Semi-supervised Learning for Packed Executable Detection

Xabier Ugarte-Pedrero[†], Igor Santos[†], Pablo G. Bringas[†], Mikel Gastesi[§], José Miguel Esparza[§]

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

[§]S21sec,
Parque Empresarial La Muga, 11, 31160
Orcoyen, Spain
Email: {mgastesi,jesparza}@s21sec.com

***Abstract*—The term malware is coined to name any software with malicious intentions. One of the methods malware writers use for hiding their creations is executable packing. Packing consists of encrypting or hiding the real code of the executable in such a way that it is decrypted or unhidden in its execution. Widespread solutions to this issue first try to identify the packer used and next apply the corresponding unpacking routine for each packing algorithm. As it happens with malware obfuscations, this approach fails to detect new and custom packers. Generic unpacking is a technique that has been proposed to solve this issue. These methods usually execute the binary in a contained environment or sandbox to retrieve the real code of the packed executable. Because these approaches incur in a high performance overhead, a filter step is required to determine whether an executable is packed or not. Supervised machine-learning approaches have been proposed to handle this filtering step. However, the usefulness of supervised learning is far to be complete because it requires a high amount of packed and not packed executables to be identified and labelled previously. In this paper, we propose a new method for packed executable detection that adopts a well-known semi-supervised learning approach to reduce the labelling requirements of completely supervised approaches. We performed an empirical validation demonstrating that the labelling efforts are lower than when supervised learning is used while the system maintains high accuracy rates.**

I. Introduction

The term malware is devoted to name any computer program that has been intentionally designed to damage computers or networks. Malware writers usually employ obfuscation techniques in order to hide the behaviour of their malicious designs. One technique malware writers usually employ is executable packing. Packing consists of cyphering or compressing the actual malicious code with the intention of hiding the real intentions of their creations. Usually, packed software has a decryption routine that retrieves the actual code from a data section of the memory where it remains cyphered and then executes it. This technique is highly employed and, according to [1], up to the 80% of the detected malware is packed.

Since malware writers commonly employ known packers, packer signature scanning has been adopted by the anti-malware industry in order to solve this issue (e.g., PEID [2] and Faster Universal Unpacker (FUU) [3]). 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 [4], 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 [5]. The execution is traced and stopped when certain events occur. 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 [6] and OllyBonE [7]). Notwithstanding, concrete heuristics cannot be applied to all the packers in the wild, as all of them work in very different manners. Other approaches for generic dynamic unpacking have been proposed that are not highly based on heuristics such as PolyUnpack [8] Renovo [9], OmniUnpack [10] or Eureka [11]. However, 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 [12].

Another common approach is using the structural information of the PE executables to train supervised machine-learning classifiers to determine if the sample under analysis is packed or if it is suspicious of containing malicious code (e.g., PE-Miner [13], PE-Probe [14] and Perdisci et al. [15]). These approaches use this method as a filtering step previous to a dynamic unpacking process computationally more expensive.

However, these supervised machine-learning classifiers require a high number of labelled executables for each of the

classes. Sometimes, we can omit one class from labelling (e.g., anomaly detection systems for intrusion detection [16]). 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 machine-learning technique specially useful when a limited amount of labelled data is available for each file class. These techniques generate a supervised classifier based on labelled data and predict the label for every unlabelled instance. The instances whose classes have been predicted surpassing a certain threshold of confidence are added to the labelled dataset. The process is repeated until certain conditions are satisfied (a commonly used criterion is the maximum likelihood found by the expectation-maximisation technique). These approaches enhance the accuracy of fully unsupervised methods (i.e., no labels within the dataset) [17].

Given this background, we propose here an approach that employs a semi-supervised learning technique for the detection of unknown malware. In particular, we utilise the method *Learning with Local and Global Consistency* (LLGC) [18] able to learn from both labelled and unlabelled data and capable of providing a *smooth* solution with respect to the intrinsic structure displayed by both labelled and unlabelled instances. For the representation of executables, we use structural information of the executables combining structural features, fields from the PE file header, and heuristics commonly used in a previous work [19], which presented an anomaly detector for packed executable detection. Summarising, our main findings in this paper are: (i) we describe how to adopt LLGC for packed executable detection, (ii) we empirically determine the optimal number of labelled instances and (iii) we evaluated how this parameter affects the final accuracy of the model and we demonstrate that labelling efforts can be reduced in the malware detection industry, while still maintaining a high rate of accuracy.

II. Structural Features of the Portable Executable Files

Features can be divided into four main groups: 125 raw header characteristics [14], 31 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 have measured the relevance of each characteristic based on Information Gain (IG) [20]. 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 an non-zero IG value.

- **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.
- **File header block**. This header block is present in both image (exe files), and object files. 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) and number of sections (0.16112).
- **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) and size of initialized data (0.19152).
- **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).
- **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), the 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).
- **Entropy values**. We have selected 24 entropy values,

commonly used in previous works [21], 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), number of sections with entropy in the range between 7.5 and 8 (0.32336), header entropy (0.19541), number of sections with entropy in the range between 7 and 7.5 (0.10074).

III. Overview of LLGC

Learning with Local and Global Consistency (LLGC) [18] 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 [18].

Formally, the algorithm is stated as follows. Let $\mathcal{X} = \{x_1, x_2, ..., x_{\ell-1}, x_\ell\} \subset \mathbb{R}^m$ be the set composed of the data instances and $\mathcal{L} = \{1, ..., c\}$ the set of labels (in our case, this set comprises two classes: packed and not packed software) and $x_u (\ell + 1 \leq u \leq n)$ the unlabelled instances. The goal of LLGC (and every semi-supervised algorithm) is to predict the class of the unlabelled instances. $\mathcal{F}$ is the set of $n \times c$ matrices with non-negative entries, composed of matrices $F = [F_1^T, ..., F_n^T]^T$ that match to the classification on the dataset $\mathcal{X}$ of each instance x_i. with the label assigned by $y_i = \text{argmax}_{j \leq c} F_{i,j}$. F can be defined as a vectorial function such as $F : \mathcal{X} \rightarrow \mathbb{R}^c$ to assign a vector F_i to the instances x_i. Y is an $n \times c$ matrix such as $Y \in F$ with $Y_{i,j} = 1$ when x_i is labelled as $y_i = j$ and $Y_{i,j} = 0$ otherwise. Considering this, the LLGC algorithm performs as follows:

if $i \neq j$ *and* $W_{i,i} = 0$ **then**
  Form the affinity matrix W defined by $W_{i,j} = \exp\left(\frac{-||x_i - x_j||^2}{2 \cdot \sigma^2}\right)$;
Generate the matrix $S = D^{-1/2} \cdot W \cdot D^{-1/2}$ where D is the diagonal matrix with its (i, i) element equal to the sum of the i-th row of W;
while $\neg$ *Convergence* **do**
  $F(t+1) = \alpha \cdot S \cdot F(t) + (1 - \alpha) \cdot Y$ where α is in the range $(0, 1)$;
F^* is the limit of the sequence $\{F(t)\}$;
Label each point x_i as $\text{argmax}_{j \leq c} F^*_{i,j}$;

Fig. 1. LLGC algorithm.

The algorithm first defines a pairwise relationship W on the dataset $\mathcal{X}$ setting the diagonal elements to zero. Suppose that a graph $G = (V, E)$ is defined within $\mathcal{X}$, where the vertex set V is equal to $\mathcal{X}$ and the edge set $\mathcal{E}$ is weighted by the values in W. Next, the algorithm normalises symmetrically the matrix W of G. This step is mandatory to assure the convergence of the iteration. During each iteration each instance receives the information from its nearby instances while it keeps its initial information. The parameter α denotes the relative amount of information from the nearest instances and the initial class information of each instance. The information is spread symmetrically because S is a symmetric matrix. Finally, the algorithm sets the class of each unlabelled specimen to the class of which it has received most information during the iteration process.

IV. Empirical Validation

To evaluate our semi-supervised packed executable detector, we collected a dataset comprising 1,000 not packed executables and 1,000 packed executables. The first set contained 500 benign executables (obtained from a clean Microsoft Windows XP installation) and 500 not packed malicious executables gathered from the website VxHeavens [22]. The packed dataset is composed of 500 executables manually packed and 500 variants of the malware family 'Zeus' protected by custom packers (packing and unpacking routines are custom-made by the malware programmer, and thus, unknown for signature based packer detectors). To generate the manually 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. For the second dataset we employed 500 variants of the 'Zeus' family that were not detected by PEiD [2] (updated to the last signature database).

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 LLGC in detecting packed executables. We used the LLGC implementation provided by the *Semi-Supervised Learning and Collective Classification* package[1] for the well-known machine-learning tool WEKA [23]. Specifically, we configured it with a transductive stochastic matrix W [18] and we employed the Euclidean distance with 5 nearest neighbours.

In particular, we measured the *True Positive Ratio* (TPR), i.e., the number of packed executables correctly detected divided by the total number of malware files: $TPR = TP/(TP + FN)$, where TP is the number of malware cases correctly classified (true positives) and FN is the number of packed executables misclassified as not packed software (false negatives). We also measured the *False Positive Ratio* (FPR), i.e., the number of not packed executables misclassified as packed divided by the total number of not packed files: $FPR = FP/(FP + TN)$, where FP is the number of not packed software cases 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

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

hits of the classifier divided by the number of instances in the whole dataset: $Accuracy = (TP+TN)/(TP+FP+TP+TN)$. Besides, we measured the *Area Under the ROC Curve* (AUC) that establishes the relation between false negatives and false positives [24]. The ROC curve is obtained by plotting the TPR against the FPR.

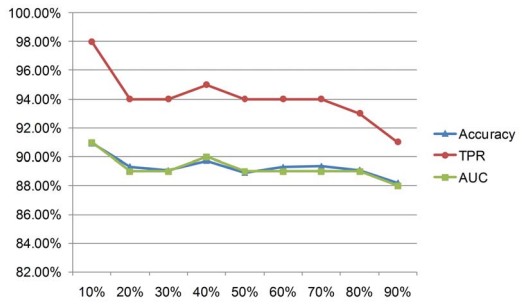

Fig. 2. Accuracy, TPR and AUC results. The X axis represents the percentage of labelled instances. The precision of the model slightly decreases along with the size of the labelled set.

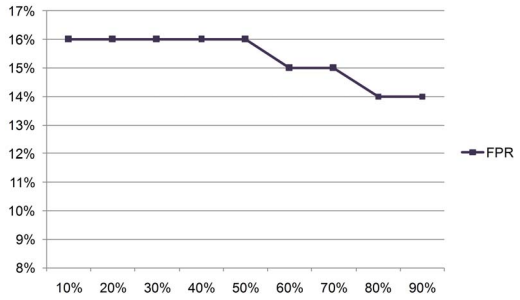

Fig. 3. FPR results. The X axis represents the percentage of labelled instances. The FPR decreases as the size of the labelled set increases.

Fig. 2 and Fig. 3 show the obtained results. In particular, we found out that the greater the size of the labelled instances set the worse the results obtained. Specifically, the best overall results were obtained with a training set containing 10% of labelled instances: a 90.95% of accuracy. The results decrease when the size of labelled dataset is increased but not in a significant way, revealing that the 10% of the dataset is sufficient to build a correct representation of a packed executable structure. Regarding the FPR, the results get better when the size of the labelled dataset increases. However, the decrease of the false positive ratio is also very low and we cannot raise significant conclusions.

V. Discussion and Conclusions

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 LLGC semi-supervised learning approach employed for packed executable identification.

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 lower than the fully supervised approaches (using our characteristics with supervised machine learning we can obtain more than 99% of accuracy), but it only requires a 10% of the labelled executables. Table I compares the work of Perdisci et al. [15], [25] with our semi-supervised approach. Obviously, the approach of Perdisci et al., using fully supervised methods outperforms our method. However, our semi-supervised method requires less executables to be labelled: a 10% of the whole dataset in order to guarantee an accuracy higher than 90%. Since the dataset of Perdisci et al., is not the same as ours, we have employed the same features described in Section II to run a premilinary test of supervised learning with our characteristics resulting on better results than the previous approach of Perdisci et al. and also, our semi-supervised method.

Since our main goal was to provide high accuracy rates but reduce the required number of labelled executables, we believe that our method is capable of doing so. Nevertheless, there are several limitations that should be discussed:

First, our approach cannot identify neither the packer nor the family of the packer used to protect the executable. This type of information can help the malware analyst in the task of both unpacking the executable and creating 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 commonly used packers.

Second, 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.

The last limitation of our approach is also applicable to other methods for detecting packed executables. Indeed, in our approach we utilise every feature that has been used in previous work [13], [14], [15], but we have added several characteristics like different entropy values, ratio between data and code sections and so on. However, these features are heuristics that are employed basically because the common packers work that way. Anyhow, new packing techniques like virtualization [26], [27], [28], 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.

Although this ability of malware packers to surpass these heuristics is a real problem, we have to notice that the majority of the packed executables are packed with known packers like UPX. Besides, there is an increasing number of malicious executables that are packed with custom packers. In our

TABLE I
COMPARISON WITH PREVIOUS WORK IN PACKED DETECTION.

Approach	Accuracy	TPR	FPR	AUC
Perdisci et al. [15]	0.994	0.996	0.008	0.997
Our approach using 10% of the dataset (LLGC)	0.909	0.980	0.160	0.910
Our features using Random Forest (Supervised)	0.993	0.990	0.004	1.000

validation, we have included a big amount 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. Our approach was able to detect the majority of these custom packers and, therefore, we consider that the results indicate that some of the features may be evaded but the evasion of all of them whilst maintaining the whole functionally is hard. We would like to test this possibility in further work.

Future work is oriented in three main ways. First, we plan to extend this approach with an specific packer detector capable of discriminating between executables packed with a known packer and the ones protected 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 LLGC. Finally, we plan to investigate the attacks these filters can suffer.

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 S21Sec for the Zeus malware family samples provided in order to set up the experimental dataset.

REFERENCES

[1] 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. [Online]. Available: http://www.mcafee.com/us/resources/white-papers/wp-good-bad-unknown.pdf

[2] PEiD, "PEiD webpage," 2010, available online: http://www.peid.info/.

[3] Faster Universal Unpacker, 1999, available online: http://code.google.com/p/fuu/. [Online]. Available: http://code.google.com/p/fuu/

[4] 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.

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

[6] Data Rescue, "Universal PE Unpacker plug-in," available online: http://www.datarescue.com/idabase/unpack_pe.

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

[8] 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)*, 2006, pp. 289–300.

[9] 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*. ACM, 2007, pp. 46–53.

[10] 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)*, 2007, pp. 431–441.

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

[12] A. Danielescu, "Anti-debugging and anti-emulation techniques," *CodeBreakers Journal*, vol. 5, no. 1, 2008, available online: http://www.codebreakers-journal.com/.

[13] 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)*. Springer-Verlag, 2009, pp. 121–141.

[14] 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.

[15] 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)*, 2008, pp. 301–310.

[16] R. Perdisci, G. Gu, and W. Lee, "Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems," in *Proceedings of 6^{th} International Conference on Data Mining (ICDM)*. IEEE, 2007, pp. 488–498.

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

[18] D. Zhou, O. Bousquet, T. Lal, J. Weston, and B. Schölkopf, "Learning with local and global consistency," in *Advances in Neural Information Processing Systems 16: Proceedings of the 2003 Conference*, 2004, pp. 595–602.

[19] X. Ugarte-Pedrero, I. Santos, and P. G. Bringas, "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)*, 2011, pp. 50–57.

[20] J. Kent, "Information gain and a general measure of correlation," *Biometrika*, vol. 70, no. 1, pp. 163–173, 1983.

[21] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 40–45, 2007.

[22] VX Heavens, available online: http://vx.netlux.org/. [Online]. Available: http://vx.netlux.org/

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

[24] 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*, vol. 35, no. 2, pp. 183–193, 2009.

[25] R. Perdisci, A. Lanzi, and W. Lee, "Classification of packed executables for accurate computer virus detection," *Pattern Recognition Letters*, vol. 29, no. 14, pp. 1941–1946, 2008.

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

[27] ——, "Rotalumè: A Tool for Automatic Reverse Engineering of Malware Emulators," 2009.

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