between them [20]. Since we are measuring distance and not similarity we have used $1 - \mathit{Cosine\ Similarity}$ as a distance measure: $d(x, y) = 1 - \cos(\theta) = 1 - \frac{\boldsymbol{v} \cdot \boldsymbol{u}}{||\boldsymbol{v}|| \cdot ||\boldsymbol{u}||}$ where $\boldsymbol{v}$ is the vector from the origin of the feature space to the first point x, $\boldsymbol{u}$ is the vector from the origin of the feature space to the second point y, $\boldsymbol{v} \cdot \boldsymbol{u}$ is the inner product of $\boldsymbol{v}$ and $\boldsymbol{u}$. $||\boldsymbol{v}|| \cdot ||\boldsymbol{u}||$ is the cross product of $\boldsymbol{v}$ and $\boldsymbol{u}$. This distance ranges from 0 to 1, where 1 means that the two executables are completely different and 0 means that the executables are the same (i.e., the vectors are orthogonal between them).

By means of these measures, we are able to compute the deviation of an executable respect to a set of not packed executables. Since we have to compute this measure with the points representing not packed executables, a combination rule is required in order to obtain a final value of distance which considers every measure performed. To this end, our system employs 3 simple rules: (i) select the mean value, (ii) select the lowest distance value and (iii) select the highest value of the computed distances. In this way, when our method inspects an executable a final distance value is acquired, which will depend on both the distance measure and the combination rule.

4 Empirical Validation

To evaluate our anomaly-based packed executable detector, we collected a dataset comprising 500 not packed executables and 1,000 packed executables. The first one is composed of 250 benign executables and 250 malicious executables gathered from the VxHeavens [21] website. The packed dataset is composed of 500 benign executables and 500 malicious executables from VxHeavens [21]. To generate the packed dataset, we employed not packed executables and we packed them using 10 different packing tools with different configurations: Armadillo, ASProtect, FSG, MEW, PackMan, RLPack, SLV, Telock, Themida and UPX.

Then, using this dataset we performed a 5-fold cross-validation to divide the not packed dataset into 5 different divisions of 400 executables for representing normality and 100 for measuring deviations. In this way, each fold is composed of 400 not packed executables that will be used as representation of normality and 1,100 testing executables, from which 100 are not packed and 1,000 are packed.

Hereafter, we extracted their structural characteristics and employed the 3 different measures and the 3 different combination rules described in Section 3 to obtain a final deviation measure for each tested executable. For each measure and combination rule, we established 10 different thresholds to determine whether an executable is packed or not.

Table 1. Results for different combination rules and distance measures. The results in bold are the best for each combination rule and distance measure. Our method is able to detect more than 99 % of the packed executable while maintaining FPR lower than 1 %.

	$1 - Cosine\ Similarity$			$EuclideanDistance$			$ManhattanDistance$		
Combination	**Threshold**	**FNR**	**FPR**	**Threshold**	**FNR**	**FPR**	**Threshold**	**FNR**	**FPR**
Mean	0.05000	0.000	0.332	0.70000	0.000	0.816	1.70000	0.000	0.544
	0.08400	0.001	0.166	0.84000	0.001	0.288	2.17000	0.001	0.260
	0.11800	0.001	0.126	0.98000	0.001	0.172	2.64000	0.001	0.150
	0.15200	0.001	0.018	1.12000	0.001	0.130	3.11000	0.001	0.048
	0.18600	**0.001**	**0.012**	1.26000	0.001	0.014	3.58000	0.001	0.012
	0.22000	0.042	0.010	1.40000	0.001	0.010	**4.0500**	**0.002**	**0.010**
	0.25400	0.532	0.010	**1.54000**	**0.002**	**0.008**	4.52000	0.017	0.008
	0.28800	0.625	0.006	1.68000	0.096	0.006	4.9900	0.086	0.006
	0.32200	0.728	0.004	1.82000	0.298	0.006	5.4600	0.247	0.002
	0.35600	0.888	0.000	1.96000	0.568	0.000	5.93000	0.379	0.000
Maximum	0.36300	0.000	0.262	1.90000	0.000	0.768	5.90000	0.000	0.570
	0.38200	0.003	0.138	1.96000	0.000	0.594	6.20000	0.000	0.340
	0.40100	0.004	0.118	2.02000	0.000	0.232	6.50000	0.000	0.194
	0.42100	**0.020**	**0.108**	2.08000	0.00	0.050	6.80000	0.001	0.048
	0.43900	0.085	0.100	2.14000	0.001	0.024	7.10000	0.002	0.028
	0.45800	0.102	0.098	**2.20000**	**0.002**	**0.014**	**7.40000**	**0.008**	**0.018**
	0.47700	0.122	0.086	2.26000	0.004	0.012	7.70000	0.033	0.008
	0.49600	0.195	0.012	2.32000	0.020	0.008	8.00000	0.077	0.004
	0.51500	0.329	0.010	2.38000	0.061	0.008	8.30000	0.239	0.004
	0.53400	0.509	0.000	2.44000	0.146	0.000	8.60000	0.378	0.000
Minimum	0.00032	0.000	0.682	0.06000	0.000	0.736	0.06000	0.000	0.736
	0.01962	**0.003**	**0.012**	0.20000	0.001	0.396	0.20000	0.001	0.396
	0.03892	0.107	0.008	0.34000	0.001	0.106	0.34000	0.001	0.106
	0.05822	0.189	0.004	0.48000	0.001	0.030	0.48000	0.001	0.03
	0.07752	0.213	0.004	**0.62000**	**0.002**	**0.014**	**0.62000**	**0.002**	**0.014**
	0.09682	0.374	0.004	0.76000	0.032	0.006	0.76000	0.032	0.006
	0.11612	0.477	0.002	0.90000	0.054	0.004	0.90000	0.054	0.004
	0.13542	0.692	0.002	1.04000	0.163	0.004	1.04000	0.163	0.004
	0.15472	0.792	0.002	1.18000	0.262	0.002	1.18000	0.262	0.002
	0.17402	0.860	0.000	1.32000	0.386	0.000	1.32000	0.386	0.000

We evaluated accuracy by measuring False Negative Ratio (FNR) and False Positive Ratio (FPR). FNR is defined as: $FNR(\beta) = \frac{FN}{FN+TP}$ where TP is the number of packed executable cases correctly classified (true positives) and FN is the number of packed executable cases misclassified as not packed software (false negatives). FPR is defined as: $FPR(\alpha) = \frac{FP}{FP+TN}$ where FP is the number of not packed executables incorrectly detected as packed while TN is the number of not packed executables correctly classified.

Table 1 shows the obtained results. Euclidean and Manhattan distances, despite of consuming less processing time, have achieved better results than cosine-similarity-based distance for the tested thresholds. In particular, our anomaly-based packed executable detector is able to correctly detect more than 99 % of unknown packers while maintaining the rate of misclassified not packed executable lower than 1 %. As it can be observed, mean combination rule presents slightly better results both for FNR and FPR. These results show that this method is a valid pre-process step for a generic unpacking schema. Since the main limitation of these unpackers is their performance overhead, a packed executable detector like our anomaly-based method can improve their workload, acting as a filter for these systems.

5 Discussion and Conclusions

Like the previous work, our method is focused on executable pre-filtering, as an initial phase to decide whether to analyse samples on a generic unpacker or not. Our main contribution is the anomaly-detection-based approach employed for packed executable identification. In contrast to previous approaches, this method does not need previously labelled packed and not packed executables, as it measures the deviation of executables respect to normality (not packed executables). Moreover, as it does not use packed samples for comparison, it is independent of the packer used to protect the executables. Although anomaly detection systems tend to produce high false positive rates, our experimental results show very low values in all cases. This fact proofs the validity of our initial hypothesis.

Anyway, it presents some limitations that should be studied in further work. First, it cannot identify the packer nor the family of the packer used to protect the executable. Such information would help the malware analyst in the task of unpacking the executable. Sometimes, generic unpacking techniques are very time consuming or fail and it is easier to use specific unpacking routines, created for most commonly used packers.

Secondly, the features extracted can be modified by malware writers in order to bypass the filter. In the case of structural features, packers could 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 by using standard sections instead of not standard ones, or filling sections with padding data to unbalance byte frequency and obtain lower entropy values. What is more, our system is very dependant on heuristics due to the relevance values obtained from IG, making it vulnerable to such attacks.

Finally, it is important to consider efficiency and processing time. Our system compares each executable against a big dataset (400 vectors). Despite Euclidean and Manhattan distances are easy to compute, cosine distance and more complex distance measures such as Mahalanobis distance may take too much time to process every executable under analysis. For this reason, in further work we will emphasize on improving the system efficiency by clustering not packed vectors and reducing the whole dataset to a limited amount of samples.

Acknowledgements

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 Iker Pastor López for his help in the experimental configuration.

References

1. Kaspersky: Kaspersky security bulletin: Statistics 2008 (2008) Available online: `http://www.viruslist.com/en/analysis?pubid=204792052`.

2. 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`.
3. PEiD: PEiD webpage (2010) Available online: `http://www.peid.info/`.
4. Faster Universal Unpacker: (1999) Available online: `http://code.google.com/p/fuu/`.
5. Morgenstern, M., 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`.
6. Babar, K., Khalid, F.: Generic unpacking techniques. In: Proceedings of the 2^{nd} International Conference on Computer, Control and Communication (IC4), IEEE (2009) 1–6
7. Data Rescue: Universal PE Unpacker plug-in Available online: `http://www.datarescue.com/idabase/unpack_pe`.
8. Stewart, J.: Ollybone: Semi-automatic unpacking on ia-32. In: Proceedings of the 14^{th} DEF CON Hacking Conference. (2006)
9. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In: Proceedings of the 2006 Annual Computer Security Applications Conference (ACSAC). (2006) 289–300
10. Kang, M., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: Proceedings of the 2007 ACM workshop on Recurring malcode, ACM (2007) 46–53
11. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: Proceedings of the 2007 Annual Computer Security Applications Conference (ACSAC). (2007) 431–441
12. Yegneswaran, V., Saidi, H., Porras, P., Sharif, M., Mark, W.: Eureka: A framework for enabling static analysis on malware. Technical report, Technical Report SRI-CSL-08-01 (2008)
13. Danielescu, A.: Anti-debugging and anti-emulation techniques. CodeBreakers Journal **5**(1) (2008) Available online: `http://www.codebreakers-journal.com/`.
14. Cesare, S.: Linux anti-debugging techniques, fooling the debugger (1999) Available online: `http://vx.netlux.org/lib/vsc04.html`.
15. Julus, L.: Anti-debugging in WIN32 (1999) Available online: `http://vx.netlux.org/lib/vlj05.html`.
16. Farooq, M.: 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), Springer-Verlag (2009) 121–141
17. Shafiq, M., Tabish, S., Farooq, M.: PE-Probe: Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables. In: Proceedings of the Virus Bulletin Conference (VB). (2009) 29–33
18. Perdisci, R., Lanzi, A., 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). (2008) 301–310
19. Kent, J.: Information gain and a general measure of correlation. Biometrika **70**(1) (1983) 163–173
20. Tata, S., Patel, J.: Estimating the Selectivity of tf-idf based Cosine Similarity Predicates. SIGMOD Record **36**(2) (2007) 75–80
21. VX Heavens: Available online: `http://vx.netlux.org/`.