Extended Abstract: Evading Packing Detection: Breaking Heuristic-Based Static Detectors

Alexandre D'Hondt[0000−0001−6160−8624], Charles Henry Bertrand Van Ouytsel[0000−0001−5720−6569], and Axel Legay[0000−0003−2287−8925]

Université Catholique de Louvain, Rue Archimede 1, Louvain-la-Neuve, Belgium
{alexandre.dhondt,charles-henry.bertrand,axel.legay}@uclouvain.be

Abstract. Nowadays, executable packing remains an open issue in its detection especially when it comes to static analysis. Packing is significantly used in malware to hide malicious code from detection systems. These last years, many studies about static packing detection addressed this problem with heuristics and machine learning, considering different ad hoc techniques, algorithms and feature sets but very few addressed it from the adversarial point of view, that is, how to fool heuristics by altering samples with targeted modifications. The objective of this work is to study to what extent it is easy to evade detection by open source static detectors that are commonly used by the community by applying alterations on packed samples, which require only slight adaptations of the related packers, resulting in evasion. An adversarial setting from the problem-space perspective is addressed by using realistic modifications of binary samples that target common significant features. For this purpose, alterations and datasets are composed and static detection is applied using the experimental toolkit Packing Box. Results of alterations are shown, in terms of information gain of features and accuracy of detection, on open source static packing detectors. Finally, their significant effects are highlighted and their effectiveness is evaluated.

Keywords: executable packing · packer detection · static analysis · adversarial examples · experimental toolkit

1 Introduction

Executable packing involves transformations that modify a binary file that preserve its semantics while having its layout changed at rest. Especially when compressing or encrypting parts of a binary, packing is useful for evading detection by static techniques like signature matching as used in an antivirus. Static techniques are those that only use information available at rest, that is, without executing the binary. On the contrary of dynamic analysis that leverages execution of the target binary to collect information at runtime, static analysis offers the major advantage of being much faster. The main challenge thus remains *performance*, that is how to leverage as much information as possible, knowing that a static approach will capture less information than a dynamic one,

while keeping processing time as low as possible, which is best achieved with a static technique. Therefore, approaches and methods in the field of static analysis continued to evolve since the early 2000's, increasingly focusing on machine learning, yielding various features sets. In this scope, a recent study on packing detection with machine learning by Bertrand Van Ouytsel *et al.* [3] highlighted a subset of *significant features* based on the features set of Biondi *et al.* [4].

On the one hand, many studies [14] addressed malware detection evasion, altering characteristics of malicious samples for targeting particular detection heuristics. In this setting, modifications are applied directly on samples and not on features. The underlying challenge is that feature modifications can usually not be mapped back to samples [14], which limits options for adversarial attacks. On the other hand, packing is widely used in malware and is sometimes even left as an open issue in malware analysis. According to statistics from Aghkhani *et al.* on a large-scale real-world dataset from 2017 and 2018, more than 60% of malicious samples were packed. As none of the currently malware-focused attacks specifically addressed the problem of packing yet, there is room for improvement in this particular field. While some alterations applicable to malware detection evasion apply to packing evasion too, they require further analysis and thorough experimentation on additional specific characteristics and dedicated detectors.

In this paper, focus is on attacking packing static detectors found in the wild, especially popular ones found on GitHub and commonly used by the community, and on finding sample alterations so that detection can be influenced and weak features can be identified. By purpose, the scope is restricted to targeting such detectors and not those based on machine learning as it will be the topic of another study. The objective is to answer the following research questions:

Q1 Which *alterations* can be defined *to impact* common *significant features* that popular static packing detectors rely on ?
Q2 Which *features* are *most vulnerable* to alterations on packed samples ?
Q3 Which static packing *detectors* are *most vulnerable* to alterations ?

The contribution of this paper is to propose an extension of the Packing Box [9] that allows to define alterations and to show with some experiments based on five carefully crafted alterations to what extent features are impacted and how much accuracy of open source detectors can be decreased. It is shown that three out of these five alterations have a significant impact on some popular packing detectors and that weak features can be identified through this process.

The remainder of this paper first presents, in Section 2, background on packing and its related state-of-the-art static detection techniques. It also addresses some related works on adversarial attacks with some alterations that were already proposed. In Section 3, the Packing Box is introduced and improvements made for supporting alterations are presented, arguing on those created for targeting features used by in-scope detectors. Then in Section 4, the threat model of the adversarial setting is stated and some experiments are conducted based on datasets of packed samples, focusing attention on the accuracy of detectors to evaluate evasion through our alterations. Section 5 finally concludes on what could be achieved and what remains to be further developed and studied.

2 Background

This section presents important concepts for addressing static packing detection, including techniques based on heuristics and machine learning, but also adversarial attacks and their alterations and how it applies to executable packing.

Packing is defined as a set of transformations that tampers with the layout of a binary file at rest while preserving its working at runtime. In other words, the process of packing aims to refactor the syntax of the target binary without modifying its semantics. Such *transformations* range from simple compression or encryption to mutation of code or virtualization. It can also be slight modifications for anti-reverse engineering. It is used for legitimate purpose, e.g. for software license protection [13], as well as for malware, i.e. for evasion from detection [1]. It is worth being noted that this definition extends the traditional one commonly found in many studies that only consider compression and encryption.

Detection techniques range from static to dynamic analysis but can be a mix of both as well. Static techniques pertain to analysing a binary without running it, that is, only considering information at rest like structural features [7] or headers [16]. On the contrary, dynamic methods require execution for collecting information at runtime such as entropy changes [2] or write operations [18] in memory. As a consequence of the inevitable execution overhead and what can be inspected at rest or at runtime, static techniques always capture less information than dynamic ones. The greatest challenge is then to identify the best information that can be used for reliable detection. Many approaches have been addressed since the early 2000's including entropy-based methods [15], heuristics [11, 16] or signature matching [12, 17] and machine learning. Bertrand Van Ouytsel *et al.* [3] identified a set of *significant features* specifically for packing detection, based on the features set of Biondi *et al.* [4]. Their methodology relies on SHAP values, a metric that allows to evaluate the robustness of features.

Adversarial attacks are classified by their *threat model* that describes the attack surface and characterizes the attacker. *Attack surface* relates to the steps where an attack can take place. For machine learning, it corresponds to the steps of the learning pipeline. Poisoning typically involves the training step while evasion targets the test phase. For characterizing the attacker, Carlini *et al.* [5] proposes a comprehensive framework that defines three components. First, attacker's *goal* specifies if the attack is targeted or untargeted. When the attack is targeted, the attack is successful when the predicted class of the detector is a specific class of interest for the attacker. When untargeted, the predicted class can be any other than the original one. Second, the *capabilities* state what the attacker can do. In a realistic scenario, attacker's ability to modify examples is often limited and is typically defined by the space in which perturbations can be achieved. Third, attacker's *knowledge* is also a limiting factor. Two settings can be distinguished: white-box and black-box. In the white-box setting, the model is fully known while in the black-box setting, it is assumed that the attacker has no access to the model's inner workings so that only the output can be seen.

Two important spaces related to attacker's capabilities can be noted: the *feature* and *problem space*. Feature-space attacks are possible when a feature vector

can be input directly to the model. The attacker can then apply a perturbation vector and analyze its effect on classification. In problem-space attacks, perturbations, called *alterations*, are applied directly to input samples. The major issue with feature-space attacks is that feature mappings are often not bijective, meaning that they do not capture the entire information from the problem space. In such cases, perturbation vectors from the feature space cannot be propagated back to the problem space [14]. Therefore, modified feature values could possibly not be mapped back to a realistic executable. For this reason, we consider only the problem space with *alterations* that are designed not to break executable samples' working. Moreover, as the focus is on common static packing detectors based on signatures or heuristics, there is no pipeline in which a feature space can be addressed for applying perturbations. Heuristics are treated as black-box models on which only the test phase is accessible.

Many studies addressed alterations of samples from the malware perspective, often focusing on a single characteristic of binaries like API imports [8,10,19] or appending bytes to the overlay [8,10,19], and targeting a specific type of malware detector. These characteristics that can be altered without breaking executables' working include headers [8,19], sections [8,19] or the signature [10,19].

In the next sections, focus is on the Portable Executable (PE) format of Windows, on its related static packing detectors integrated in the Packing Box and on a threat model relying on the problem space and evasion in a black-box setting. Section 3 addresses the material from the Packing Box required for the experiments of Section 4, including alterations targeting the set of significant features of Bertrand Van Ouytsel *et al.* [3]. Section 4 starts by describing the complete threat model as part of the experiments methodology.

3 Framework

This section presents the essentials of the Packing Box and how it was tuned to define alterations for our experiments. It explains functionalities required to further address our experiments of Section 4.

Packing Box [9] is the reference toolkit for managing experiments. It is an Ubuntu-based Docker image that contains a comprehensive toolset for controlling the machine learning pipeline from dataset generation to model training and testing. But more importantly, it integrates a series of detectors presented and evaluated on the basis of their performance by D'Hondt *et al.* [9], including Bintropy, Detect It Easy (DIE), Manalyze, PEiD, PEPack, PyPackerDetect, REMINDer and Retargetable Decompiler (RetDec). It aims to support multiple executable formats including Portable Executable (PE) from Windows. Given the wide variety of sources for PE samples and tools, the scope of this paper is limited to this format. Packing Box also provides the ability to manage experiments, datasets and models for enhancing repeatability and shareability of results with peer researchers. When creating an experiment, a dedicated workspace is set up for storing datasets and models but also for embedding resources such as features declarations and specific data such as lists of standard or packer section

names. Packing Box defines items including packers, detectors, ML algorithms, features or *alterations* using *YAML*[1] *files*, which is a convenient and readable format for declaring aspects such as installation steps as well as for documentation for including comments or source and reference hyperlinks. In the scope of this study, we use experiments to manipulate and sort datasets and embed alterations definitions and scripts for evaluating the accuracy of detectors.

Alterations are applied in the problem-space, that is before feature computation, and aim to modify samples so that an impact can be observed at the test phase of the pipeline. Note that, as the scope of this paper is limited to black-box static packing detectors relying on signatures and heuristics, attack options are limited to the test phase anyway. Alterations are defined to target multiple kinds of features used in common heuristics, including section characteristics, e.g. name or flags, entropy and signatures, typically from the Entry Point.

A01 *rename packer sections* – This modifies sections with common packer or empty section names with standard names, targeting the Entry Point (EP) section first then other sections and choosing replacement names from the 6 most common standard names, including *.text*, *.data* or even *.bss*, then random standard names among a list composed from the PE format specification. It ensures that no section name is repeated. Targeted features: *number of standard sections* as common packer section names get replaced with standard ones.

A02 *move EP to new low-entropy code section* – This moves the EP to a new code section with a standard section name that plays the role of *trampoline* by defining a jump to the Original EP and it fills the section up to 64 bytes with a low-entropy pattern. Targeted features: whether the *EP is not in a standard section* as many packers use a code section with a non-standard name, *bytes after the EP* as then having a new series of bytes containing the trampoline instructions and a low-entropy pattern will not match signatures anymore.

A03 *fill sections with low-entropy pattern* – This fills sections with a pattern of 3 bytes when the virtual size, that is the size as expanded in memory at runtime, is higher than the size of section's data, that is the size on disk. This aims to lower entropy while not breaking the executable as sections normally gets expanded to their virtual size with zeros anyway. Targeted features: *byte entropy of the entire PE file* and *byte entropy of the code section*.

A04 *add low-entropy code section* – This adds a new code section with a size of 10% of the overall file size, ensuring it holds a low entropy by using a pattern of 3 bytes. Targeted features: *byte entropy of the entire PE file* as filling in the new section with 10% of the overall size with a low-entropy pattern necessarily impacts the overall entropy significantly.

A05 *add 20 common API imports* – This adds 20 common API imports to the Import Address Table (IAT), ensuring that there is no duplicate. Targeted features: *ratio of malicious API calls imported* as it will decrease as a result of adding common API imports.

[1] Yet Another Markup Language

The aforementioned alterations thus answer Q1, focusing on significant features from the current literature. The next section addresses the experimental setting and how these alterations were tested.

4 Experiments

This section first presents the setting and methodology used to conduct experiments aimed to answer Q2 and Q3. It then presents the results of the approach in terms of impact of alterations defined in Section 3 on the information gain of features and on the accuracy of in-scope detectors.

4.1 Methodology

The *environment* is set up by building Packing Box's Docker image and running its command-line interface. The reference dataset is `dataset-packed-pe`, available on GitHub, and is an alternative version of PackingData from Choi *et al.* [6] augmented with subsets of samples we packed with 5 new packers. It consists of subsets from 115 to 150 samples for each of the following packers: Alienyze, Amber, ASPack, BeroEXEPacker, Enigma VirtualBox, Eronana's Packer, Exe32pack, EXpressor, FSG, JDPack, MEW, Molebox, MPRESS, Neolite, NSPack, Packman, PECompact, PEtite, RLPack, Telock, Themida, UPX, WinUPack, Yoda's Crypter and Yoda's Protector.

The *threat model* is defined according to the framework of Carlini *et al.* [5]. In-scope detectors are those from the Packing Box presented in Section 3. Their attack surface is limited to the test phase, meaning that tools can be run and their output inspected ; this scenario relates to evasion. Attacker's goal is targeted towards fooling the detector into concluding that an initially packed sample is not packed. Attacker's capabilities are limited to tampering with packed samples and feeding them to the detectors ; the attacker's perspective is limited to the problem space by applying realistic perturbations on samples. Finally, attacker's knowledge is considered black-box, even though selected detectors are open source and their heuristics could therefore be reverse-engineered.

The *methodology* is divided into two parts, one for focusing on the impact of alterations on features and another on the impact on detectors, for the purpose of correlating results. Both involve three stages: first, a preparation of the baseline dataset (BL) and one dataset for each alteration (A0X), then second, application of alterations on these, and third, representation and analysis of results.

4.2 Impact of alterations on features

This experiment uses six datasets: BL for the reference dataset and A0X for this dataset altered with each alteration. Information Gain (IG) is measured using the Mutual Information (MI) between the feature values and the target label. As only packed samples are considered, this metric relies on the labels of the 25 different packers present in the baseline dataset (BL). Example in Figure 1

shows values for feature *byte 0 after the EP* for 5 packers out of 25. On the left side, 4 distinct values are present in BL. On the right side, there is only 1 value left as **A02** moves the EP to a new section starting with a trampoline instruction. This yields a high value of MI as depicted in Figure 2.

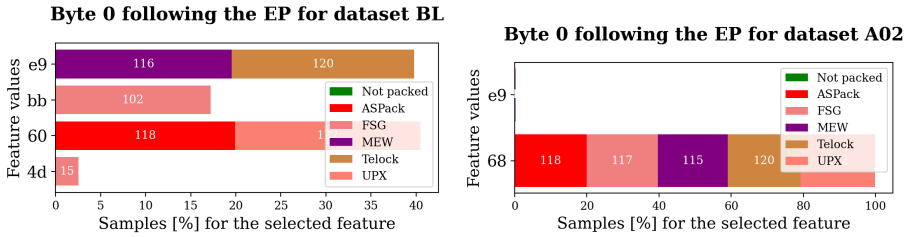

Fig. 1. Values of feature *byte 0 after EP* for BL and A02

In Figure 2, values are normalized based on the overall highest value. It results in a heat map that highlights most significant effects of alterations.

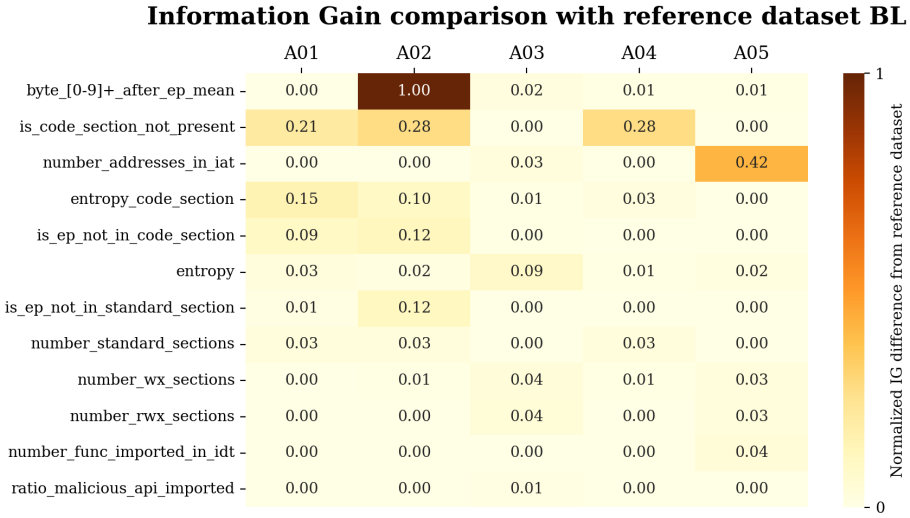

Fig. 2. Effect of practical alterations on significant features

A01 mostly impacts features related to the code section since it renames packer sections to standard names starting with *.text*. However, the number of standard sections is not that impacted as many packers use standard names combined with only one or two packer-specific names, hence the resulting number

of standard sections does not significantly vary. **A02** mostly affects bytes after the EP as expected. As it creates a new section with trampoline instructions to the Original EP, it completely changes the 64 bytes after the EP. This alteration also affects features related to the absence of code section and those related to the EP as the new code section has a standard name. **A03** slightly impacts *entropy* where binary samples have sections for which the raw size significantly differs from the virtual size. This is especially the case for packers like UPX that hold a first section of length zero that gets expanded in memory at runtime to a virtual size aimed for uncompressed data. As only a few packers work this way, the overall impact on Information Gain is not that significant. **A04** mostly impacts the feature about the presence of a code section, as this alteration adds a new code section where some packed samples do not have one. As it aims to add 10% size of low-entropy bytes, it does not significantly impact entropy-related features, even though it can influence the overall entropy just enough to fool related heuristics. **A05** mostly impacts the number of addresses of imported functions in the IAT as expected. This has no major impact on other features.

These results highlight a few sensitive features, answering Q2, among the most significant identified by Bertrand Van Ouytsel *et al.* [3]. These include *bytes after the EP* trivially exploited by moving the EP, features based on code section *.text* or entropy-based features exploited by injecting just enough low-entropy bytes sequence to bring entropy beneath heuristics' thresholds.

4.3 Impact of alterations on detectors

This experiment uses each per-packer subset from the reference dataset. Detection is applied for each detector on each subset and accuracy is aggregated with a weighted sum. Table 1 shows the results of mass-detection with unchanged results in gray, variations from zero to five percents in olive, variations of more than five percents in orange and significant variations yielding accuracy of less than two percents in red.

Detector	BL	A01	A02	A03	A04	A05
Bintropy	59.46%	59.76%	59.76%	**1.88%**	**0.01%**	56.83%
DIE	86.13%	77.52%	27.71%	84.66%	82.42%	83.50%
Manalyze	40.71%	**1.85%**	40.71%	44.44%	40.71%	40.71%
PEiD	79.69%	79.69%	**0.00%**	79.69%	77.78%	79.73%
PEPack	81.19%	81.19%	44.18%	81.09%	79.24%	81.19%
PyPackerDetect	95.03%	80.28%	95.87%	86.99%	95.03%	95.03%
REMINDer	42.07%	42.07%	**0.00%**	24.15%	42.07%	42.07%
RetDec	70.60%	70.60%	44.12%	70.57%	70.60%	66.99%

Table 1: Accuracy of detectors for BL and A0X

A01 is effective against Manalyze, which indicates that this detector relies solely on packer section names. The remaining 1.85% accuracy comes from NSPack samples that could not be altered without error. It also disturbs detection by DIE and PyPackerDetect to a lesser extent, meaning that these detectors

solely rely on section names for some specific packers. **A02** is highly effective against PEiD, indicating that this tool solely relies on signatures with many of them made to match from the EP. It is also effective against REMINDer which, on the contrary of PEiD, relies on a simple heuristic based on entropy of the section containing the EP. With this alteration, this entropy falls beneath the threshold of REMINDer as 64 low-entropy bytes are appended after the trampoline. This also shows significant effects on DIE, PEPack and RetDec. **A03** breaks detection by Bintropy and significantly disturbs REMINDer as both rely on entropy. In the case of Bintropy, even though many packers do not use a null-size section for expansion in memory for decompression, it is frequent that sections have differences between the raw and virtual size, causing enough drop in entropy to break Bintropy's heuristic. In the case of REMINDer, only the EP section matters, hence the lower efficiency. **A04** has limited impact on the detectors except Bintropy. This shows that none of them rely on entropy except REMINDer which, this time, is not impacted as this alteration does not touch the EP section. The low-entropy pattern is enough to break Bintropy's heuristic, the remaining 0.01% coming from corrupted samples. **A05** causes only slight perturbations, showing that detectors are not sensitive to API imports.

These results, answering Q3, confirm that signatures and heuristics relying on entropy and code sections are easy to fool. Moreover, heuristics relying on one feature like Manalyze (i.e.: common packer section names) can easily be evaded. Popular detector like DIE combining approaches based on signatures and heuristics can be attacked with a decent success rate. Given the results of PyPackerDetect, this requires further attention as it may use heuristics relying on features that are not covered here.

5 Conclusion

In this paper, the problematic of executable packing and performance of static detection was stated, addressing adversarial attacks for testing related significant features and their robustness. The framework relying on Packing Box was presented with its enhancement for supporting alterations of binary samples. Five of these alterations were defined, implemented and evaluated for testing against a chosen set of detectors (Q1). Furthermore, three out of these five were shown to have impact on significant features from the literature, including bytes after the Entry Point and features based on entropy and code sections (Q2), and to evade detection by popular open source packing detectors integrated into the Packing Box including Bintropy, DIE, PEiD and RetDec (Q3).

A future study will be conducted on machine learning based detectors with an extended set of alterations. The goal will be to study the robustness of features and algorithms to these alterations and to refine the set of significant features.

Acknowledgements. The authors are funded by the CyberExcellence project (RW, Convention 2110186). They want to express their gratitude to Romain Jennes who made a significant contribution through his master's thesis.

References

1. Anderson, H.S., Kharkar, A., Filar, B., Evans, D., Roth, P.: Learning to evade static PE machine learning malware models via reinforcement learning (2018)
2. Bat-Erdene, M., Park, H., Li, H., Lee, H., Choi, M.S.: Entropy analysis to classify unknown packing algorithms for malware detection. International Journal of Information Security (2016)
3. Bertrand Van Ouytsel, C.H., Dam, K.H.T., Legay, A.: Analysis of machine learning approaches to packing detection. Computers & Security (2023)
4. Biondi, F., Enescu, M.A., Given-Wilson, T., Legay, A., Noureddine, L., Verma, V.: Effective, efficient, and robust packing detection and classification. Computers & Security (2019)
5. Carlini, N., Athalye, A., Papernot, N., Brendel, W., Rauber, J., Tsipras, D., Goodfellow, I., Madry, A., Kurakin, A.: On evaluating adversarial robustness (2019)
6. Choi, M.J., Bang, J., Kim, J., Kim, H., Moon, Y.S., Díaz-Verdejo, J.: All-in-one framework for detection, unpacking, and verification for malware analysis. Security and Communication Networks (2019)
7. Choi, Y.S., Kim, I.K., Oh, J.T., Ryou, J.C.: PE file header analysis-based packed PE file detection technique (PHAD). In: International Symposium on Computer Science and its Applications (2008)
8. Demetrio, L., Biggio, B., Roli, F.: Practical attacks on machine learning: A case study on adversarial windows malware. IEEE Security & Privacy (2022)
9. D'Hondt, A., Bertrand Van Ouytsel, C.H., Legay, A.: Experimental toolkit for manipulating executable packing. In: International Conference on Risks and Security of Internet and Systems (2023)
10. Fang, Z., Wang, J., Li, B., Wu, S., Zhou, Y., Huang, H.: Evading anti-malware engines with deep reinforcement learning. IEEE (2019)
11. Han, S., Lee, K., Lee, S.: Packed PE file detection for malware forensics. In: International Conference on Computational Science and Its Applications (2009)
12. Kancherla, K., Donahue, J., Mukkamala, S.: Packer identification using Byte plot and Markov plot. Journal of Computer Virology and Hacking Techniques (2015)
13. Khan, M., Akram, M., Riaz, N.: A comparative analysis of software protection schemes. International Arab Journal of Information Technology (2014)
14. Ling, X., Wu, L., Zhang, J., Qu, Z., Deng, W., Chen, X., Wu, C., Ji, S., Luo, T., Wu, J., Wu, Y.: Adversarial attacks against windows PE malware detection: A survey of the state-of-the-art. Computers & Security (2021)
15. Lyda, R., Hamrock, J.: Using entropy analysis to find encrypted and packed malware. IEEE Security & Privacy (2007)
16. Pareek, H., Arora, R., Singh, A.: A heuristics-based static analysis approach for detecting packed PE binaries. International Journal of Security and Its Applications (2013)
17. Shin, D., Im, C., Jeong, H., Kim, j., Won, D.: The new signature generation method based on an unpacking algorithm and procedure for a packer detection. In: International Journal of Advanced Science and Technology (2011)
18. Ugarte-Pedrero, X., Balzarotti, D., Santos, I., Bringas, P.G.: SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In: IEEE Symposium on Security and Privacy (2015)
19. Wu, C., Shi, J., Yang, Y., Li, W.: Enhancing machine learning based malware detection model by reinforcement learning. In: International Conference on Communication and Network Security (2018)