Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem

Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, Davide Balzarotti

Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem

Alessandro Mantovani
EURECOM, France
alessandro.mantovani@eurecom.fr

Simone Aonzo
DIBRIS - University of Genoa, Italy
simone.aonzo@dibris.unige.it

Xabier Ugarte-Pedrero
Cisco Systems
xpedrero@sourcefire.com

Alessio Merlo
DIBRIS - University of Genoa, Italy
alessio@dibris.unige.it

Davide Balzarotti
EURECOM, France
davide.balzarotti@eurecom.fr

***Abstract*—An open research problem on malware analysis is how to *statically* distinguish between packed and non-packed executables. This has an impact on antivirus software and malware analysis systems, which may need to apply different heuristics or to resort to more costly code emulation solutions to deal with the presence of potential packing routines. It can also affect the results of many research studies in which the authors adopt algorithms that are specifically designed for packed or non-packed binaries. Therefore, a wrong answer to the question *"is this executable packed?"* can make the difference between malware evasion and detection.**

It has long been known that packing and entropy are strongly correlated, often leading to the wrong assumption that a low entropy score implies that an executable is NOT packed. Exceptions to this rule exist, but they have always been considered as one-off cases, with a negligible impact on any large scale experiment. However, if such an assumption might have been acceptable in the past, our experiments show that this is not the case anymore as an increasing and remarkable number of packed malware samples implement proper schemes to keep their entropy low. In this paper, we empirically investigate and measure this problem by analyzing a dataset of 50K *low-entropy* Windows malware samples. Our tests show that, despite all samples have a low entropy value, over 30% of them adopt some form of runtime packing. We then extended our analysis beyond the pure entropy, by considering all static features that have been proposed so far to identify packed code. Again, our tests show that even a state of the art machine learning classifier is unable to conclude whether a low-entropy sample is packed or not by relying only on features extracted with static analysis.

I. Introduction

Both benign and malicious applications have valid reasons to hide or disguise their internal behavior; the former to deter attempts to reverse engineer their code and break software protection mechanisms, and the latter to evade detection from antivirus engines and security products. A wide range of *anti-reversing* techniques exist that modify the binary code of a program to make it difficult for humans to understand and for computers to analyze. Among them, *code obfuscation* and *runtime packing* are the most frequently adopted by both malware and goodware authors.

On the one hand, *Obfuscation* aims at rewriting a program in a way that preserves its semantic but complicates its form. This can be done, for example, by flattening the control-flow, inserting dead code or opaque predicates, or by adding sequences of instructions that can confuse disassemblers ([27], [12], [11], [30], [15], [38]). Obfuscation plugins are often included in popular compiler toolchain infrastructures (e.g., Obfuscator-LLVM [21] and Proguard [23]).

On the other hand, *Runtime Packing* is a technique that was originally introduced to save disk space by compressing (at rest) and decompressing (at runtime) the code of an application. More generally, the term is used today to describe a class of techniques designed to store a compressed, encrypted, or otherwise encoded copy of the original program – thus preventing any static analysis of the code itself. Packed samples rely on a short unpacking routine that allows them to reconstruct the original application code in memory and then execute it.

While the exact fraction of packed malware samples is still unclear, in a recent study by Rahbarinia et al. [34], the authors found that 58% of the malicious downloaded files are packed with an off-the-shelf packer. However, their estimation does not take into account the presence of custom packers (35% of packed malware adopts custom packers, according to [29]). Moreover, the authors rely on signature-based tools that are known to generate many false positives – as we show in more detail in Section V. In any case, the widespread adoption of packing makes the problem of correctly and efficiently answering the question *"is an executable packed?"* fundamental in malware analysis. In fact, many classes of techniques – such as static analysis, clustering, and similarity among samples – do not work or provide poor results in the presence of packed executables. This forces researchers to pre-process packed samples by introducing a very costly and time-consuming dynamic unpacking phase, or by completely replacing static approaches with more resilient solutions based on dynamic analysis.

A wrong classification of packed samples can also pollute the datasets used in many malware analysis studies. For

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24297
www.ndss-symposium.org

instance, researchers often rely on datasets that include both packed and not packed samples, and errors in this separation can lead to unreliable or difficult to reproduce experimental results.

To solve these problems, the security community developed a number of efficient tests to assess the presence of packing. Historically, the Shannon entropy of a program was adopted for this purpose, as both encrypted and compressed data are characterized by a very high entropy — which can be easily distinguished from that of machine code. While early studies (e.g., [26]) classified executables just according to their *average* entropy, researchers quickly moved towards entropy computations performed at a lower granularity, i.e., by relying on sliding windows or by calculating the entropy of individual sections. These more fine-grained techniques were often described as very successful in identifying the presence of packing. For instance, Han and Lee [19] reported 99% of accuracy and precision by looking at the entropy of individual sections. Another well-known approach to identify the presence of packing relies on the use of custom signatures, as applied by popular tools like Detect It Easy [3], Manalyze [4], and PEiD [7]. However, this solution is prone to errors, and it is unable to identify previously-unknown packing routines – as we show in more detail in Section V.

Since entropy became a discerning metric to discover packed code, both researchers and malware authors experimented with techniques to pack executables while maintaining the entropy low. For instance, in 2010, Baig et al. [10] discussed the possibility of using different encodings to reduce the entropy and evade the checks performed by antivirus software. However, the study was purely theoretical, and the authors did not provide any evidence of the actual adoption of such schemes in the wild. Two years later, Ugarte et al. [49] found that samples belonging to the *Zeus* family contained trivial countermeasures to tamper with entropy checks. In this case, the malware authors padded the encrypted data by inserting the same byte (or a subset of bytes) repeated multiple times – in Section IV we inspect the details of how this and the other schemes affects the entropy. Since then, this phenomenon has been sporadically mentioned by malware analysts, but it has never been discussed in detail, and its adoption by malware authors has never been measured. Therefore, even if the existence of low-entropy packing was known to researchers, it was often dismissed as statistically irrelevant and with a negligible impact on practical experiments. As a result, researchers (such as in [26], [33], [39], [48], [34], [47]) continued to resort to entropy-based metrics and static signatures to identify the presence of packing. For instance, in the extensive analysis and large-scale measurement of malware packing performed to date [47], the authors selected their samples from VirusTotal [8] by querying for files with an entropy greater than seven.

Research questions

Even though security experts do not rely solely on entropy to identify packed samples, there is no systematic study that measured how prevalent low-entropy packing schemes are in the wild and whether existing techniques are able to correctly classify these samples. Moreover, there are popular tools (discussed in Section V), academic papers (discussed in Section VI), and even books ([44], [24]) that still adopt the approximation *packed* $\approx$ *high entropy*. The goal of this paper is to show that this simple approximation is not correct in a large number of cases, and to improve our knowledge of low-entropy packing by answering the following research questions:

1) Which tricks and which packing techniques are used by real-world malware to lower their entropy?
2) How widespread are these techniques in the wild? Is low entropy packing a significant trend that needs to be considered when designing malware experiments?

To answer these questions, we assembled a dataset containing 50,000 low-entropy malicious samples belonging to multiple families. We then developed a dynamic analysis tool to classify each sample and categorize the scheme and transformations applied to the packed code. Our dynamic analysis tool reported that over 30% of them adopt some form of runtime packing.

For this reason, we decided to investigate if other features can still be used to detect the presence of packing. In fact, while some papers (e.g., [26], [33], [47]) and tools (e.g., [7], [3]) consider only entropy to distinguish packed from non-packed malware, state of the art solutions use a combination of different static features, often based on PE structural properties. Therefore, we introduced two additional research questions:

3) Are existing *static* solutions able to distinguish low-entropy packing from unpacked samples?
4) If not, can we do that by combining all static features that have been proposed to date in related works, or new research is needed to solve this problem?

In Section V we show how the most popular and actively maintained static tools available today perform on our dataset. Finally, in Section VI, we collected all the static features that have been proposed in previous studies as reliable indicators of the presence of packing. We then trained several classifiers on the union of these features and tested them on our dataset of low-entropy malware (containing both packed and not packed samples). It is important to note that our goal was not to design a new classification scheme based on the combination of all existing features but only to understand whether these features can successfully classify samples in the presence of low-entropy packers.

II. Background

A. Packing and Scope of our Study

Over the years, anti-reversing techniques have evolved in several directions, which include code obfuscation, compression, encryption, polymorphism, metamorphism, and runtime packing. This process and the real-world adoption of these techniques have been largely discussed in [50], [31], [36], [47]. Since packing, as used by today's malware, does not have a precise definition, it is essential to clarify which techniques we cover, and which we do not, in the rest of the paper.

First of all, to draw a line between packing and other forms of anti-reversing, we consider packing only when I) the original code of the application is already present in the file

but is NOT present in an executable form (i.e., it is encrypted, compressed, or otherwise transformed), and II) the original instructions are later recovered and executed at runtime. We consider instead obfuscation when the code is present in the binary and it retains the ability to be executed, even if it is hard to understand (for humans and/or automated tools) or analyze because it was re-written with the goal of hindering reverse engineering. For the same reason, if a program encrypts all its data but not its instructions, we do not consider that as a form of packing in our study.

Dynamically-generated code (that also includes self-modifying code) is a generic term that refers to techniques used to generate or modify code at runtime dynamically. In a broad sense, packing relies on these techniques, and it is, therefore, a form of dynamically-generated code. However, not all forms of dynamically generated code are packing – for instance in the case of just-in-time compilers. To distinguish among the two, in our study we measure the size of the unpacked code (see Section IV) and use this information to separate the cases when the actual application code is unpacked from the cases when just a small snippet of code (e.g., a shellcode) is generated at runtime.

Second, we limit our analysis to *runtime* packers that recover and execute the original code at runtime. Droppers that download a compressed archive from the Internet, unpack them on disk and then run the contained application are outside our scope (as both the dropper and the dropped files could be independently statically analyzed).

Finally, we do not consider emulators (like those included in the Themida packer) that transform the original instructions into a new instruction set and then execute them by using a custom emulator. In fact, in this case, the original code is *never* recovered, but instead permanently replaced with an (often randomized) instruction set.

B. Entropy of Executable Files

Entropy is a metric to measure the uncertainty in a series of numbers (or bytes) or, in other words, to capture how difficult it is to independently predict each number in the series. The difficulty in predicting successive values can increase or decrease depending on the amount of information the predictor has about the function that generated the numbers, and any information it retained about the prior numbers in the series.

In particular, the Shannon entropy H of a discrete random event x tries to predict the number of bits required to encode a piece of data, as given by the formula:

$$\mathrm{H}(x) = -\sum_{i=1}^{n} \mathrm{P}(x_i) \log_2(\mathrm{P}(x_i))$$

where $\mathrm{P}(x_i)$ is the probability of the i^{th} unit of information (such as a number) in event x's series of n symbols. This formula generates entropy scores between 0.0 and 8.0 when considering that each symbol can have 256 values as it is the case for binary data. Both *lossless compression* and *encryption* functions typically generate high entropy data. In fact, lossless compression functions start by generating a statistical model for the input data, then use such a model to map input data to bit sequences in a way that frequently encountered data will produce a shorter output than infrequent ones; this removes predictability, which increases the entropy. The same applies to encryption functions, as they are specifically designed to generate unpredictable data.

Since the generation of a *packed executable* often relies on compression and/or encryption to disguise the application code, packed files are usually characterized by having a high entropy. As a consequence, entropy was the primary metric used in the past to classify *packed* executables [26].

However, many file formats for executables, such as Portable Executable (PE), Executable and Linkable Format (ELF), and Mach Object (MO) divide the file into a number of isolated sections. Obviously, this way of partitioning an executable affects the distribution of its entropy. For instance, machine instructions are often redundant, thus resulting in middle-range (typically 5-to-7) entropy scores, while strings of English text result in even lower entropy values (on average 4.7 [18]) due to the limited number of characters they employ. To better discriminate among different areas of an executable, researchers replaced file-level entropy scores with a more fine-grained computation performed at the level of individual sections [19] or by applying a (sliding) windows over the program's bytes [32], [48].

C. Entropy and XOR Encryption

Since packing usually encrypts code to hide it, we set up an empirical experiment focused on PE x86 code encryption to distinguish between average and high entropy values of plain and encrypted code. On Windows 7, we installed the top 10 applications from the Microsoft Store [28], including top browsers and the Visual Studio IDE. We then randomly selected $1,000$ PE executable files, both 64 and 32 bit, from the Program File folder[1]. For each of them, we calculated the entropy of their `.text` sections; then we *XORed* the `.text` section with a randomly generated key, and we re-calculated the entropy of this new encrypted data. We repeated the experiment 128 times, changing the key length from 1 to 32 Bytes.

Figure 1 shows the evolution of the entropy for different key lengths. The circle shows the mean of the 128 experiments, the thick vertical line is the standard deviation, and the thin vertical line shows the range between the maximum and minimum value. As shown in the image, the entropy slowly grows accordingly to the length of the key. When the key length is only 1 Byte long, the entropy does not change as this is just a substitution of the plain-text code and does not alter the frequency of the symbols. Our test shows that the average entropy of real-world plain x86 code is around 6.2 ± 0.3, and by using a 2-bytes key the entropy increases to 6.7 ± 0.3. Figure 2 shows the difference between the means of the entropy of the XORed code and the original code – emphasizing the rapid effect that the key size has on the entropy of the data.

Finally, we observed that state-of-the-art approaches [47] and frequently used tools (e.g., [3], [4], [6] – discussed in Section V) adopt 7.0 as entropy threshold to separate packed and not packed executables. According to our graphs, this

[1]We ensured that each file was not previously packed by using the tool we presented in Section III.

Fig. 1. `.text` section entropy w.r.t. XOR encryption

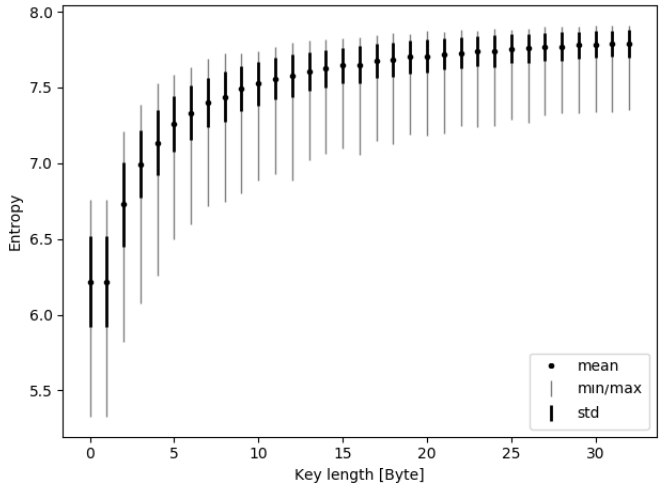
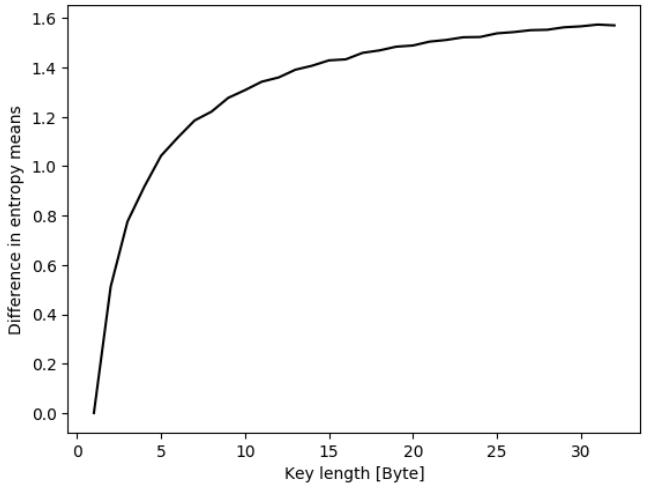


Fig. 2. Encrypted `.text` section – difference in entropy means



value is obtained on average by xor-ing the code with a key of 3 bytes. In the rest of our paper, we will use this threshold to distinguish *low entropy* data ($H < 7.0$) from *high entropy* data ($H \geq 7.0$) and we use this value to construct our low-entropy malware dataset.

III. Prevalence of Low-Entropy Packing

A. Dataset

We built our dataset by downloading $50,000$ Portable Executable (PE) files, (excluding libraries and `.Net` applications), randomly selected among those submitted to VirusTotal [8] between 2013 and 2019. We only selected PE samples classified as malicious by more than 20 antivirus engines, and such that the entropy of each section, of the entire file, and of potential overlay data[2] were less than 7.0 (as motivated in Section II-C). We adopted these conservative criteria to ensure that a sample is certainly malicious and contains neither compressed nor encrypted data.

Furthermore, we collected a second smaller dataset containing 476 samples used in APT campaigns [1], which satisfies the same low entropy constraints. From now on, we will refer to this dataset as the *APT dataset*. The samples belonging to the APT dataset were collected over a period spanning from 2015 to 2018.

B. Analysis

To carry out our first experiment, we designed and implemented **Packer Detector** (hereafter, *PD*), a dynamic analysis tool built on top of the PANDA [17] analysis framework. The goal of *PD* is to precisely trace unpacking behavior by monitoring when a sample executes a memory area which it previously modified. The analysis of a single sample produces as output several text files that are subsequently analyzed to reconstruct the behavior of the sample.

Since the original code of a packed sample is hidden, the unpacking procedure must carry out some operations to retrieve, restore, and store somewhere in memory the unpacked code. For this reason, *PD* dynamically executes the target sample on a virtual machine (Windows 7, 64 bit) by spawning the corresponding process and monitoring its registers and memory content. Each sample is executed until the main process exits or until a maximum timeout of 40 minutes is reached. The virtual machine gateway points to an INetSim[3] instance, which provides fake HTTP/S and DNS responses to deceive the sample under analysis into believing that it is connected to the Internet. Despite INetSim delivers fake files based on the file extension in the HTTP request (e.g., .html or .exe), it is configured to avoid returning any executable code, since we do not want to analyze malware which uses external data because it can violate our low entropy constraints.

PD collects write accesses on the memory of the main process and, if applicable, of its child processes, and stores this information as a list $[WL]$ (Writes List) of tuples. The memory accesses are detected thanks to the PANDA callback `PANDA_CB_VIRT_MEM_AFTER_WRITE` which is raised every time a process performs a memory write. If the PID matches with that one of the process under analysis (or with one of his child ones), we store a tuple in the list $[WL]$. Each tuple contains the Program Counter (PC) register pointing to the instruction that triggered the write operation and the target address AW (Address Written) of the operation. This means that for the i-th write operation to the address AW_y performed by the instruction at address PC_x, we have a tuple $\langle PC_x, AW_y \rangle_i \in [WL]$. If the write operation involves more than one byte, the system stores them separately. For instance, if a sample executes the instruction " `mov WORD PTR [0x1000], 0x4142` " at the address `0x1234`, *PD* manages the size directive adding the tuples $[\langle \texttt{0x1234}, \texttt{0x1000} \rangle, \langle \texttt{0x1234}, \texttt{0x1001} \rangle] \in [WL]$. Furthermore, if the PC reaches a previously written address in the tuple $\langle PC_x, AW_y \rangle \in [WL]$ (i.e., AW_y points to instruction that it is getting executed), it copies the tuple into another list $[WXL]$ (Written and eXecuted List); at the end of the execution this list will contain all the written-then-executed

[2]The overlay is just appended data to the end of the executable file that is ignored when loading an executable into memory because it is not covered by the PE header. Anyway, opening the executable file in reading mode will allow access to the entire file including the overlay portion.

[3]https://www.inetsim.org/

addresses and the PC values that triggered the write operation. When the sample terminates, *PD* analyzes the $[WXL]$ list: if the sample is packed, the list is not empty and it encompasses some memory regions of consecutive addresses (modulo the x86 length of instructions) that contained the unpacked code. However, when a sample manually loads a Dynamically Linked Library (DLL) and then executes one of its functions, *PD* would detect this behavior as part of an unpacking routine. To remove this noise, our tool further checks whether the program counter points to code that belongs to a DLL, and remove these cases from our analysis. We also use a threshold of 800 bytes on the length of the $[WXL]$ list to exclude samples which simply decrypt a short shellcode, a behavior that we do not consider a form of packing and that anyway would likely not significantly affect the overall entropy.

The heuristic adopted by *PD* can also generate false negatives (i.e., packed samples detected as not packed) if the sample runs incorrectly because of an unexpected crash, incorrect command-line arguments, missing dependencies, or virtual environment evasion [4]. To avoid the risk of polluting our dataset with wrong labels, we decided to conservatively discard the samples that did not exhibit a sufficient amount of runtime behavior, and that therefore might have been incorrectly executed. This includes samples that did not invoke at least ten disk- or network-related syscalls as well as samples whose executed instructions did not span at least five memory pages. For this reason the *PD* hooks the disk/network-related syscalls (for instance *NtOpenFile* or *NtCreateFile*) relying on the syscall hooking interface offered by PANDA. We also keep track of the code coverage of the sample, i.e., the number of instructions executed compared to the total number of instructions in the executable sections (typically `.text`). The goal of these selection criteria is not to detect evasive malware, which is still an open problem, but to remove from our dataset those samples that could be incorrectly classified as not packed simply because they failed to run. By applying these simple heuristics, we removed a total of $3,705$ malware samples from our dataset. Based on our conservative thresholds, it is safe to assume that the remaining samples executed long enough to at least unpack their code. For this reason, from now on, we consider $46,295$ as the total number of samples over which we compute our results.

C. Results

During our analysis, we run into a class of samples that, while packed with a high-entropy scheme, evaded our set of filters described in Section III-A. These samples contained encrypted data, but the data was not stored in any of the section nor the overlay area. For instance, a family of file infectors adopted this technique to inject its encrypted code in an area created between the PE header and the first section. While this data belongs neither to the PE header nor to any section, it is automatically loaded in the main memory at runtime (unlike, for instance, the overlay data that needs to be manually loaded by the program). Moreover, since the size of this encrypted code is small with respect to the size of the entire file (approximately 2.6%), it has little impact on the total entropy

[4]Virtual environment evasions are techniques aimed at detecting whether an executable is running on bare-metal or a virtual machine (regardless of it being emulated or based on a hypervisor).

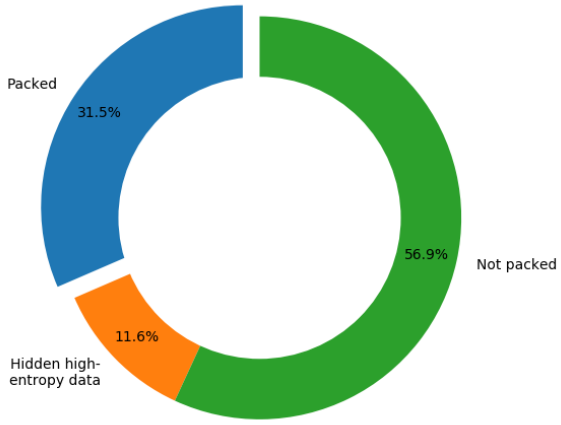


Fig. 3. Dataset composition (cardinality = $46,295$)

of the file. In addition to the area between the PE header and the first section, we have also discovered samples that used the empty area (if present) among sections to store their packed data. In total, 11.6% $(5,386/46,295)$ of the samples in our dataset adopted this interesting, and to the best of our knowledge previously undocumented, scheme to store packed code in a way that evades common entropy-based checks. Among them, the two prevailing families were *hematite* (64%) and *hworld* (35%). Since these samples successfully evaded our entropy checks but without using a low-entropy scheme, we decided to consider them as a separate category in our dataset.

Over the remaining low-entropy samples, our tool discovered that a stunning 31.5% $(14,583/46,295)$ employed some form of packing. This shows that entropy alone is a very poor metric to select packed samples and that roughly one-third of the samples with entropy lower than seven are still adopting some form of runtime packing to prevent static analysis. This percentage is even higher if we exclude the samples with hidden high-entropy data. In other words, if we pick a random malware sample that contains no information with entropy higher than seven, according to our experiments there is a 35.6% probability $(14,583/40,909)$ that it is packed with a low-entropy scheme. The overall composition of our dataset is summarized in Figure 3.

We also downloaded the VirusTotal report of every sample in our dataset and using *AVclass* [40], a malware labeling tool, and we have identified the family associated to each sample. Table I reports a ranking of the top ten families in the packed and not-packed categories.

Finally, in the *APT dataset* we did not find any sample that has hidden high-entropy data, while low entropy packing schemes were adopted by 15% of the samples. This shows that low entropy schemes are a well-known practice for malware authors nowadays and the phenomenon is significantly widespread in the wild, leading us to our next research question: *which packing techniques do malware authors adapt to keep the entropy below the suspicious threshold?*

TABLE I. TOP 10 FAMILIES DISTRIBUTION IN OUR DATASET

Packed	%	Not packed	%
sivis	28.0	lamer	10.4
unruy	11.6	daws	8.8
vobfus	9.5	vbclone	8.0
dealply	5.4	sivis	7.5
upatre	4.1	triusor	4.5
shipup	4.0	flystudio	4.0
gepys	3.5	zegost	3.9
vilsel	2.9	mailru	3.6
sality	2.3	high	2.9
hematite	2.0	nitol	2.4

TABLE II. LOW ENTROPY SCHEME

Scheme	Effect on Entropy
Padding	Decrease
Encoding	Decrease
Mono-alphabetic Substitution	Unchanged
Transposition	Unchanged
Poly-alphabetic Substitution	Slightly increase

IV. LOW ENTROPY PACKING SCHEMES

In this section, we describe the experiments we conducted to enumerate and analyze the different techniques adopted by malware authors to keep the entropy below detectable levels, and measure the frequency in which they appear in our dataset. We emphasize that we refer to *low entropy packing schemes* regardless of their effect on the entropy (increasing, decreasing, or unchanged), as long as such schemes produce *low* entropy data according to our results in Section II-C. Moreover, it is important to note that sophisticated packers often involve several layers of unpacking routines, in which the first layer unpacks the second one, which in turn unpacks the next layer and so on until the original code is reconstructed. However, for our purpose, we only need to study the first unpacking layer, as it is the only visible from a static analysis point of view and the only one that determines the entropy of the data. As we will discuss later in this section, malware authors may also decide to use stronger encryption in deeper layers as long as they keep the entropy of the first layer low.

A. Schemes Taxonomy

We can divide the low entropy schemes observed in the wild into five main categories, summarized in Table II. The table also shows the effect that each scheme has on the final entropy. While some techniques can be used to effectively lower the entropy of data (and therefore 'hide' an already packed sequence of bytes), others can only maintain (or slightly increase) the current entropy, thus requiring to be applied as standalone solutions on the original application code.

Byte Padding includes all techniques in which additional low-entropy data is added to the packed section to decrease the overall entropy. This data typically consists of a single byte, or a repetitive subset of bytes, that are either appended at the end of the code or interleaved with the packed instructions. The unpacking routine, accordingly, skips over the padding while restoring the original instructions. *Byte padding* alone is not a packing technique, and therefore it is often used in combination with other encryption or compression schemes.

Encoding-based schemes decrease the overall entropy by representing the packed information using a different number of bits, thus encoding the same data with a different alphabet of symbols. Although we observed some samples applying well-known encoding schemes to pack their code, other malicious samples often implement their **custom encoding** (during our analysis we just observed 6-bit alphabets). As encoding-based schemes can lower the entropy of high-entropy data, they can be used to mask multi-layers approaches that also employ traditional encryption packing.

Monoalphabetic Substitution-based approaches aim at replacing every single byte in the packed payload with a different byte, computed either by using a simple algorithm (e.g., a *XOR* with a 1-byte key) or by looking up each symbol in a translation table.

Transposition is another technique that does not alter the byte distribution and the entropy of the data. In this case, either individual bytes or sequences of bytes are shuffled around to recompose the original code. Sometimes the transposition scheme is fixed, while in other cases the samples embed the 'instructions' to reassemble the bytes in the correct order in the packed data itself.

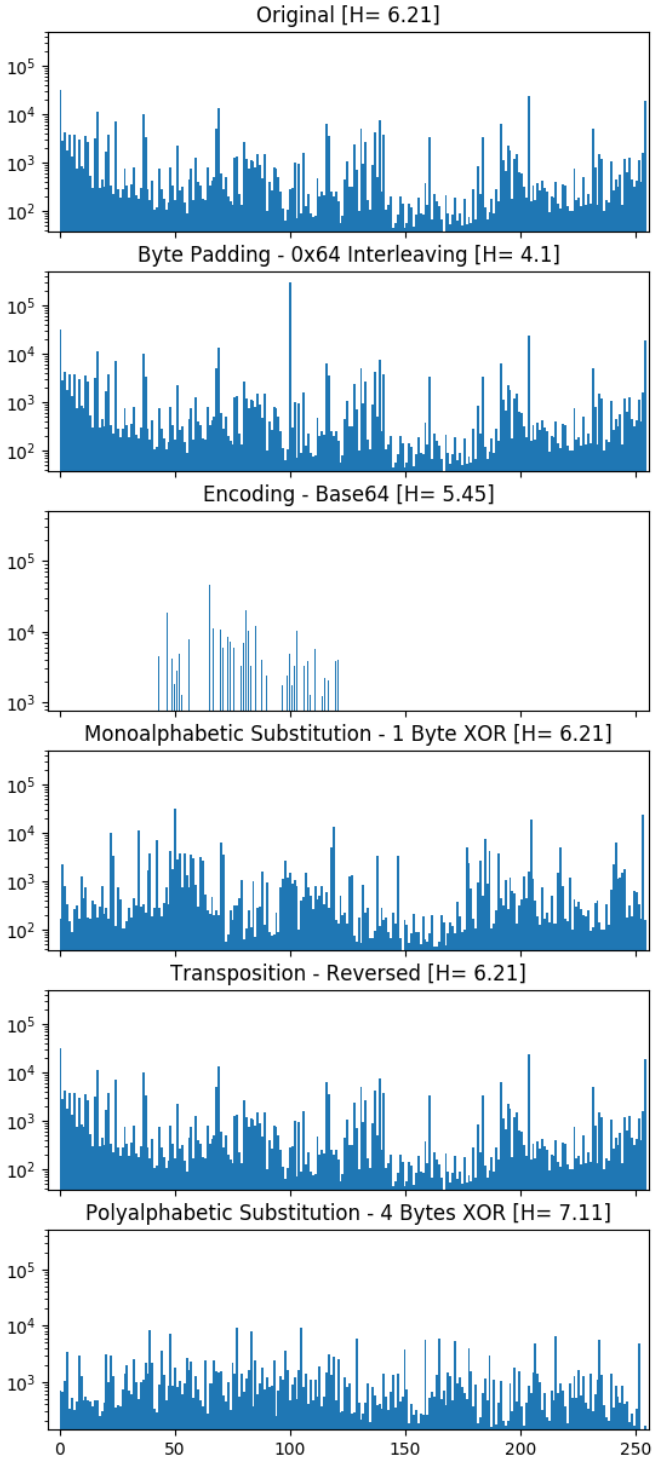
Polyalphabetic Substitution schemes are simple cryptographic techniques that extend simple byte substitution by using multiple substitution alphabets. Common examples of this approach are the classic Vigenère cipher or the `XOR` encryption with a multi-byte key. While these techniques usually result in an increased entropy score, the use of very short keys (e.g., 2-4 bytes, as shown in Section II-C) do not significantly modify the byte distribution, and therefore it limits the increase of the entropy level.

B. Schemes in action

To give an idea of how such schemes work on a real example, we have taken a benign file from the samples we used in the experiment in Section II-C. In particular, we have chosen a sample with the entropy of its `.text` section corresponding to the average entropy we previously measured in the same experiment. Then, we applied an example of each of the different low-entropy scheme listed above on its `.text` section. As shown in Figure 4, we implemented respectively: padding – interleaving the byte `0x64` after each original byte (thus doubling the size), encoding – base64, substitution – XOR with a one-byte key, transposition – byte ordering reversed, poly-alphabetic substitution – XOR with 4 Byte long key. The graph shows for each byte $[0, 255]$ (represented on the x-axis) its frequency in the data (on the y-axis) plotted on a logarithmic scale. In the padding plot, the `0x64` byte is the most frequent; this scheme is noteworthy for the way that it effectively decreases the entropy, with the downside of increasing the original source size. The encoding plot contains only the bytes belonging to the *base64* scheme, decreasing the entropy accordingly. Looking closely at the Substitution plot, the reader can notice that the frequencies are shuffled w.r.t.

the original distribution; for example, given that we used the byte `0x32` as the key, the original `0x00` byte frequency has been moved (`0x32` $\oplus$ `0x00` = `0x32`) to the `0x32` (50 in decimal) frequency. Given that the frequency distribution does not consider the order, the transposition and the original plot are identical, including the entropy. Lastly, the poly-alphabetic substitution is characterized by a more uniformly distributed bytes frequency, and in fact it is the only one that increases the entropy over the 7 threshold.

Fig. 4. Byte Frequency Distribution w.r.t. Schemes



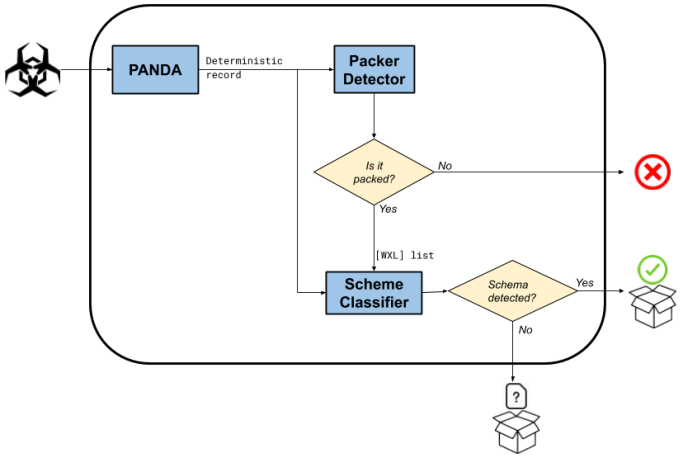


Fig. 5. Architecture of our analysis tools

C. Scheme Classifier

Once Packer Detector identifies a sample as using some form of runtime packing, a more refined analysis is needed to detect to which of the previously introduced five categories the low-entropy packing scheme belongs to. To accomplish this second set of experiments, we developed another dynamic analysis tool, also based on PANDA, that we call the **Scheme Classifier**. This tool relies on the output of Packer Detector and applies some heuristics based on the fact that every packing scheme needs to follow the same steps: i) locate and access the source buffer that contains the low entropy packed data, ii) perform operations on such data, iii) write the unpacked data in the destination buffer. We sketched the architecture of our tools and how they are integrated together in Figure 5.

It is worth remembering that the output of Packer Detector is a list of tuples, named $[WXL]$. Each tuple $\langle PC_x, AWX_y \rangle \in [WXL]$, contains the program counter PC_x of the instruction that triggered the write operation in memory, and the target address AWX_y where the information was subsequently stored and executed. Accordingly, this information defines the memory regions that contain the destination buffer of the unpacking routine. Moreover, given that PANDA supports the deterministic record and replay of a sample, the tool performs its analysis by replaying the same trace that was recorded by Packer Detector. For each unpacking operation, the Scheme Classifier disassembles (using Capstone [2]) and analyses the assembly instructions executed just before the memory write to the destination buffer. It then parses the assembly code and, by relying on the PANDA framework, it reads the values contained in the registers and in the referred memory addresses. The instructions and the values we obtain are used for two reasons: first, the Scheme Classifier performs a backward data-flow analysis to locate the source buffer (S_b) – where the packed data is located. This corresponds to the identification of all the memory read operations which are supposed to contain the packed code which is being unpacked. For this purpose the Scheme Classifier relies on the PANDA callback `PANDA_CB_VIRT_MEM_AFTER_READ`, which is triggered every time a memory read operation is performed by the process we are tracking (in our case the sample under observation). Second, the Scheme Classifier extracts all the mathemat-

ical operations that are applied to the source bytes to generate those in the destination buffer (D_b). To achieve this second step, the tool makes use of the `PANDA_CB_INSN_EXEC` exported by PANDA, which allows us to analyse all the instructions actually executed by the sample. Since we already know the program counter values corresponding to interesting memory writes (as Packer Detector provides such values in the $[WXL]$ list as mentioned before), we just need to track these specific values and the previous mathematical instructions without taking care of the other instructions executed. It then uses these two pieces of information to classify the possible packing scheme adopted by a sample, by following this sequence of rules:

1) The Scheme Classifier first applies some rules to identify the presence of known encoding schemes (*base64*, *base32*, ...) in the source buffer S_b. If it recognizes a standard encoding, the Scheme Classifier marks the sample as *encoding*.
2) If the frequency of the bytes in the source and destination buffers is the same, but bytes appear in a different order, it classifies the scheme as a *transposition*.
3) If the byte distribution in the D_b is shuffled with respect to the S_b and the entropy is the same, the Scheme Classifier reports it as *mono-alphabetic* substitution.
4) It then looks for arithmetic/logic operations (XOR, ADD, ...) that modify the S_b and write to the D_b. If it finds an interesting cryptographic operation, it tries to extract the potential encryption key by analysing the disassembly and reading the value stored in the registers and memory. For example, if before a memory write, the target value was previously XORed with a 2-bytes fixed value, this means that the sample is using a XOR encryption with a 2-bytes key. In this case, the unpacking scheme is classified as *poly-alphabetic* (the *mono-alphabetic* case is captured in the previous step).
5) If no interesting operations are detected, the Scheme Classifier looks at the entropy of the input buffer. If it is the same that would be obtained by applying a known encoding to the output buffer, but the set of symbols is different, it marks it as a potential *custom encoding*.
6) When S_b and D_b match except for a subset of bytes that is present with high frequency in S_b and with low frequency in D_b, the Scheme Classifier infers that *byte padding* is being used.
7) When the Scheme Classifier cannot apply any of the previous techniques, it marks the scheme as *unknown*, and leave it for a further manual investigation.

D. Results

In Table III we report the result of the Scheme Classifier; namely, the distribution of low-entropy schemes that we observed in our fine-grained analysis performed over all the $14,583$ samples found by Packer Detector. When possible, we also specify the specific type of transformation that is employed. It is also worth noting that the heuristics applied by the Scheme Classifier are extremely time-consuming: in average, they require around 90 minutes per sample.

`XOR`-based encryption is by far the most prevalent technique in our dataset, accounting for more than 76.7% of the analyzed samples. It is present both in its simplest form (`xor` with a single constant byte) in 29.8% of the cases, as well as with multi-byte keys of various length. A basic (`base64`) encoding was used in 3.9% of the samples, while padding accounted for slightly more than 8%. In 97.9% $(14,276/14,583)$ of the cases the Scheme Classifier detected a tangible unpacking scheme, so we are reasonably sure that the vast majority of the samples discovered by the *PD* are actually packed. The remaining 2.1% $(307/14,583)$ contains either samples adopting unforeseen schemes that we could detect with our tool, or possibly samples using other forms of dynamically-generated code that were not removed by our heuristics.

TABLE III. SCHEME DISTRIBUTION

Scheme	Type	%
Padding	-	8.0
Encoding	standard	3.9
	custom	0.5
Mono-alphabetic Substitution	XOR	29.8
	ADD	5.2
	ROL/ROR	0.5
Transposition	-	0.3
Poly-alphabetic Substitution	XOR	46.9
	ADD	2.8
Unknown	-	2.1

V. SIGNATURE AND RULE-BASED PACKER DETECTION

So far, we have discussed the nature and measured the prevalence of different low-entropy packing schemes adopted by real malware in the wild. Our experiments show that this is a ubiquitous phenomenon and that entropy alone cannot be used as a reliable indicator to identify the presence of packing. However, beyond simple entropy, security researchers also proposed other tools and techniques to identify packed samples. In this second part of the paper, we measure to which extent these alternative approaches allow us to distinguish packed from non-packed samples in presence of low-entropy schemes.

Signature-based solutions identify known packers by relying on a (typically manually curated) set of patterns that are associated with known off-the-shelf packers. Existing engines for pattern detection vary in complexity, from the ones that work on raw bytes to those that recognize and reason about the file structure. This difference consequently influences the expressive power of the employed signatures.

For our experiments, we have chosen the most popular and actively maintained tools available today that rely on open signatures: Detect It Easy [3], Manalyze [4], and PEiD [7].

Detect It Easy (DIE) adopts an open architecture of signatures, based on a scripting language similar to JavaScript. This language provides great flexibility and expressive power that allows DIE to declare complex and fine-grained signatures.

PEiD is another widely used tool for statically analyze PE files, looking for most common packers, cryptors, and compilers. PEiD signatures only contain low-level byte patterns, which can be optionally matched either at the PE file's entry point or anywhere in the file.

Manalyze is a static analyzer for PE files, composed of several plugins. Its *packer detection* plugin adopts signatures based on the name of the PE sections (for example the UPX packer compresses all existing sections and renames them as `UPX0`, `UPX1`, etc.) as well as several rule-based heuristics designed to capture anomalies in the PE structure typically associated with the presence of packing, including unusual section names, sections both writable and executable, low number of imported functions, resources bigger than the file itself, and sections with entropy greater than 7.0 – that is the same threshold we used for constructing our dataset.

DIE and PEiD also have a dedicated component for the entropy. Even if they have different thresholds (DIE 7.0 by default, PEiD is not open source so we cannot report the precise number), all of them classify an executable as packed when its entropy is greater than a certain value. Also, the python module *pefile* [6], often used to parse and edit PE headers, contains a function that estimates if the input executable is packed, and it is solely based on the entropy. This fact highlights how this metric is still relevant nowadays and how popular tools still support the correlation between high entropy and packing.

Signature scan results

Probably because of its finely tuned signatures, `DIE` detects no well-known packer in our entire dataset. This is not a bad result, as we expect the vast majority of samples in our dataset to rely on custom packing routines. In fact, popular off-the-shelf packers are widely known and easily recognizable, thus making it more unlikely for them to 'fly under the radar', which is the main advantage of adopting a custom low-entropy scheme.

In contrast, both `PEiD` and `Manalyze` generated a large number of alerts, as summarized in Table IV. The result of both tools are comparable, but also quite surprising, as they consistently detected the presence of packing more often in not packed samples than in the packed group. For instance, signature-based mechanisms recognized 1.7%-to-2.6% of samples in the packed group but misclassified 9.6%-to-13.1% of the entries in the not packed dataset. For `Manalyze` this is due to the presence of sections names that correspond to those used by some off-the-shelf packers. We cannot say for sure why the malware authors used those names. They could be fake clues used on purpose to deceive automated tools into believing that a sample is packed with a known packer and, consequently, to trigger the use of unpacking routines that would invariably fail on the program.

Table V shows the top five common packers detected by these signature-based systems. Given that our dataset only contains samples with low entropy, the presence of compressor packers (UPX, UPolyX[5], and ASPack) and a crypto packer (Petite) immediately suggests that these are probably all false positives. In any case, we run existing unpacking tools for UPX, UPolyX, and ASPack and confirmed that all of them failed and found no sign of packing. We also manually inspected samples reported as PolyEnE and Petite (as no tools are available for these packers) and again confirmed that there were no traces of these packers. A closer look at the matching PEiD signatures revealed that they were often too general, or designed to match anti-disassembly tricks and strings that could also be used in other contexts. The only case we were able to confirm consisted of three samples recognized as packed (also confirmed by our Packer Detector) with Beria. Samples packed with Beria contains two types of byte, which we call "original" and "metadata". During the unpacking routine, the metadata bytes are evaluated through an algorithm that computes the correct offset where the original bytes need to be written inside the destination buffer. This approach does not increase the entropy as the original bytes appear unchanged (just not in the correct order) and the metadata bytes follow a strict and repetitive pattern.

Table IV also reports the alert generated by the `Manalyze` heuristic component, which flagged 57% of the packed samples and 23% of non-packed samples as likely packed. By investigating the internal logs, these misclassifications are mainly due to the presence of unusual section names or of executable permission on writable sections.

In conclusion, existing signature-based tools are well suited to detect the presence of common off-the-shelf packers, but unfortunately, generate a large number of false alerts on non-packed samples. Even worse, these false positives are more frequent on non-packed malware than on those packed by using low-entropy schemes, which suggest that these samples are difficult to classify statically.

VI. ML-BASED PACKING DETECTION

If the use of signatures or hard-coded heuristics failed to detect the packed samples in our dataset, this does not rule out the possibility to find other discriminatory features that can help identify even the most elusive form of packing.

Therefore, in the following section we explore alternative static analysis approaches proposed by other researchers. We first survey the state of the art and gather all the features that have been proposed in the past. In order to evaluate the performance of these features, we implement a machine learning classifier based on the union of all these features. With this, we do not intend to propose a new classification system, nor to compare existing approaches with respect to each other. Instead, like in previous sections, our goal is simply to evaluate whether these features are able to correctly classify our low-entropy set of samples.

A. Feature Extraction

Lyda and Hamrock [26] were the first to take into consideration entropy (computed initially over the entire file) as a metric to classify packed malware. The basic idea was then refined to calculate the entropy for each section of the sample [19] or over small byte windows [49].

Researchers also investigated the use of machine learning to train a classifier over a large number of static features. To start with, many authors [14], [33], [42], [46], [16], [41] proposed features that captured specific anomalies that packers introduce in the PE file format. Even if such features could identify off-the-shelf packers with a high level of accuracy, samples making use of custom packers could successfully evade all the checks based on the file structure alone. As a

[5]UPolyX is basically a scrambler (thus, it does not affect entropy) that needs a UPX packed input file to produce a number of different output files.

TABLE IV. SIGNATURE-BASED DETECTION RESULTS

Dataset	Manalyze signatures	Manalyze heuristics	PEiD	Manalyze Sig $\wedge$ PEiD
Packed	242 (1.7%)	8,358 (57.3%)	386 (2.6%)	214 (1.5%)
Not Packed	2,518 (9.6%)	6,023 (22.9%)	3,438 (13.1%)	2,487 (9.4%)
Hidden H-E data	0 (0%)	14 (0.3%)	2 (0.1%)	0 (0%)

TABLE V. WELL-KNOWN PACKERS DETECTIONS

Packed		Not Packed	
Name	**%**	**Name**	**%**
UPX	1.1	UPX	10.0
ASPack	0.5	ASPack	1.2
UPolyX	0.5	UPolyX	1.2
Petite	0.1	PolyEnE	0.7
PolyEnE	0.1	Petite	0.4

consequence of this, malware researchers extended the sets of features to include some that would allow capturing the presence of custom packing routines better. For instance, the approaches proposed in [32], [9], [39] and [48] adopt a larger collection of features that include structural PE attributes, heuristic values, entropy-based metrics, byte n-grams, and disassembly opcodes. Other heuristics includes the raw data per virtual size ratio (computed over all the sections), the ratio of the sections with virtual size higher than raw data or the fact that the entry point is outside any section.

In 2010, Jacob et al. [20] extended the idea of n-grams analysis by proposing a methodology for detecting the similarity between packed samples, which takes into consideration the n-grams fetched from the code sections and tunes them according to the noise introduced by the different packers. In 2019 Lim et al. [25] proposed to analyze executable files as a stream of bytes, and discuss several statistical properties to determine the randomness level of each stream. Finally, we considered some dynamic unpackers ([13], [43]) that rely on static features to understand if the unpacking procedure is correctly terminated. Unfortunately, while some of these works included custom packers in their experiments, none of the aforementioned proposals considered the presence of samples packed with low-entropy schemes.

We are only aware of two exceptions to this rule. In 2012, Ugarte et al.[49] performed several experiments which included some samples of the Zeus botnet, one of the first families that adopted a low entropy packing scheme. However, the approach proposed by the authors is tailored to the single specific case documented in their paper and would fail to address other common low-entropy techniques. Therefore, we did not include this technique in our study.

Raphel et al. [35], instead, focused their study on the use of XOR-based encoders. In this work, XOR encryption is recognized as a form of obfuscation mainly used to encrypt small parts of the code like shellcodes. The idea was to refine the use of entropy to recognize samples that adopted a XOR-based scheme. Mainly, their approach relies on 5 steps: (i) extraction of fragments from files; (ii) computation of entropy for each fragment; (iii) concatenation of fragments; (iv) computation of entropy for each concatenated fragment; and (v) construction of a similarity distance matrix based on the previously computed values for each file pair in the dataset. Like in the previous case, this solution targets a very specific problem and is not directly applicable to the type of packers we are studying because the authors designed it with the purpose of detecting small portions of encrypted code (essentially schellcodes). Anyway, we considered this approach in our evaluation.

To summarize, we can group the proposed features in six different families:

- PE Structure: values extracted from the PE headers.
- Heuristics: features produced as a result of common knowledge about characteristics of packed PE files.
- Opcodes: sequences of assembly instructions extracted from the executable sections.
- N-grams: sequences of N bytes extracted from the entire file or some of its sections.
- Statistical features: evaluation of statistical properties about the randomness of a sequence.
- Entropy features: features based on the computation of entropy with respect to some areas of the file (sections, overall file, sliding windows).

Table VI summarizes all the presented static analysis approaches and lists the categories of features as well as how the authors constructed the dataset they used in their experiments.

B. Evaluation of Static Features on Low-entropy Packers

In this section, we evaluate the reliability of the previously discussed static analysis techniques in detecting packed samples.

To assess this, we use our dataset of 40,909 samples (i.e., all running programs minus the samples with hidden high-entropy data, because those can be detected with proper entropy analysis). For the same reason we also decided not to include in the dataset any *high-entropy* packed samples, i.e., those using traditional packing schemes such as UPX, ASPack, and Armadillo. In summary, our dataset contained $14,583$ samples packed with *low-entropy schemes* and $26,326$ *not packed* samples. From now on we will refer to this subset of samples as the *ML dataset*.

For each malware in the *ML dataset*, we extracted all the features adopted by the 15 state-of-the-art approaches discussed in the previous section, and summarized in Table VI. We refer each approach through an index i, where $0 <= i <= 14$. The i^{th} approach applies several ML algorithms using an input vector of n features $V(i) = [f_{0_i}, ..., f_{n_i}]$ where f_{m_i} represents the m^{th} feature of the $i-th$ approach, with $f_{m_i} \in \mathbb{R}$. To simplify the experimental setup, we joined the feature vectors $V(i)$, for $i = 0, .., 14$, in a single vector $W = [V(0)|...|V(14)]$. If two or more approaches rely on the same feature, we considered it only once. We point out that the vector W includes the entropy features as well. To verify if entropy still plays a role as discerning metric, we define the

TABLE VI. OVERVIEW OF PREVIOUS APPROACHES

Paper	Proposed approach	Features	Dataset construction
Lyda et al. [26]	Entropy analysis	Entropy	*Not Packed*: benign executables *Packed*: Packers manually applied to benign executables
Choi et al. [14]	Machine learning	PE Structure Heuristics	*Not Packed*: benign executables and PE files from AV manually analysed *Packed*: benign executables and PE files from AV vendors manually analysed
Perdisci et al. (1) [33]	Machine learning	PE Structure Heuristics Entropy	*Not Packed*: benign executables *Packed*: malware from MALFEASE project and application of a set of packers benign executables
Perdisci et al. (2) [32]	Machine learning	Same as [33] N-grams	*Not Packed*: malware from MALFEASE project filtered with state-of-the-art unpackers ([37], [22]) *Packed*: malware from MALFEASE and benign executables
Santos et al. [39]	Machine learning	PE Structure Heuristics Entropy	*Not packed*: benign executables and malware from VxHeavens *Packed*: Variants of the 'Zeus' family and application of a set of packers to the benign executables
Ugarte-Pedrero et al. [48]	Machine learning	Same as [39] N-grams	*Not packed*: benign and malicious executables filtered by PEid, entropy analysis, IAT entries, imported dlls and ratio of standard sections *Packed*: Application of a set of packers to the benign executables, malware reported by PEid as not packed, malware from 'Zeus' family
Devi et al. [16]	Machine learning	PE Structure Entropy	*Not packed*: benign executables *Packed*: Application of UPX to benign executables
Ugarte-Pedrero et al. [49]	Entropy analysis	Entropy	*Not packed*: benign executables and malicious samples taken from VxHeavens and checked with PEid *Packed*: application of a set of packers to benign samples and malware from Zeus family
Shafiq et al. [41]	Machine learning	PE Structure	*Not packed*: Benign executables and malware from VxHeavens/Malfease filtered by PEid *Packed*: malware from VxHeavens/Malfease detected as packed by PEid
Raphel et al. [35]	Entropy analysis	Entropy Statistical	*Not packed*: benign executables *Packed*: Application of a set of packers/encoders to the benign executables
Lim et al. [25]	Machine learning	Entropy Statistical	*Not packed*: benign executables and evaluation of similarity for adding other binaries *Packed*: Benign and malicious samples tested with [47]
Han et al. [19]	Entropy analysis	Entropy	*Not packed*: benign executables *Packed*: malicious samples from honeypot
Treadwell et al. [46]	Machine learning	PE Structure	*Not packed*: Windows System files and malicious samples filtered by PEid *Packed*: malicious samples filtered by PEid
Arora et al. [9]	Machine learning	PE Structure Heuristics	*Not packed*: benign executables and malicious samples from honeypots *Packed*: Malicious samples tested and emulated with [45]
Jacob et al. [20]	Machine learning	Opcodes	*Not packed*: Benign executables fetched from Windows Installation and unpacked malware *Packed*: Executables packed with the most popular off-the-shelf packers

vector $\tilde{W}$ as the feature vector containing all the features of W except for all the entropy features.

We split the *ML dataset* into *train set* (TrS) and *test set* (TeS) and we run the classifiers on different subsets of TrS and TeS. TeS is composed by a subset TeS_{packed} of packed samples, and a complementary subset $TeS_{notPacked}$ of not packed samples, s.t. $TeS_{packed} \cup TeS_{notPacked} = TeS$.

We indicate FP and FN the sets of false positives and false negatives samples, respectively. The set FP contains the not packed samples which are classified as packed, while FN contains the packed samples which have been classified as not packed. In particular, we focus on the number of errors the classifiers make respectively on packed and not packed samples:

$$Err_{notPacked} = \frac{|FP|}{|TeS_{notPacked}|} \quad (1)$$

$$Err_{packed} = \frac{|FN|}{|TeS_{packed}|} \quad (2)$$

We show our results in Table VII. For each classifier we report the ratio between *training* and *testing* sets, and the $Err_{notPacked}$ and Err_{packed} obtained by using the two feature vectors W and $\tilde{W}$ (i.e., with and without entropy features). Our experiments, summarized in Table VII, indicate that none of the classifiers provide a high level of accuracy – with the best model implementing MLP and achieving the 11.91% as false negatives rate but also the 6.89% as false negatives rate. It is worth noting that in most of the cases, the classifiers show a high Err_{packed} ratio, which means that a significant number of packed binaries are classified as not packed. This suggests low entropy schemes can effectively be used by malware authors to bypass classifiers based on static features alone. For instance, we noticed that several files have PE headers appearing perfectly normal (sections named with standard names, entry point correctly located inside `.text`, a high number of entries in the `IAT`, etc.). While this somehow decreases the level of obfuscation provided by traditional packing schemes, it still succeed in protecting the application code against automated static analysis routines.

TABLE VII. RESULTS OF ML EXPERIMENTS

Classifier	Training-Testing	$Err_{notPacked}(W)$	$Err_{packed}(W)$	$Err_{notPacked}(\tilde{W})$	$Err_{packed}(\tilde{W})$
SVM	75%-25%	4.43%	25.01%	4.12%	24.57%
	50%-50%	4.31%	28.41%	3.97%	26.20%
	25%-75%	4.44%	32.01%	4.11%	29.85%
MLP	75%-25%	6.34%	12.70%	5.86%	12.15%
	50%-50%	6.87%	16.14%	6.24%	14.73%
	25%-75%	6.89%	11.91%	6.33%	12.93%
RF	75%-25%	0.20%	32.77%	0.23%	31.54%
	50%-50%	0.18%	29.46%	0.20%	28.46%
	25%-75%	0.21%	28.84%	0.20%	26.83%

With this we do not want to say that static features used in previous studies are useless. In fact, they do much better than entropy alone. However, in presence of low-entropy packed samples all classifiers trained on these features perform quite poorly, and certainly far worse than what was reported in previous experiments. For instance the authors of [19] claim to reach the 0.0% as false positive rate and 2.5% as false negative rate by only relying over entropy metrics while in works that employ ML features, the authors declare to obtain a false positive rate of 0.8% ([39]).

VII. CASE STUDIES

In this section, we discuss in more details three malware samples that implement low entropy packing techniques. We also investigate why they are (or are not) detected by the features introduced in the previous section. We hope that this can help to understand better the internals of real-world low-entropy packing schemes and the reason why malware writers adopt them.

A. Case I: Simple XOR Encryption

For the first case study, we look at a sample[6] that belongs to the *berbew* family. By looking at the code located at the application's entry point, it is easy to identify a simple XOR encryption algorithm that applies a fixed 4-bytes key to decrypt in place the `.text` section. The hardcoded decryption key is `0x6d02676d`. Since the first and last digits are the same, the encryption only raises the overall entropy of the packed code to 6.9; it is reasonable to believe that this repetition was a conscious decision introduced to lower the entropy. The malware author also padded the code of its `.aciof` section with a large number of `0x90` bytes (corresponding to the `nop` x86 instruction) – likely for the same purpose.

Although this sample can evade any entropy-based check, it is easily detectable by using other static features. In fact, this PE file contains several anomalous values – including the `RWX` permissions of the `.text` section and the non-standard name of the section `.aciof`.

While this scheme is relatively simple and not particularly interesting from a research point of view, we decided to include it in our case studies because it is representative of the vast majority of low-entropy techniques we observed in our dataset and because our Scheme Classifier could automatically categorize it. Next, we are going to present two more complex cases that required manual investigation to be classified.

[6] `md5= 7186708dd7a1b0dbf9294909679ec30b`

B. Case II: Transposition Scheme

Our second example is a sample[7] that belongs to the arsenal of a cyber-espionage group dubbed GreyEnergy [5]. Since 2015, this malware has been used as part of attacks against energy companies and other high-value targets in Ukraine and Poland. Most specifically, the binary is a *loader*, i.e., the code in charge of stealthily loading the real malware into the target system.

The sample hides the packed data in the `.text` section, within the range `[0x1000, 0x211ff]`, for a total of `0x21200` bytes. This packed data, `PackedSrc` from now on, has an entropy of 6.59, and it contains, in a scattered disposition, all the data that is necessary to create a valid PE file. A simplified algorithm of the packing scheme is presented in Algorithm 1 (the original technique also involved operations between integers of different sizes that we omit for brevity). The unpacker uses five memory regions dynamically allocated (using the `VirtualAlloc` API) as reported in Table VIII.

TABLE VIII. GREYENERGY DYNAMICALLY ALLOCATED MEMORY REGIONS

Name	Size [Byte]	Permissions
Mdst	0x20200	RW
Moffsets	0x808	RW
Mtmp1	0x200	RW
Mtmp2	0x200	RW
Mexe	0x24000	RWX

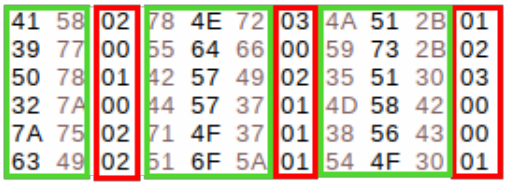
The first step is a call to `init(Moffsets, n)` to initialize the `Moffsets` memory region (line 1 and 6), that represents an array of integers. This array is initialized with `n` integers s.t. $Moffsets = \{\forall i = 0...n | 0 \leq Moffsets[i] < n\}$ and every number in range $[0, n]$ is contained in the `Moffsets`. Those properties allow the unpacker to later use the `Moffsets` region as a lookup table that implements a bijective function $f : [0, n] \rightarrow [0, n]$.

The algorithm then splits the `PackedSrc` and `Mdst` in 514 chunks of 256 bytes each and it copies every chunk from `PackedSrc` to `Mdst` (line 4), but not consecutively: it uses the `Moffsets` table (initialized in line 1) for looking up the proper offset in the destination buffer (line 3).

After that, it splits again `Mdst` in chunks (this time 257 chunks of 512 bytes) and each chunk is copied into `Mtmp1` (line 8); then, one byte at a time, it is copied into `Mtmp2`

[7] `md5= 7a7103a5fc1cf7c4b6eef1a6935554b7`

Fig. 6. Pattern stored inside the .rsrc section



by using the offsets specified in the re-initialized (in line 6) `Moffsets` table (line 10). At the end, `Mtmp2` is directly copied into `Mdst` (lines 12-14) and in turn into the executable region `Mexe` (line 16).

When the unpacking procedure is completed, the sample parses the unpacked PE in the `Mexe` memory, and loads (using the library function `LoadLibraryA`) every `dll` requested in the *Import Table*. Then it modifies the *Process Environment Block* structure's `ImageBaseAddress` field [8], so that it points at the very beginning of the unpacked PE file. Finally, it jumps to the entry point of the unpacked PE[9].

The remarkable achievement of this scheme is that the byte distribution, and consequently the entropy, of the packed and unpacked regions are identical. Moreover, from the static analysis point of view, this sample is undetectable using both signatures and ML techniques, among the ones described in the previous sections.

```
1  init (Moffsets, 0x202);
2  for (i = 0, j = 0; i ¡ 0x202; i += 1, j += 0x100) do
3  |  offset = Moffsets [i] * 0x100;
4  |  memcpy (Mdst [offset], PackedSrc [j], 0x100);
5  end
6  init (Moffsets, 0x200);
7  for (i = 0; i ¡ 0x8080; i += 0x80) do
8  |  memcpy (Mtmp1, Mdst [i], 0x200);
9  |  for (j = 0; j ¡ 0x200; j += 1) do
10 |  |  Mtmp2 [Moffsets [j]] = Mtmp1 [j];
11 |  end
12 |  for (k = 0; k ¡ 0x200; k += 1) do
13 |  |  Mdst [i+k] = Mtmp2 [k];
14 |  end
15 end
16 memcpy (Mexe, Mdst, 0x20200);
```

Algorithm 1: GreyEnergy unpacking scheme

C. Case III: Custom Encoding

Our final sample[10] uses two layers of packing. The second (deepest) layer, relies on a traditional XOR encryption scheme (with an 8-bytes key) and ROR/ROL loops that produced packed data with high entropy. To mask this fact, the malware authors added a first layer of packing that reduced the entropy from 7.63 to 6.57 by adopting a custom encoding scheme.

[8]The ImageBaseAddress field contains the address where the legitimate process executable is loaded.

[9]md5= ab8df9b7389ae890c3396a238bdc4606

[10]md5= c03bc642c5a49c55efb2d07a7272af2e

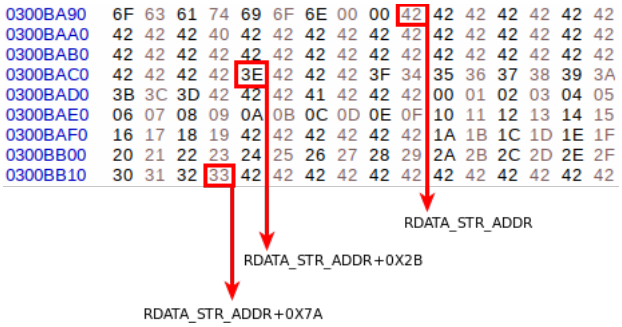


Fig. 7. The string `0x0300ba99` in `.rdata` section

```
1  i = 0;
2  res = 0;
3  for addr ← RSRC_START to RSRC_END do
4  |  offset = readByte(addr);
5  |  byte = readByte(RDATA_STR + offset);
6  |  if byte != 0x42 then
7  |  |  res = res << 6;
8  |  |  res = res + byte;
9  |  |  i = i + 1;
10 |  end
11 |  if i == 4 then
12 |  |  writeToMemory(res);
13 |  |  res = 0;
14 |  |  i = 0;
15 |  end
16 |  addr = addr + 1
17 end
```

Algorithm 2: Pseudocode of the first layer

The first layer relies on the content of two sections: `.rsrc` and `.rdata`. Figure 6 shows some bytes extracted from the `.rsrc` section. It is clear that the data consists of sequences of three bytes (highlighted by the green rectangles) within the range [`0x2b`, `0x7a`], separated by a single byte [`0x00`, `0x03`] (highlighted by the red rectangle in the image).

The `.rdata` section contains a buffer filled with some characters without a particular meaning (mainly the "*B*" character, `0x42` in hexadecimal). However, from the offset `0x2b` to `0x7a` (as shown in Figure 7), the buffer contains bytes ranging from `0x00` to `0x3f`.

Algorithm 2 summarizes the unpacking procedure in pseudo-code. The code loops through all values in the `.rsrc` Section (line 3) and uses each byte as offset to access the string (lines 4-5). If the value of the read byte is `0x42`, the algorithm moves to the next byte (line 6), while others are combined four at a time by adding each value to the previous one shifted by six bits (lines 7-8). The result is finally written to another memory region (line 12), before resetting the counter and restarting the loop (lines 13-14).

The PE structure of this file does not contain any anomaly, and the above-described custom scheme (that uses the same symbols of the Base64 scheme) is able to hide the packed code from n-grams and opcodes analysis; therefore this sample evades all the previously described ML techniques.

VIII. Conclusions

In this work, we conducted a set of experiments on real-world malware to demonstrate that existing static approaches fail to take into consideration the threat represented by low-entropy packed malware and that this phenomenon is relevant enough that cannot be ignored when designing malware experiments. Although previous works [49], [35] have discussed the existence of low-entropy packing schemes as case studies, our work is the first to study this phenomenon in depth, and to measure the prevalence of this technique over a large dataset.

While it might be true that high-entropy file are often packed, our experiments show that the opposite is not correct – i.e., the fact that the entropy is low is not sufficient to conclude that the file is most likely not packed. This is important as many studies and tools still use the entropy alone to classify a sample as packed or not.

The results of our large scale dynamic analysis performed on $46,295$ samples shows that 31.5% of low-entropy files were packed, proving that this type of malware represents an actual and widespread reality. As final proof of our results, we have also analyzed a reduced set of 476 APT-linked (Advanced Persistent Threat) malware that represent state of the art for complex attacks. We found that in this context the phenomenon of low-entropy packed malware occurs with a frequency of the 15%. In Section IV we have categorized how such schemes keep their entropy low and the frequency in which this technique is adopted in the wild.

We then investigated why actual static analysis techniques are unable to detect the presence of low-entropy packing. We have studied two kinds of approaches: those based on signature/heuristic in Section V, and those based on machine learning in Section VI. On the one hand, signatures are just well suited to detect the presence of common off-the-shelf packers, while heuristics generate a large number of false alerts on non-packed samples. On the other hand, we evaluated the performance of static feature-based classifiers, when entropy is no longer a reliable way to detect packers. Unfortunately, our experiments show that this is not the case as even the best classifier was able to detect only 70% of the packed samples in our dataset. Our results show that the accuracy of these classifiers degrades drastically in the presence of low entropy packers, which means that the results reported in the past relied significantly on the entropy to discern between packed and not packed files, and that the datasets employed may have not correctly represented the low-entropy packers that we found in the wild. Moreover, the machine learning experiments tell us that the static features proposed so far are inadequate and needs to be extended to allow for a accurate classification of packed samples. This does not mean that these approaches, including simple entropy-based measurements, must be abandoned. Instead, our work emphasizes the need for new solutions to this open problem, and that the existence of low-entropy packing must be considered in future experiments conducted by researchers and practitioners.

Moreover, we share[11] the hashes of the samples, labeled with the corresponding category by Packer Detector (described in Section III-A) in the hope that other researchers will use it as a basis for further studies.

[11] http://www.s3.eurecom.fr/datasets/low_entropy_malware/LEM_dataset.7z

Acknowledgment

The authors would like to thank Enrico Russo of the University of Genoa; his expertise has allowed us to speed up the analysis significantly.

This research was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 771844 – BitCrumbs).

References

[1] Apt and cybercriminal campaign collection. https://github.com/CyberMonitor/APT_CyberCriminal_Campagin_Collections, Accessed February 17, 2020.

[2] Capstone. http://www.capstone-engine.org/, Accessed February 17, 2020.

[3] Detect-it-easy signatures. https://github.com/horsicq/Detect-It-Easy, Accessed February 17, 2020.

[4] Manalyze. https://github.com/JusticeRage/Manalyze, Accessed February 17, 2020.

[5] The new malicious software grayenergy. https://cert.gov.ua/news/45, Accessed February 17, 2020. Accessed February 17, 2020, Ukrainian language.

[6] pefile. https://github.com/erocarrera/pefile, Accessed February 17, 2020.

[7] Peid. https://www.aldeid.com/wiki/PEiD, Accessed February 17, 2020.

[8] Virus total. https://www.virustotal.com/, Accessed February 17, 2020.

[9] Rohit Arora, Anishka Singh, Himanshu Pareek, and Usha Rani Edara. A heuristics-based static analysis approach for detecting packed pe binaries. *International Journal of Security and Its Applications*, 7(5):257–268, 2013.

[10] M. Baig, P. Zavarsky, R. Ruhl, and D. Lindskog. The study of evasion of packed pe from static detection. In *World Congress on Internet Security (WorldCIS-2012)*, pages 99–104, June 2012.

[11] C. Barria, D. Cordero, C. Cubillos, and R. Osses. Obfuscation procedure based in dead code insertion into crypter. In *2016 6th International Conference on Computers Communications and Control (ICCCC)*, pages 23–29, May 2016.

[12] Jean-Marie Borello and Ludovic Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, Aug 2008.

[13] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards paving the way for large-scale windows malware analysis: generic binary unpacking with orders-of-magnitude performance boost. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 395–411. ACM, 2018.

[14] Yang-Seo Choi, Ik kyun Kim, Jin-Tae Oh, and Jae cheol Ryou. Pe file header analysis-based packed pe file detection technique (phad). *International Symposium on Computer Science and its Applications*, pages 28–31, 2008.

[15] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[16] Dhruwajita Devi and Sukumar Nandi. Pe file features in detection of packed executables. *International Journal of Computer Theory and Engineering*, 4(3):476, 2012.

[17] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2015.

[18] Behrouz A Forouzan. *Cryptography & network security*. McGraw-Hill, Inc., 2007.

[19] Seung-Won Han and Sang-Jin Lee. Packed pe file detection for malware forensics. *The KIPS Transactions: PartC*, 16(5):555–562, 2009.

[20] Grégoire Jacob, Paolo Milani Comparetti, Matthias Neugschwandtner, Christopher Kruegel, and Giovanni Vigna. A static, packer-agnostic filter to detect similar malware samples. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 102–122. Springer, 2012.

[21] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm–software protection for the masses. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 3–9. IEEE, 2015.

[22] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *In Proc. ACM Workshop Recurring Malcode (WORM*, pages 46–53. ACM, 2007.

[23] Eric Lafortune. Proguard. https://sourceforge.net/projects/proguard/, 2004. Accessed February 17, 2020.

[24] Michael Ligh, Steven Adair, Blake Hartstein, and Matthew Richard. *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code*. Wiley Publishing, 2010.

[25] Charles Lim, Kalamullah Ramli, Yohanes Syailendra Kotualubun, et al. Mal-flux: Rendering hidden code of packed binary executable. *Digital Investigation*, 28:83–95, 2019.

[26] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2), 2007.

[27] Tímea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica*, 30:3–19, 08 2009.

[28] Microsoft. Microsoft store top free apps. https://www.microsoft.com/en-us/store/top-free/apps/pc. Accessed February 17, 2020.

[29] Maik Morgenstern and Hendrik Pilz. Useful and useless statistics about viruses and anti-virus programs. In *Proceedings of the CARO Workshop*, 2010.

[30] Ginger Myles and Christian Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, Apr 2006.

[31] P. OKane, S. Sezer, and K. McLaughlin. Obfuscation: The hidden malware. *IEEE Security Privacy*, 9(5):41–47, Sep. 2011.

[32] R. Perdisci, A. Lanzi, and W. Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 301–310, Dec 2008.

[33] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Classification of packed executables for accurate computer virus detection. *Pattern recognition letters*, 29(14):1941–1946, 2008.

[34] B. Rahbarinia, M. Balduzzi, and R. Perdisci. Exploring the long tail of (malicious) software downloads. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 391–402, June 2017.

[35] Jithu Raphel and P. Vinod. Information theoretic method for classification of packed and encoded files. In *Proceedings of the 8th International Conference on Security of Information and Networks*, SIN '15, pages 296–303, New York, NY, USA, 2015. ACM.

[36] Kevin A. Roundy and Barton P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1):4:1–4:32, July 2013.

[37] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 289–300, Dec 2006.

[38] M. Saleh, E. P. Ratazzi, and S. Xu. Instructions-based detection of sophisticated obfuscation and packing. In *2014 IEEE Military Communications Conference*, pages 1–6, Oct 2014.

[39] Igor Santos, Xabier Ugarte-Pedrero, Borja Sanz, Carlos Laorden, and Pablo G Bringas. Collective classification for packed executable identification. In *Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference*, pages 23–30. ACM, 2011.

[40] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[41] M. Zubair Shafiq, S. Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In Engin Kirda, Somesh Jha, and Davide Balzarotti, editors, *Recent Advances in Intrusion Detection*, pages 121–141, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[42] Muhammad Zubair Shafiq, S. Momina Tabish, and Muddassar Farooq. Pe-probe : Leveraging packer detection and structural information to detect malicious portable executables. 2009.

[43] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A framework for enabling static malware analysis. In Sushil Jajodia and Javier Lopez, editors, *Computer Security - ESORICS 2008*, pages 481–500, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[44] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[45] Adrian E Stepan. Defeating polymorphism: beyond emulation. In *Proceedings of the Virus Bulletin International Conference*, 2005.

[46] Scott Treadwell and Mian Zhou. A heuristic approach for detection of obfuscated malware. pages 291–299, 01 2009.

[47] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 659–673. IEEE, 2015.

[48] Xabier Ugarte-Pedrero, Igor Santos, Iván García-Ferreira, Sergio Huerta, Borja Sanz, and Pablo G Bringas. On the adoption of anomaly detection for packed executable filtering. *Computers & Security*, 43:126–144, 2014.

[49] Xabier Ugarte-Pedrero, Igor Santos, Borja Sanz, Carlos Laorden, and Pablo Garcia Bringas. Countering entropy measure attacks on packed software detection. In *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*, pages 164–168. IEEE, 2012.

[50] W. Yan, Z. Zhang, and N. Ansari. Revealing packed malware. *IEEE Security Privacy*, 6(5):65–69, Sep. 2008.