File Packing from the Malware Perspective: Techniques, Analysis Approaches, and Directions for Enhancements

TRIVIKRAM MURALIDHARAN, Malware Lab, Cyber Security Research Center, Ben-Gurion University of the Negev, Beer-Sheva, Israel and Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, Beer-Sheva, Israel
AVIAD COHEN, Malware Lab, Cyber Security Research Center, Ben-Gurion University of the Negev, Beer-Sheva, Israel
NOA GERSON, Malware Lab, Cyber Security Research Center, Ben-Gurion University of the Negev, Beer-Sheva, Israel
NIR NISSIM, Malware Lab, Cyber Security Research Center, Ben-Gurion University of the Negev, Beer-Sheva, Israel and Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, Beer-Sheva, Israel

With the growing sophistication of malware, the need to devise improved malware detection schemes is crucial. The packing of executable files, which is one of the most common techniques for code protection, has been repurposed for code obfuscation by malware authors as a means of evading malware detectors (mainly static analysis-based detectors). This paper provides statistics on the use of packers based on an extensive analysis of 24,000 PE files (both malicious and benign files) for the past 10 years, which allowed us to observe trends in packing use during that time and showed that packing is still widely used in malware. This paper then surveys 23 methods proposed in academic research for the detection and classification of packed **portable executable (PE)** files and highlights various trends in malware packing. The paper highlights the differences between the methods and their abilities to detect and identify various aspects of packing. A taxonomy is presented, classifying the methods as static, dynamic, and hybrid analysis-based methods. The paper also sheds light on the increasing role of machine learning methods in the development of modern packing detection methods. We analyzed and mapped the different packing methods and identified which of them can be countered by the detection methods surveyed in this paper.

CCS Concepts: • **Security and privacy → Malware and its mitigation**;

Additional Key Words and Phrases: Packing, packer, identification, analysis, detection, malware, PE file

ACM Reference format:
Trivikram Muralidharan, Aviad Cohen, Noa Gerson, and Nir Nissim. 2022. File Packing from the Malware Perspective: Techniques, Analysis Approaches, and Directions for Enhancements. *ACM Comput. Surv.* 55, 5, Article 108 (December 2022), 45 pages.
https://doi.org/10.1145/3530810

Authors' addresses: T. Muralidharan, N. Gerson, and N. Nissim, Malware Lab, Cyber Security Research Center, Ben-Gurion University of the Negev, Beer-Sheva, Israel and Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, Beer-Sheva 8410501, Israel; emails: muralidh@post.bgu.ac.il, noagolan65@gmail.com, nirni@bgu.ac.il; A. Cohen, Malware Lab, Cyber Security Research Center, Ben-Gurion University of the Negev, Beer-Sheva 8410501, Israel; email: aviadjo@gmail.com.

0360-0300/2022/12-ART108 $15.00
https://doi.org/10.1145/3530810

1 INTRODUCTION

In recent times, a trend of increasing complexity and obfuscation has been observed in malware and entirely new generations of malware families have come into existence, threatening modern computer systems and cybersecurity. Malware not only violates the security and privacy of computer users, but it can also result in significant financial loss [1] and in the denial of critical services. Organizations have become increasingly aware of the potential harm that can be caused by malware, and thus invest a significant amount of resources in protecting themselves from potential malware attacks. A vast history of wreaking havoc by malware has educated these organizations that the price of recovering from an attack and repairing the damage caused would be costly and may even be priceless if sensitive or confidential data has been ex-filtrated or permanently deleted or encrypted.

Executable packing is the most commonly used technique for the protection of proprietary code and there is a wide variety of open-source and commercial packers available [2]. In addition to this legitimate use, code packing is also the most common obfuscation method used by malware authors to disrupt malware detection that is based on static analysis. According to the literature, the majority of malware on the Wildlist [106] are packed [3]. Research performed in 2010 [4] found that 35% of malicious code was packed by a custom packer. In addition, statistics published in 2013 showed that over 80% of viruses (malware) are packed, while 50% of new malware samples are simply repacked versions of existing malware [1]. In 2016, Symantec [107] reported that the ratio of packers used by Android malware increased from 10% in November 2015 to 25% in August 2016.

We introduce newer statistics that shed additional light on the packing trends, based on our scanning and analysis of PEs in the comprehensive VirusTotal malware repository. The data on which the statistics and figures presented below are based was obtained by running a **VirusTotal (VT)** intelligence query on 2,000 randomly accessed files for each year from 2010 to 2021. Our analysis was based on a total of 24,000 files. Of the 2,000 randomly selected files retrieved from VT for each year, there were, on average, 25% benign files and 75% malicious files. Figure 1 shows the trends of packed and unpacked malware in the wild over the years (from 2010 to 2021), and Figure 2 shows the trends in the packers commonly used to pack files during the same time period.

As can be seen in Figure 1, the trend of packing executables is not dying out. Between 2010 and 2021, 21.35% (on average) of files were packed. There seems to have been a period in which there was less packing between 2017 and 2020 (where 11.6% of files, on average, were packed), but this can be partially attributed to the effect of a growing trend during that time of launching attacks via documents (rather than via PEs), such as DOCX files [78, 124, 125], PDF files [121, 126], JPEG images [122], and more. The figure also breaks down the packing further, presenting the percentage of packing of malicious and benign files; as can be seen, the percentage of malicious files is usually higher than that of benign files, with the difference between the percentages reaching up to 22% in some of the years.

The packers most commonly used to pack malicious files during the same time period are presented in Figure 2. As can be seen, UPX seems to be the most popular packer used until 2018. This can be attributed to the fact that it is free and easy to; moreover, UPX has not been blacklisted by anti-virus vendors.

However, the figure shows that in 2018, a Delphi-based packer, BobSoft Mini Delphi, became the most popular choice for packing malicious files [120]. The reason for this is twofold. First, by checking for various artifacts and signs that it is in an analysis environment (such as unnatural or non-existent mouse movements), Bobsoft facilitates runtime evasion if it is executed in sandboxed environments, and second, UPX pales in comparison to a Delphi-based packer like Bobsoft, as it mainly functions as a compressor and does not have the same capacity for runtime evasion.

Fig. 1. Packing trends based on a random sample of 24,000 PE EXE files (2,000 each year) on VT.

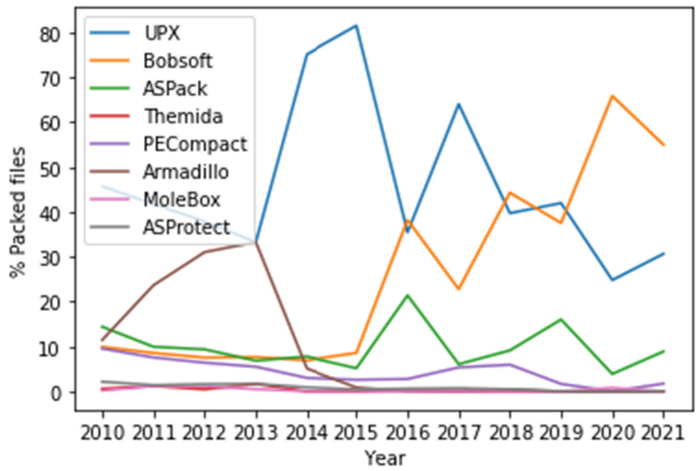


Fig. 2. Share of packers commonly used to pack malicious files.

Protectors such as ASProtect, Themida, and MoleBox seem to be less popular, since they are either blacklisted or commercial, paid services. ASPack, PECompact, and Armadillo seem to be good intermediate choices for packing.

The number of newly created packers used for packing malware is rising. Packer writers create new packers, modify existing packers, and use advanced strategies, such as anti-unpacker and evasion methods [5], making it more complicated, expensive, and time-consuming for security researchers to develop conventional static analysis methods in order to detect malware in packed executable files [3]. In addition, if malware has been packed, repacked, or multi-layer packed, its detection through a signature-based anti-malware solution (e.g., antivirus) is impossible [6]. This presents the immediate need for an effective static analysis-based detection solution that isn't conventional in its approach. The identification and classification of packing techniques applied on executable files has significant importance as it helps reveal the true nature of suspicious executable files [1]. Thus, security solutions, such as anti-malware systems, must be able to deal with

(identify or classify) a large and diverse number of unique packers and must be able to adapt to entirely new sets/families of packers, since new packing algorithms are created daily [6].

Our contributions in this paper are as follows:

- We provide extensive background on packing principles, unpacking (the process of de-obfuscating a packed file) and anti-unpacking techniques, and the motivation for using packing.
- We survey the most relevant and recent academic papers aimed at the detection of packed executable files. In particular, we surveyed several significant studies pertaining to PE file detection using static analysis, dynamic analysis, or both, and have mapped the studies based on whether Machine Learning (ML) methods were used for the purpose of devising the solutions they proposed.
- We present a taxonomy of packing methods which is based on their level of complexity.
- We provide a comprehensive comparison of existing methods and tools proposed in academic research based on their objectives and capabilities.
- We provide a comprehensive comparison of existing methods and tools proposed in academic research based on their objectives, requirements, capabilities, and limitations.
- We propose a concrete direction and method for packing identification and classification based on an advanced machine learning method that utilizes multivariate time-series data (MTSD) analysis of API calls evoked while executing a packed file.

This paper focuses on packed **Portable Executable (PE)** files (*.exe) used on the Microsoft Windows **operating system (OS)**, since it is the most commonly used OS today, widely used by individuals and organizations alike [7]. Currently, packer writers often focus their efforts on Windows based systems, mainly targeting PE files [8, 9]. It should be noted that the Linux OS is also widely used, but the domain of packed ELF files is less prevalent and therefore the attacks and malware aimed at Linux are far less popular and varied than those targeting the Windows OS.

The rest of the paper is organized as follows. In Section 2, we provide the necessary background for this area of research. In Section 3, we review some popular packing techniques, and we review published academic research and existing tools that perform unpacking and anti-unpacking. In Section 4, we review methods that are meant for detecting and/or classifying packing. In Section 5, we summarize the main insights generated from this survey. In Section 6, we discuss and conclude this survey.

2 BACKGROUND

In this section, we provide the information needed for the reader to understand this survey's domain and discuss packing fundamentals and the implications of packing.

2.1 PE File Structure (Windows OS)

In order to understand the nature of the content of a packed file, we must first understand the different pieces of information an executable file contains. The portable executable file format is a data structure that is responsible for containing all of the information needed for the Windows OS loader to execute the wrapped executable code. This format is used in 32-bit and 64-bit Windows OS executables, object code, and **dynamic-link libraries (DLLs)**. The PE file structure includes dynamic library references for linking, **application programming interfaces (APIs)**, export and import tables, resource management data, etc. Typically, a non-packed PE file contains a clear set of standard sections and names [10].

A PE file's structure begins with an MS-DOS header which is followed by a PE header vis-a-vis, a PE signature, COFF file header, and an optional header [11]. The PE header is comprised

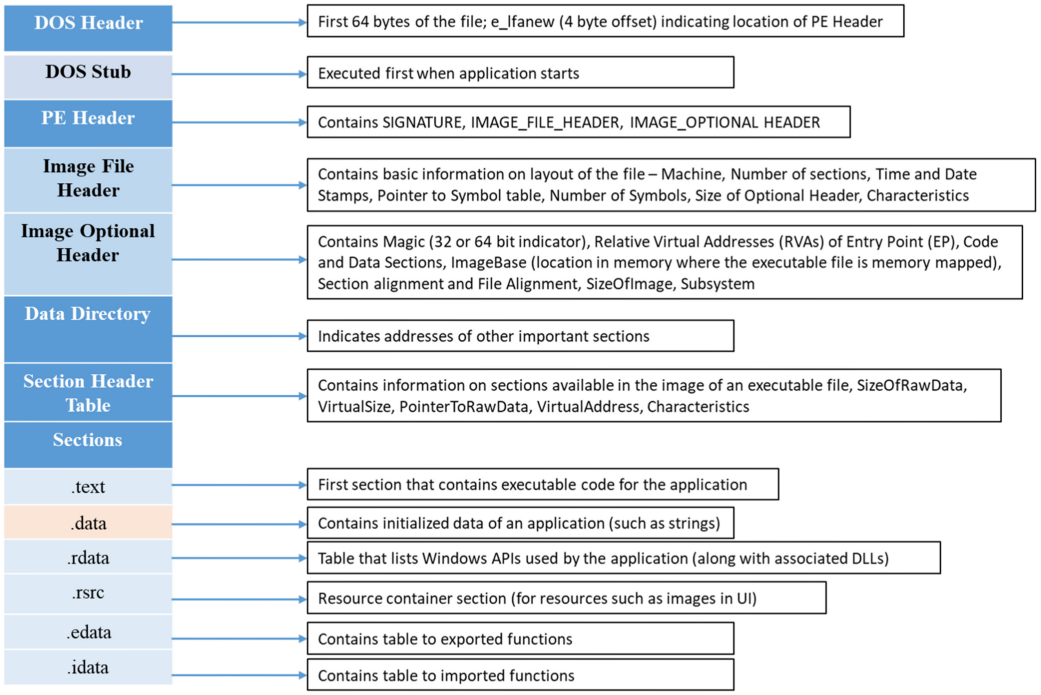


Fig. 3. PE file structure.

of three parts: the file header; the optional header and section header; and the code and data sections [12]. The PE header includes information about the code, type of application, required library functions, and space requirements [13]. The PE header instructs the operating system on how to map the executable in memory. The PE header contains valuable information for malware analysis [3], e.g., metadata and basic information about the file itself. Figure 3 illustrates the PE file structure.

Following the PE header, there are two main sections: the import and export sections. Imports are functions used by a program, which are actually located in another program, such as a **DLL (dynamic-link library)**, which contain code libraries. Code libraries are connected to the main program by linking, i.e., importing functions needed by programs at runtime. Program writers link imports to their programs, in order to avoid reimplementing certain functionality in multiple programs (reusability). For instance, code libraries that contain functions common to many programs can be connected to the chief executable by linking. They can be linked statically during runtime or dynamically. Knowledge of the way in which the library code is linked is essential to understanding malware, because the information in the PE file header is determined by how the library code has been linked. In addition to imports, DLLs and EXEs (executable files) export functions in order to communicate with another program's code. The PE file contains information about the functions the file exports. In most cases, programmers assign meaningful names to their functions. Thus, the names of the functions in the import or export section can provide a good idea of what the executable does. The number of DLLs and imported functions of a packed program is much smaller than the number found in a benign program, as described by [13, 14].

The code segment of a Windows PE file is not standalone like a plain assembly instruction. A lot of information might be required by that code segment in order to be executed, and that

information might be present in the other nine different sections of the file. Some applications may have only few sections, while others may have more [15]. The PE file structure mainly consists of the sections (called standard sections) as seen in Figure 3.

The data items in the PE file structure are the key characteristics for identifying whether or not an executable is packed [12]. Valuable features can be extracted from a PE file, including the number of standard and nonstandard sections, the number of executable sections, the number of readable/writable/executable sections, the number of entries in the **import address table (IAT)**, the PE header, code, data, and the file entropy, as was implemented in [6, 10].

2.2 Compression

Data compression is performed in a variety of fields today. Often, its goal is to reduce space, eliminate redundancies and package data more compactly and efficiently. Broadly, compression can be categorized as either lossless or lossy.

Lossless compression is used to pack data in ways that allow the data to be retrieved exactly as it was before compression. The cost of doing this lies in its computational aspect – wherein the device that performs the packing and unpacking will have to perform additional mathematical calculations which will take longer. A common method of performing lossless compression is through Huffman coding [119], which is a method of assigning shorter aliases to frequent and unique pieces of data in the file being compressed.

Lossy compression, on the other hand, is used to pack data in ways that result in data loss. While this is performed on most media files, such as music, videos, and images, it often results in a reduction in the content's quality.

The difference between compression techniques can be seen in images. PNG images are lossless, whereas JPEG images are lossy.

From the perspective of malware packing, lossless compression is used more often than lossy compression, as lossless compression ensures that none of the functions of the binary being packed are lost, when transforming them into unrecognizable states.

2.3 Encryption

Data encryption is used to protect file content from third parties that are unauthorized to view/access the content, to ensure data integrity using digital signatures, and to verify identity using certificates. Today it is common for digital documents, such as PDFs and Microsoft Office files, to be password protected.

However, from the perspective of malware packing, file encryption plays a more sinister role insofar as to allow malware to hide its payload and evade detection until the executable is installed and it obtains privileges to execute and make modifications to critical components of the filesystem. The type of encryption used also matters. There are two broad types of encryption: symmetric and asymmetric. If the encryption is symmetric, a single shared key is used to encrypt and decrypt the content. This can lead to various scenarios like the following:

- A command and control-based encrypted malware that installs itself and waits for a decrypt key from the control station to decrypt its payload and cause harm to the host system
- Malware that dynamically generates the key used to encrypt its malicious payload by performing a variety of checks and computations (such as looking for specific system artifacts) in order to determine whether it is being executed in a sandboxed analysis environment. If the malware is indeed being run in a sandbox, the decrypt key generation fails and the malware does not decrypt and execute its malicious payload.

Asymmetric encryption is often used in cases where the malware needs to be reused. Some kinds of ransomware utilize asymmetric and symmetric key encryption in tandem in order to ensure reusability as follows:

- The ransomware author generates a public key and embeds it in the malware.
- The ransomware has routines that generate a random symmetric key that is used to encrypt the files of the victim's system.
- The public key is then used to encrypt the symmetric key.
- After payment of the ransom, the victim is asked to submit the encrypted symmetric key to the ransomware author who then decrypts it using his/her private key and returns the decrypted symmetric key to the victim.
- The decrypted symmetric key is then used to decrypt the files of the victim.

In this manner, exposure of the actual decrypting key is prevented, thus allowing reusability of the mechanisms across several victims.

2.4 Packing

A packer is a program that transforms an executable binary file into another configuration using compression and/or encryption, in order to protect/hide the executable's original content. A packer's input and output are executable files. Packing (of executable files - *.exe in Win OS) is the most common technique for code protection, and it is used legitimately by organizations. However, code packing is also the most common obfuscation method used by malware authors to disrupt malware detection and analysis. Packers are used both for the reduction of the executable's file size and in order to avoid being subjected to static analysis [17], a process aimed at finding patterns within the code which help determine whether or not a file is malicious/faulty. Packing allows faster and safer distribution of executables over the network. Packing also applies encryption which makes it more difficult for hackers to conduct reverse engineering in an attempt to break the software license protection. Malware authors primarily use packing techniques for obfuscation, in order to hide their malicious code from malware scanners, meaning that a packed executable file is more resistant to anti-malware scanning and exposure; this is due to the compression and encryption that packing employs. In addition, packed executables increase the scanning time of the **antivirus (AV)** as runtime packers require additional work from the scanner, such as checking the file code and the unpacking process itself [3]. While, as mentioned above, packing is used legitimately for protection and classification, researchers have found that most malware programs are packed in order to evade detection [2, 6, 18–21].

Structurally, a packer is an executable program that takes an executable file or dynamic-link library, compresses and/or encrypts its contents, and then packs it into a new executable file. The packed file contains two main parts: (1) the first component is made up of a number of data blocks which form the compressed/encrypted version of the original executable file, (2) the second part is responsible for dynamic unpacking, during runtime, in order to retrieve the original executable file. When the packed file is running, the unpacking stub is executed in order to unpack the original executable code into memory and resolve all of the imports of the original executable, and then the unpacking stub transfers the control to the original file. In the case of PE file packers, the process in which the unpacking stub resolves imports is done by using the LoadLibrary API call which is aimed at notifying the Windows OS to load DLL into the address space of the process and return handles using the GetProcAddress API. The GetProcAddress call aims to obtain the virtual addresses of the symbols these libraries export. Usually, the execution of the original file is unchanged and starts from its **Original Entry Point (OEP)** with no runtime performance penalties [3].

When executing a regular executable file, the loader reads the PE header from the disk and allocates space in the RAM for each of the executable's sections, according to their settings in the header. The loader then duplicates those sections into the allocated spaces in the RAM. However, the PE header of the packed executable might be formatted so that it contains several sections for the unpacking process, and then the unpacked file produces additional sections (the original program). Alternatively, the PE header might contain many sections to include all of the program sections and the unpacking stub. The unpacking stub unpacks the original code for each section and copies it into the allocated memory block [13].

The appearance and traces of the malicious code inside an executable file are changed after the executable file has undergone the packing procedure. In order to illustrate changes that occur in response to the packing or unpacking process, we present the following example. Assume that X is a regular PE file that contains known malicious code, and X' is an encrypted version of X. When X' is executed, it is decrypted in memory. In that case, a signature-based detection solution will be able to detect the malicious code during runtime. However, if X is packed, that detection solution will try to statically match the signature of X with X′ (not during runtime). As the malicious code of X is encrypted in X′, no match will be found. Therefore, X evades detection and contaminates the targeted device if X′ is executed [2].

A packed PE file and the original file have the same functionality. Existing packers and their compression or encryption algorithms retain some of the properties present in the original code (e.g., byte alignment). However, these methods transform some or all of the original code bytes into a series of random-looking data bytes. Therefore, a packed PE file has a different byte structure and signature than the original.

To summarize the information on packers mentioned above, packer developers have different motives for writing their packers [12]. A packer is a defense mechanism against software hackers who try to access the software's files and alter them in many ways (e.g., using program analyzers, debugging, memory dumping, and data directory analysis). The packer provides an extra layer of protection without affecting the original functionality of an application. Generally, this protection is accomplished through a combination of encryption and compression that hinders reverse engineering. Encryption makes unpacking more difficult, and this is an advantage for malware authors, as it increases the number of variants that can be generated by easily changing the encryption key. Before executing the first instruction of an executable, decryption and decompression must take place at execution time.

Figure 4 presents the process of packing and unpacking the PE file. As can be seen, all of the original sections are compressed after packing, and an unpacking stub is added. Figure 4(a) presents the original executable before packing (all strings and code sections are visible as regular clear text). Figure 4(b) presents the packed version of the original executable (the code sections and strings are compressed and/or encrypted in the "packed original code" section); as can be seen, during packing, an "unpacking stub" is added at the head of the executable. Figure 4(c) presents the rebuilt, de-obfuscated, original executable, after unpacking.

Figure 4 also illustrates the four major steps of a PE file's packing-unpacking lifecycle. The packing process begins with the obfuscation of the content of the original PE file, followed by the addition of various anti-analysis routines that are meant to either make analysis more difficult and/or to prevent the exposure and execution of the obfuscated payload. When the packed executable is executed, the entry point of the packed executable is usually in the unpacking stub (which contains the aforementioned anti-analysis routines). If the various anti-analysis checks pass, the original PE file is unpacked and written to memory (the methods used for unpacking differ based on the packer used to pack the PE file). After this, import resolution takes place and the **Import Address Table (IAT)** of the original executable is rebuilt. Following this, control of

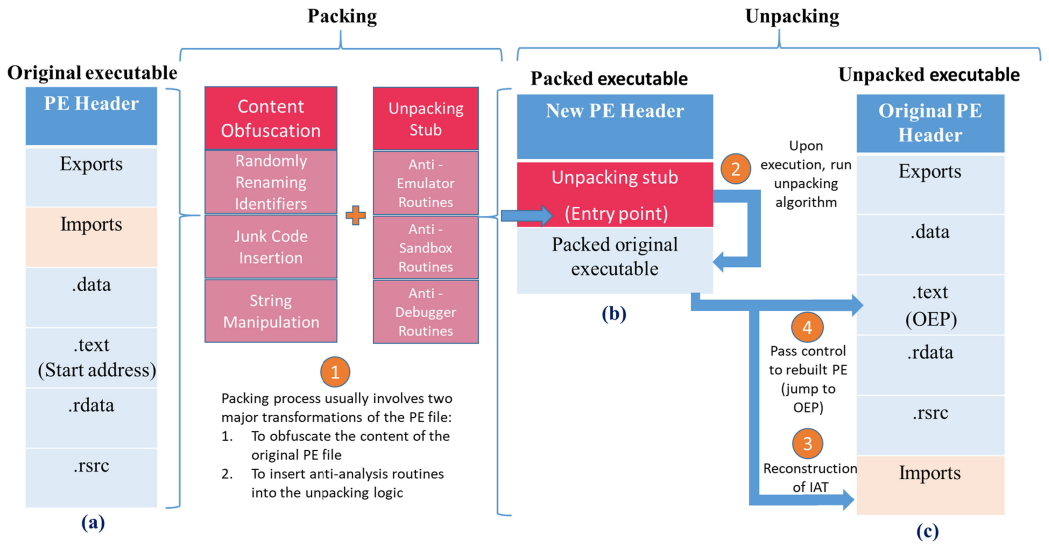


Fig. 4. Packing and unpacking process: (a) the original file, (b) packed version of the original file, and (c) the structure of the original file after unpacking.

execution is transferred to the **Original Entry Point (OEP)** of the original executable which then executes its payload.

It is important to note that a PE file can also be packed in a nested way using multiple different packers, which means that the aforementioned packing-unpacking lifecycle can be chained together multiple times (based on the number of times the PE file is iteratively packed).

2.4.1 Packing's Effect on PE File Structure. In this section, we present and discuss the changes that occur in the PE file structure due to packing techniques.

Packers usually first encode malware code by using some compression or encryption techniques and attach an unpacking stub to the packed malware. Packers change the distribution of the code's bytes by creating a "flatter" distribution which can lead to false positives in packer identification while using the $\chi 2$ uniformity test [22] or other similar statistical methods. In the case of compression, frequent values are substituted by references or short symbols. Compression and encryption severely hinder any similarity computation on executables, since the content of code sections is modified, both in terms of byte sequences and statistical properties [8, 14].

Jacob et al. [22] manually examined a number of popular tools (e.g., LZ77 for compression and RC4, DES, or AES for encryption) used by malware authors, in order to better understand the ways in which packers change the body of an executable file. The authors conclude that in cases in which a packer writer uses both compression and encryption, all similarity between the original executable and its packed version is canceled. Nevertheless, some information is preserved by both compression and weak encryption algorithms (usually these algorithms are faster and cost-effective in terms of computing power needed). According to their findings, existing packers and their compression or encryption algorithms retain some of the properties present in the original code. A pair of similar code distributions remains similar following packing if its byte alignment (arrangement) and permutations (transformation) are managed. As mentioned before, compression usually modifies the byte alignment. It was found that some symbols' sequences are incompressible by dictionary-based compression. Consequently, compressors preserve the similarity of the symbol distribution of two programs. In contrast, encryption does not change the

Table 1. Impact of the Different Packing Algorithms on the Binary Content

	Dictionary Compression	Range Encoding	Arithmetic Encryption
Principle	Compression is achieved by replacing repeated occurrences of data with references to a single copy of that data which exists earlier in the uncompressed data stream.	Compression is achieved by replacing a single integer range representation.	Encryption of blocks by reversible arithmetic operations.
Alignment	Byte alignment is preserved.	Byte alignment is not preserved by the shortened integer ranges.	Byte alignment is preserved.
Sequences	Order of incompressible blocks is preserved.	The sequences are shortened, yet the order remains the same.	Sequence is preserved with the encrypted values.
Distribution	Flattened – relative references that introduce infrequent values replace frequent blocks.	Distribution is destroyed yet can be reconstructed over the encoded ranges.	N-gram distribution is permuted, where n is the size of the encryption block.
Packer utilizing this mechanism	UPX	NsPack	PolyEnE

Table 2. Differences between Packed and Unpacked Files in Terms of File Properties

Property	Packed File	Unpacked File
Number of import functions	Smaller number	Higher number
IAT	Reconstructed	Standard
Code visibility	Low	High
Debugger flags	Opening packed file in some debuggers causes a "may be packed" warning to be issued	No such warning is displayed
Section names	Abnormal section names such as "UPX0" or "Dialog"	Standard section names
Section sizes	Abnormal sections sizes such as .text section of size zero	No abnormality seen
Entropy of byte sequences	High entropy	Low entropy

byte alignment. Yet, it changes the bytes' distribution (blocks) in the original program. This transformation (permutation) depends on the encryption key. Table 1, that is based on previous study of Jacob et al. [22], presents the impact of the different packing algorithms on the binary content.

Another prominent change is that packers erase the PE file's **import address table (IAT)**, which acts as a lookup table for dynamically linked API calls, in order to increase the complexity of API call analysis. First and foremost, when a packed malicious PE is executed, the unpacking stub decodes the malicious payload binary to memory pages and reconstructs its IAT before resuming the execution of the original code from the OEP [23].

There are seven main differences between packed and unpacked files [13]. In the case of packed files it can be seen in Table 2.

Researchers take advantage of the randomization phenomenon (number 7 above), applying statistical tests, such as the χ^2 test and using tools aimed at finding the longest sequence of identical bytes in order to further analyze the packed PE file [22].

2.5 Packer Categories

There are four main varieties of packers: compressors, crypters, protectors, and bundlers [117].

2.5.1 Compressors. These types of packers are only built for shrinking files and usually do not insert anti-analysis/unpacking routines into the files they pack. A few examples of compressors include UPack, UPX, and ASPack.

2.5.2 Crypters. These types of packers have encryption and obfuscation routines built into their logic and apply these anti-analysis techniques over the content of the files they pack, thus making the files difficult to analyze statically. Some examples of crypters include Yoda's Crypter and PolyCrypt PE.

2.5.3 Protectors. These types of packers combine the methods used by compressors and crypters. Armadillo and Themida are examples of protectors.

2.5.4 Bundlers. Bundlers are used when multiple files need to be compressed and executed as a package. Bundlers are used to combine the necessary data and executable files into a single executable file which, when run, unpacks and accesses files in the package without writing to disk (in memory execution). PEBundle and MoleBox are examples of bundlers.

2.6 Unpacking Techniques

In this section, we present unpacking techniques that unpack packed malware in order to analyze the behavior of the malware. We then briefly outline the limitations and shortcomings associated with the process of unpacking packed executables.

The unpacking process is the opposite of the packing process. The original file is restored into the volatile memory when the unpacking process is accomplished. The first phase of the unpacking mechanism is decompression. After reviving the original executable, the execution flow jumps from the end point of the decompression module to the OEP of the original executable, using the "jump" instruction. The unpacking process is performed when the OEP is reached [19].

A non-packed executable is loaded by the OS to the volatile memory. With packed programs, the unpacking stub is the mechanism responsible for extracting the original code from a packed file. The unpacking stub is loaded by the OS, which then proceeds to load the original program. The unpacking stub can be seen by a malware analyst, however it is necessary to understand the different parts of the stub in order to unpack the executable. When static analysis is performed on the packed program, the stub, not the original program, is analyzed (i.e., the packed executable must be unpacked before its execution).

There are four ways to unpack packed executables in order to analyze them: automated static unpacking, automated dynamic unpacking, manual dynamic unpacking and manual static unpacking. Automated static unpacking [13, 43] programs decompress and/or decrypt the file without running it; these programs are only useful for a single-layer packer. In contrast, automated dynamic unpacking runs the executable and enables the unpacking stub to unpack the original code. This technique relies on determining the boundary between the end of the unpacking stub and the beginning of the original executable, which might be difficult to accomplish when camouflage methods are applied. The unpacking process will fail if this boundary is not correctly determined. Automated dynamic unpacking is done in an isolated environment, such as a virtual machine or sandbox, in order to avoid host machine infection when executing the packed malware. There is a need to mention that currently there is a lack of reliable, publicly available, automated dynamic unpackers; on the other hand, automated static unpacking tools are widely used today. Manual dynamic unpacking is the process of restoring the original code, its OEP and IAT, using debugging tools such as ImpRec [109] which can restore an IAT that has been destroyed, and LordPE, [110] a tool aimed at editing PE headers. Manual unpacking can be divided into two main approaches. The first approach is based on discovering the packing algorithm and writing a program which

Table 3. Commonly Used Unpacking Techniques

Technique Name	Description
JIT compilation	This technique involves performing compilation during the execution of the program at run time, rather than before execution.
IAT Hooking	The Import Address Table (IAT) contains information such as DLLs, a list of function names and addresses of these function names from the DLLs that can be called by the binary during execution, etc. Hooking the IAT of a specific target program in the analysis environment allows monitoring of all libraries that are being called and the functions that are being executed.
Memory Dumping	Using memory dumping tools like Scylla [127] to take a snapshot of the memory of the sandbox environment to save the loaded state of the file being analyzed in memory
OEP hunting through debugging	Finding the Original Entry Point through manual debugging using techniques such as setting breakpoints on APIs like LoadLibrary, VirtualAlloc.

will attempt to reverse its execution order in order to unpack the original file. The second strategy is to execute the packed program as it is, so that the unpacking stub will unpack the program, and then dump the process out of memory, and manually set up the PE header.

Automated unpacking techniques are faster and easier to use than manual dynamic unpacking techniques and do not require manual human intervention, but automated techniques do not always work properly, since you must first identify the target packer, and there is not always a suitable automated unpacker available. Static unpacking is a technique used to reveal the hidden characteristics of malware. Most packers enable automated packing and unpacking as well as decrypting. In addition, there are also manual techniques to do so [13, 21]. Manual unpacking is expensive and time-consuming. Usually, static unpacking of executables is more difficult to perform than dynamic unpacking because of the need to employ reverse engineering [44]. Unfortunately, existing generic unpackers rely on a dynamic instrumentation of executables [15, 22, 29], and thus also suffer from performance limitations due to the code execution [22].

The major shortcoming of standard runtime packers (which are also known as self-extracting archives, i.e., software that unpacks itself in memory when the packed file is executed) is that the original code has to be executed at some point. This disadvantage can be exploited by manual unpacking (after debugging the file and determining when the original code is totally unpacked, the file can be dumped into the memory by setting headers and rebuilding the import address table). Another method involves applying reverse engineering on the unpacking stubs and using the information exposed as a result to create packer-specific unpackers that can statically unpack executables, i.e., without executing them. Many runtime packers try to block these unpacking techniques by using anti-debugging and code obfuscation procedures. It must be noted that some general packers, such as UPX, support unpacking out-of-the-box [45]. Over the years, researchers have tried to improve or customize methods to unpack packed executables. Coogan et al. [43] suggested an automatic unpacking mechanism that uses static analysis methods in order to identify the unpacking code. Then, the proposed mechanism uses this code section to produce a customized unpacker for that binary [2]. In contrast, Guo et al. [46] presented the **Just-In-Time (JIT)** unpacking method, and Yu et al. [47] proposed AGUnpacker, an unpacking technique which removes code obfuscation from executables. The latter is able to decide when the program decrypts itself into memory completely by tracing the JUMP instruction. Table 3 describes a few methods for dynamic code unpacking.

2.7 Anti-Unpacking Techniques

Packers rarely incorporate anti-unpacking techniques to obfuscate their unpacking code instructions, however there are several techniques that can be used to hamper debugging of an executable, especially when using dynamic analysis. Recently, the use of anti-unpacking techniques and

Table 4. Commonly Used Anti-Unpacking Techniques

Technique Name	Description
Anti-dumping	This technique changes the memory addresses of the running process in order to prevent further analysis of the dumped memory. The change is mainly employed in the following sections: PE header, imports, and entry point codes, which provide valuable information for analysts.
Anti-debugging	This technique is used by malware writers to identify when the code is being investigated by an external debugger (the most convenient tool for tracing code execution) and thwart this. This technique makes it difficult for signature-based solutions (such as AV) to easily use a debugger [48].
Anti-intercepting	Some unpacking mechanisms work by intercepting the execution of newly written pages, in order to guess when the unpacking process is finished and the control shifts to the host. This technique aims to prevent the packer from newly written page instructions [49].
Anti-emulating	This technique is used to attack the software environment, such as an emulator or a virtual machine, which is needed for safe code execution. Such attacks are dummy loops which delay the execution of the main malicious payload or increase error codes caused by invalid parameters which fail to decrypt the data [49].

obfuscation methods has been increasing, and they are being applied to packers more frequently [3]. Table 4 presents several commonly used anti-unpacking techniques.

Dealing with anti-debugging techniques through manual unpacking is extremely difficult and time-consuming. Classification of the packing algorithm used based on packer identification might provide analysts with valuable information about the packer's anti-debugging and obfuscation techniques [19, 50] and would be the first step in dealing with anti-debugging techniques (as it would significantly improve analysis times).

3 PACKING TECHNIQUES

In this section, we discuss the use of packing techniques and obfuscation methods, focusing on their pros and cons, in order to understand what motivates users to implement them on their executables. At the end of this section, we provide deeper insights into two packers: UPX and Themida. We chose to elaborate on these two, since they represent two distinctly different packing approaches.

3.1 A Taxonomy of Packing

Packing algorithms are heavily used in malware development because of their ability to help malware evade detection. Malware writers take advantage of the major drawback of static analysis-based detection tools which rely on a signature repository and easily evade detection by packing the malicious payload in layers of compression or encryption. Code packing techniques are evolving rapidly, providing new features and implementing methods that make the packed malware increasingly difficult to detect through static analysis techniques. Many existing packers can be easily modified, customized, and expanded upon using a variety of compression algorithms. Given the fact that many compression algorithms and packers are open-source, it becomes an added advantage to anyone seeking to create new variants of packers. According to [4], 35 % of malicious code is packed by a custom packer, and the number and complexity of new packers continues to increase.

When a packed binary begins running, the original code is written in memory pages and then gets executed. The procedure of writing to memory and then executing the written memory can be repeated many times, i.e., the dynamically produced code itself might continue generating new code and executing it. Each repetition of dynamically generated code is called a layer [26]. In other words, a layer is a series of memory addresses that are executed and result from code that was written by another layer.

Packers commonly used by malware developers can be classified into three main types: single-layer packing, repacking, and multi-layer packing algorithms. All three kinds are aimed at helping malware evade detection. The simplest case is single-layer packing, since it uses only one packer in order to pack a given binary, where the size, number of sections, and name of the executable are changed. Repacking is a more complicated technique in which the same packer is used at least twice in order to pack a binary; repacking uses compression techniques that are similar to those used by a single-layer packer. When a repacked file is executed, each layer must be sequentially unpacked. In this case, packers contain repacked unpacking layers which are executed sequentially in order to unpack each of the sections. The most complex packing type is multi-layer packing, which contains multiple unpacking layers, each of which is executed sequentially to unpack the routine that follows it. Once the original code has been reconstructed, the last transition transfers the control back to it. This category differs from the repacking technique, because the layers are created by different types of packers. The multi-layer packing approach utilizes a combination of several different packers to pack a file. This property of multi-layer packing facilitates the generation of a large number of packed binaries from the same input and consequently can result in the creation of many malware variants from the original malware. This algorithm changes the size, number of sections, and name of a single-layer packed executable [6].

In 2015, Xabier et al. [27] conducted a study based on 389 packers and found that 92.7% of them were multi-layer packing algorithms, while research performed by Bonfante [28] in the same year found that 67% of 28 packers were using multi-layer packing techniques. Multi-layer packing makes it difficult to determine the end of the unpacking, however research has shown that detecting the end of unpacking layers can be reduced to an NP-complete problem under certain assumptions [29]. There are two main factors that influence multi-layer packing's complexity. First, the "written-then-executed" feature is just an indication of dynamically generated code that is subsequently executed, however it is not the original code (i.e., a routine of a set of memory addresses that were written by some layer and then executed by the following layer). Second, the original code may be located in each layer (i.e., the original code is not necessarily found in the deepest layer). Sometimes the deepest layer may contain junk code just to mislead analysis systems that work on the assumption that the deepest layer contains the malicious payload. To implement the unpacking process, the traditional approaches involve having to go through each layer and unpacking it in order to determine the OEP. This process requires the allocation and usage of a large number of memory addresses and is computationally expensive, thereby increasing runtime [6, 26].

Another notable characteristic of advanced packers is that they utilize various anti-analysis techniques to interfere with unpacking tools. Because of this malware tries to detect when it is being analyzed by antivirus systems, so it can modify its behavior to avoid triggering any alarms. Such cases where packers have anti-debugging capabilities in their unpacking-stubs are, admittedly, rare. Anti-debugging capabilities of a malware indicate its ability to detect the presence of a debugger - malware can either read some values from the file or it can use some API calls to detect if it is being debugged. Both anti-analyzing and anti-debugging slows down and even prevents manual reverse engineering. To trace the progress of a packer's self-decoding, universal unpacking is typically based on dynamic analysis methods, such as debugging, dynamic binary instrumentation, and API hooking.

As mentioned, complex packers might involve several layers; the layers may include unpacking iterations in a chain structure, i.e., the first layer unpacks the next layer, and so on, until the original code is rebuilt. Some packers create several processes aimed at unpacking the original code. Those packers take the form of droppers (files whose purpose is to install other more severe malware on the system) and create a file that is subsequently executed, while others create a separate process and then inject the unpacked code into it. In some cases, packers even intervene

deeply by unpacking and repacking the code on-demand at different points of the execution process [27]. According to Cesare et al. [30], 88.26% of malware were recognized as variants of already analyzed malware samples. Given this growing phenomenon, this high percentage signifies the importance of detecting malware variants when detecting unknown malware samples. Another observation is associated with the finding that 34.24% of packed malware is similar to existing malware [25]. This suggests that many new samples of malware are repacked versions of existing malware. Over the years, researchers have proposed different ways to pack files. Gupta et al. [15] proposed a technique of packing by using hash key-based encryption. Using this technique, multiple applications can be packed into a single executable at the same time. However, Li et al. [31] proposed a technique for packing an executable file based on tokens, using anti-debugging and anti-dumping of the memory segment. Their results showed that this algorithm is time-consuming compared to traditional methods.

At first, packers only used compression mechanisms aimed at reducing the file size and accelerating network distribution. Later, packer developers added encryption to the basic compression algorithms in order to impede reverse engineering. Unpacking tools, on the other hand, are slow to adapt to the evolution of packers, resulting in the reduced effectiveness of newly proposed malware detection solutions [20]. Recently, more sophisticated techniques, such as virtualization and dynamic unpacking, have been integrated into packing algorithms. In virtualization, the original program is interpreted into virtual instructions that are then executed by an embedded virtual machine [22]. Unpacking virtualization obfuscators are a growing phenomenon which have been applied in HashiCorp, Themida, VMprotect, and Obsidium packers. Dynamic unpackers execute a packed program in a sandboxed analysis environment and monitor its execution. They then try to determine when the program has finished unpacking itself, so that they can locate the unprotected code in the memory [32].

In order to better understand the problem of packer identification and classification, we created a taxonomy (Figure 5) based on the packer's level of complexity, in terms of its packing structure (Level 1 represents the lowest level of complexity, while Level 8 represents the highest complexity level. In each level of the taxonomy, we defined the packer configuration.

Code packing is primarily used in the Windows OS, although in recent years it is being used in other platforms as well (e.g., gzexe for Linux, VMProtect for Mac OS X, multi-platform UPX, and HASP) [33]. Some of the reasons for this is that Windows OS provides a larger attack surface for potential attackers, and that it is also the OS that is more likely to be used in the target victim's workstation.

3.2 Packer Distribution

Most packers are available free of cost and are easy to use. UPX and Themida are two of the most common Windows OS file packers [3]. One of the reasons for Themida's popularity is its ability to perform code virtualization and thwart attempts at analyzing files that were packed using this packer during dynamic analysis. Ironically, this packer was developed with the aim of protecting Windows applications from hackers and IP theft. Other very popular packers, all of which are available online, include FSG, MEW, MPRESS, NSPACK, ACPROTECT, MOLEBOX, PECOMPACT, PELOCK, etc. [1]. All packers mentioned above are packers for PE files and are supported by the Windows OS (UPX also supports the Linux environment) [35, 36]. There are fewer packers supporting just Linux environments, such as Burneye and Midgetpack. According to [36], who collected packed malware samples in order to understand Linux malware, out of 380 packed binaries only three were not UPX variations. All UPX modifications were "cosmetic," such that the ELF does not appear to be packed with UPX, although they all made use of the same compression algorithm. Table 5 lists popular packers, their share among all other popular packers,

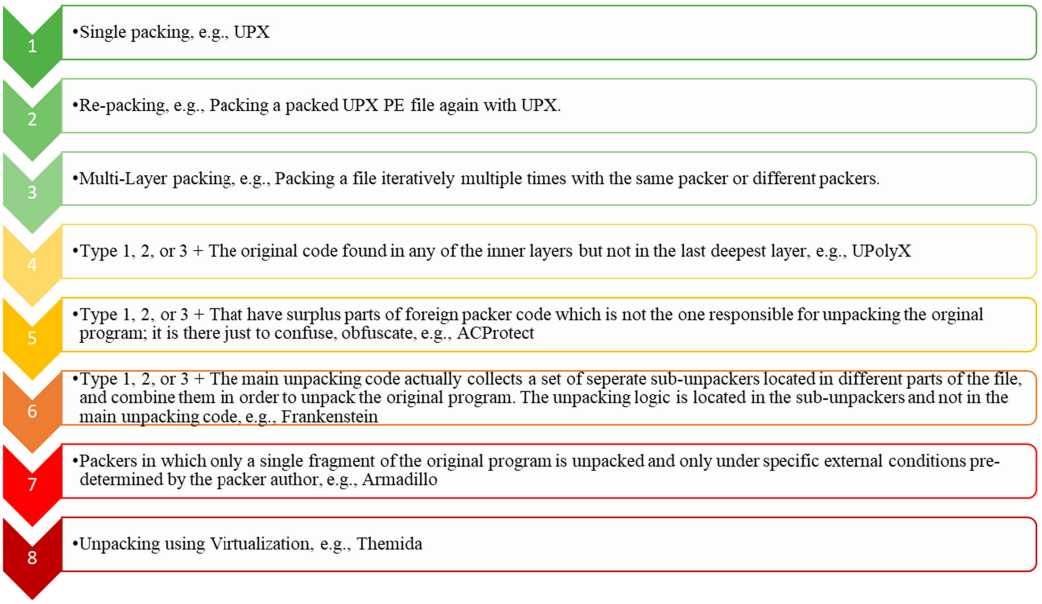


Fig. 5. Taxonomy of packers based on the complexity level of their packing structure; higher numbers are more complex.

and the number of files they packed out of the 2,000 files randomly sampled each year between 2010 and 2021; note that most of them only operate in Windows (with the exception of UPX which operates in Windows, Linux, and Mac).

3.3 Case Study of Two Packers

We now describe two packers that are used for packing executable files: UPX and Themida. These two packers were chosen since they are dramatically different both in terms of how they operate as well as the level of complexity of packing techniques used – with UPX being among the simplest and Themida being among the most complex packers.

3.3.1 UPX. Introduced in March 1998, **Ultimate Packer for eXecutables (UPX)** has become one of the most widespread packing tools used. UPX offers a compression ratio higher than other compression tools such as WinZip or GZIP. It offers a decompression speed of 10 MB/sec which is faster than other tools. Moreover, using UPX compression and decompression do not require high memory overhead;additionally, compression reduces the file's size by 50–70% when using UPX. Its most important advantage is that it is universal, i.e., it supports many file formats and platforms [38] (e.g., Windows, Linux, Mac). The fact that it is based on open-source code allows the user to create modifications within the algorithm in order to increase the complexity of its standard unpacking process. Usually, users create cosmetic changes to the packed file so that it does not seem to be packed with UPX. Among the most common changes are changing the magic number, changing UPX strings, the insertion of junk bytes, and even a combination of all of these methods.

UPX works by compressing the sections stored within the PE file section table (each row of the section table is, in effect, a section header. The section table is generally located one byte after the headers [11]). UPX reformats the binary; all existing sections (e.g., text, data, rsrc) are compressed, and three new sections are created, named UPX0, UPX1, and UPX2. Table 6 presents a PE file's new section names after packing with UPX, along with each of their respective descriptions. After

Table 5. Packer Distribution by Year

Packer	Total Packed	Packing Share	Malicious Packed PE Files Out of the 2,000 Randomly Sampled Files for each year											
			2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
Armadillo	534	12.99%	59	119	136	200	17	2	0	0	1	0	0	0
ASPack	416	10.12%	74	50	41	41	26	12	70	17	20	29	5	31
ASProtect	44	1.07%	11	7	7	10	3	1	2	2	1	0	0	0
BobSoft Mini Delphi	847	20.61%	51	43	33	46	23	20	125	64	97	68	85	192
EXECryptor	10	0.24%	1	4	2	3	0	0	0	0	0	0	0	0
eXPressor	1	0.02%	0	1	0	0	0	0	0	0	0	0	0	0
FSG	17	0.41%	6	3	2	6	0	0	0	0	0	0	0	0
KByS	1	0.02%	0	0	0	0	1	0	0	0	0	0	0	0
MEW	10	0.24%	2	0	2	6	0	0	0	0	0	0	0	0
MoleBox	16	0.38%	1	6	5	3	0	0	0	0	0	0	1	0
Morphine	2	0.04%	1	0	0	1	0	0	0	0	0	0	0	0
nPack	2	0.04%	0	0	0	2	0	0	0	0	0	0	0	0
NSPack	39	0.94%	7	4	4	17	2	0	2	2	0	1	0	0
Obsidium	2	0.04%	1	0	0	0	0	1	0	0	0	0	0	0
ORiEN	1	0.02%	1	0	0	0	0	0	0	0	0	0	0	0
Packman	2	0.04%	0	1	0	0	0	0	0	0	0	0	0	1
PECompact	210	5.11%	49	38	28	33	10	6	9	15	13	3	0	6
PENinja	1	0.02%	0	1	0	0	0	0	0	0	0	0	0	0
Petite	24	0.58%	4	1	4	3	0	0	1	0	0	0	0	11
RLPack	10	0.24%	4	1	1	3	0	0	0	0	0	1	0	0
Telock	12	0.29%	0	3	2	2	1	0	1	0	0	2	0	1
Themida	22	0.53%	3	6	2	10	0	0	1	0	0	0	0	0
UPack	36	0.87%	4	5	3	15	0	1	0	1	0	1	6	0
UPX	1,850	45.02%	235	210	166	200	251	190	116	180	87	76	32	107

Table 6. PE File's New Section Names After Packing with UPX

Section Name	Description
UPX0	An empty section where the code is uncompressed.
UPX1	The compressed form of the original binary and its uncompressing code.
UPX2	Contains all imports needed to execute the binary.

compressing the sections, a new code section is added at the end of the file, which decompresses all of the packed sections during execution. The stub in UPX1 executes and rebuilds the original code by reconstructing the import table (using UPX2) and uncompressing the code in UPX1 into the empty space of UPX0. It is worth mentioning that the compression algorithms of the UPX packer (named UCL) produce output with high entropy and consequently, an n-gram distribution closer to uniformity than its input [39].

There are six steps during the execution of a UPX packed executable file. The execution begins from new OEP (from the newly added code section at the end of the file). First, using the PUSHAD instruction (push all of the general-purpose registers in the stack) saves the current register status. Then, all of the packed sections are unpacked into the volatile memory, resolving the import table of the original executable file, and by using the POPAD instruction (pop all of the

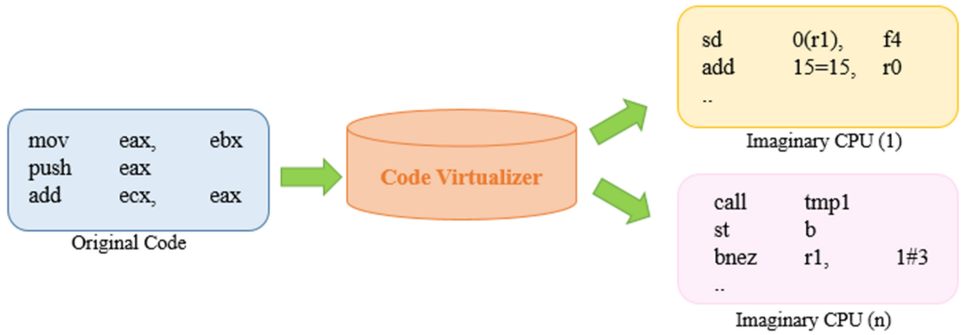


Fig. 6. Original instruction set transformation using a code virtualizer to multiple imaginary CPUs with varying instruction sets.

general-purpose registers from the stack), the original register status is restored. Finally, there is a JUMP instruction to the OEP to start the actual execution [40, 41].

3.3.2 Themida. Presented in 2010 by Oreans Technologies, Themida is a widely used, sophisticated, commercial packer. Themida uses the SecureEngine protection technology. SecureEngine code runs on the same level as the OS (kernel), which allows the execution of any kind of protection technique without being restricted by the operating system. On the other hand, current cracker tools cannot detect, study or attack protection routines that are designed to run on the same kernel [42]. First, it decompiles some code blocks of the executable, and then each code block is protected. Protection may be applied by encryption or CodeReplace technology which randomly takes some parts of code and replaces them with garbage code. Then, overall protection is added, and an output, a protected executable, is produced. Themida's unique advantage is that the unpacking stub code is further protected by virtualization obfuscation.

Themida's code virtualizer converts the original code into virtual opcodes that can only be understood by an internal virtual machine. These virtual opcodes and the virtual machine itself differ for every protected application. Code virtualization consists of the conversion of binary code (instruction set) from a specific machine into another binary code that can be deciphered by another machine. Figure 6 presents a code virtualizer that generates various types of virtual machines with a different instruction set for each, i.e., a particular block of instructions can be transformed into a different set of instructions for each machine; this prevents an attacker from identifying any generated virtual opcode after the transformation, i.e., the attacker will not be able to find the original instructions by decompiling a block of code. Instead, the attacker will find a new instruction set. This forces the attacker to analyze and identify the inner workings of each opcode and virtual machine for each of the protected applications. To summarize, virtualization obfuscates the execution of virtual opcodes and the study of each unique virtual machine in order to prevent an attacker from studying how the virtual opcodes are executed. It is also worth noting that the code virtualizer changes the header of the protected application slightly, meaning that a compressor can be used on top of the code virtualizer [42].

Themida's virtual machine panel allows the integration of virtual machine technology into the executable file. In this case, one unpacking cycle instruction is replaced by different bytecode, and the attached virtualization engine will simulate this bytecode at runtime. Themida, for example, applies virtualization obfuscation to its unpacking routine which will result in instruction size explosion when tracing unpacking progress. Under the unpacking virtualization routine, tracing the packed layers becomes extremely complicated and time-consuming. While executing the

Table 7. Packers Used by Hacking Groups/Infamous Malware

Packer Used	Related Hacking Groups / Malware
Custom Packer	Anchor, APT3, Dyre, Elderwood, Emotet, FatDuke, FinFisher, GreyEnergy, H1N1, IcedID, jRAT, Cobalt Strike, Soft Cell, TrickBot, Uroburos
UPX	APT29, APT39, China Chopper, Dark Caracal, Dark Comet, HotCroissant, OSX_OCEANLOTUS.D, Patchwork, Rocke, SeaDuke, TA505, Trojan.Karagany, yty, Zobrocy, ZeroT
Themida	APT38, Lazarus Group
Enigma	APT38
VMProtect	APT38, Metamorfo
Obsidium	APT38
RPolyCryptor	Astaroth
MPRESS	Dark Comet, Daserf

encrypted code, Themida will first decrypt it inside the macro, then execute it, and finally encrypt it again [26].

3.4 Packer Usage in the Threat Actor Space

Table 7 presents the packers used by many hacking groups for packing their malware [105]. As can be seen, there are several that use custom packers for the purpose of packing their malware, which highlights an additional challenge in the domain of packing detection and classification – as a custom packer can be entirely unique and has a very high chance of evading existing pattern recognition techniques.

Note that while Table 5 indicates the use of packers by well-known hacking groups, it does not present the actual market share of the packers mentioned. It is likely safe to assume, however, that if such groups are using the packers listed in Table 5, other groups, and possibly even independent malware authors (who are not listed in the table), are also using these packers, both legitimately and maliciously.

In the next section, we discuss unpacking techniques.

4 DETECTING PACKED FILES

This section describes the main techniques that have been used by the packer identification and classification studies that have been surveyed. In general, we provide an overview of academic research on methods and tools aimed at the analysis of packed files, detection and identification of packed files, and a comparison of the methods and tools based on various criteria. We have divided the content into two major sub-sections – methods that deal with detecting/classifying packed files without using machine learning and using machine learning, respectively. First, in Figure 7, we present a taxonomy which divides the academic research performed into two main categories: research that is based on machine learning techniques, and studies that are not. Each major category has been divided into three subcategories: static analysis, dynamic analysis, and hybrid analysis. In each branch, we present works that perform packer detection and (in blue) and packer classification (in orange). Also, tools have been marked with a "*". A description of the research follows.

4.1 Non-Machine Learning based Packer Identification/Classification Techniques

In this sub section, we describe the main approaches for malware detection, particularly focusing on packed malicious executables. It is important to determine whether a binary has been packed or not as a first step, particularly when analyzing a large number of unknown binaries. Many studies (e.g., [1, 6, 45, 51–55]) perform unpacking methods when analyzing malware. However, Osaghae et al. [44] emphasized the motivation for classifying packed programs as malware

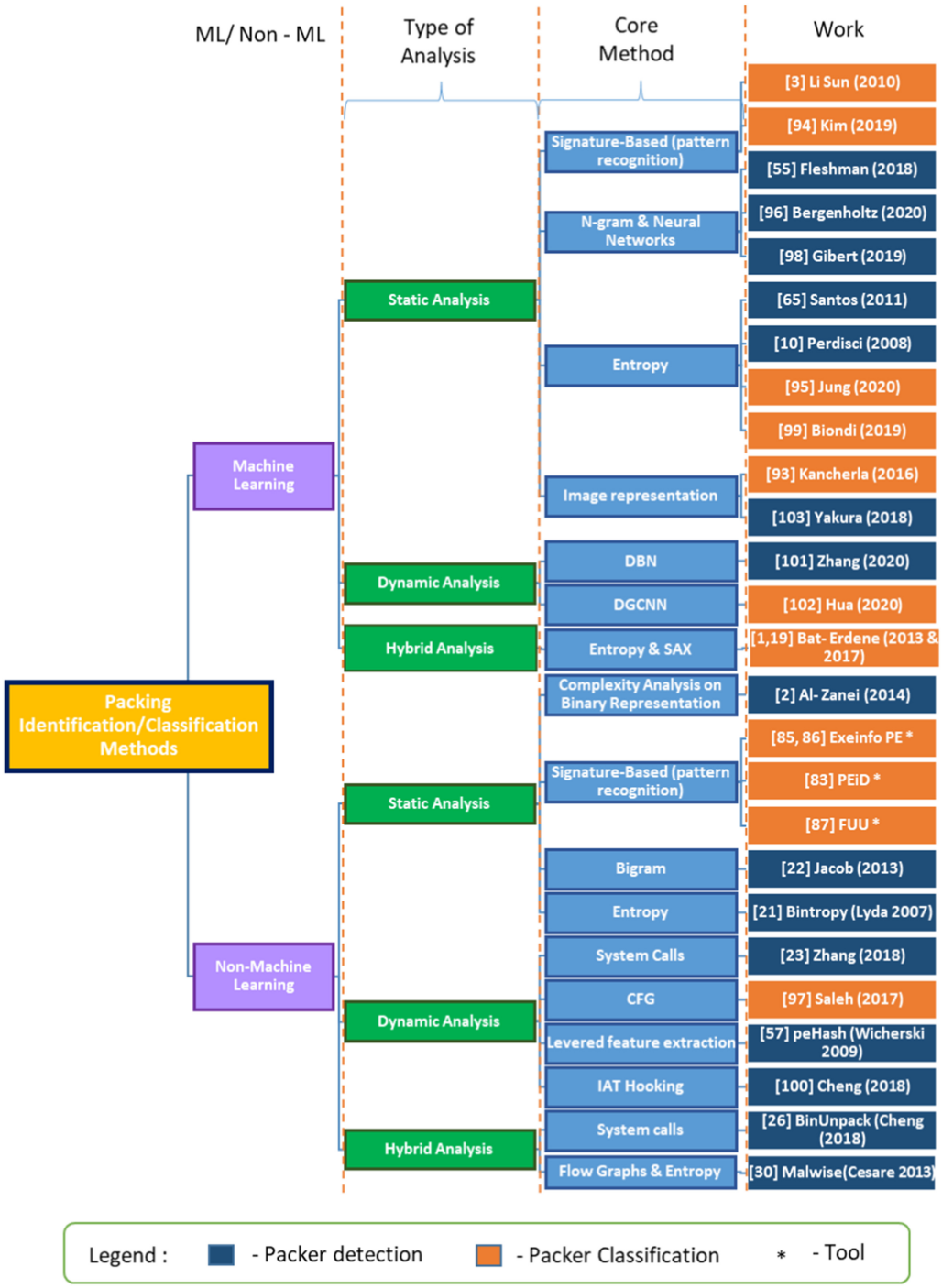


Fig. 7. Taxonomy of Packing identification and classification methods.

programs, instead of focusing on the unpacking process, prior to malware analysis. Four reasons are given to support their approach: (1) more than 80% of existing malware samples are packed, (2) a packed illegitimate program mistakenly classified as malware should not be considered a false alarm condition, because the reason why it was packed was not exposed to the user, (3) legal issues (for example, if a forensic investigation is conducted on a user's computer system and the packed programs found were confidential and stolen from elsewhere, it could lead to a legal problem for the user since he/she might be held responsible for being in the possession of such programs), and (4) complexities in the unpacking process which require reverse engineering expertise. The process of unpacking might also be time-consuming and require more computational resources.

There are several analysis methods that can help achieve the goal of determining whether a binary has been packed or not as an initial step. Advanced methods for the identification and classification of known and unknown packed files are based on three main approaches: static analysis, dynamic analysis, and hybrid analysis (i.e., a combination of static and dynamic analysis).

4.1.1 Static Analysis. Static analysis involves examining and scrutinizing files or applications solely by analyzing the file's content without actually executing it. Static analysis can be performed both manually and automatically (using tools). Basic static analysis can be used to gather a variety of high level information and meta-data such as the code path, the file size, and the file format, a cryptographic hash, imported shared libraries, the compiler used to generate the analyzed file, or even just a list of human-readable strings that are contained in the file or low level information gathered by disassembling or decompiling the file [2, 13, 45]. Some of the information that can be gathered is not directly related to the objective (e.g., information about the shared library, compiler, and file format), yet it can assist in its own disassembly and decompilation.

The main advantage of static analysis is its simplicity in terms of implementation, monitoring, and measuring, and the fact that it is not associated with long runtime or high computational overhead. In addition, since static methods do not involve executing a potentially malicious file, there is less risk of damaging the system that the analysis is performed on. Moreover, because the file is not executed, the analysis is virtually undetectable, meaning the examined file cannot detect that it is being analyzed and thus cannot change its behavior deliberately or interfere with the inspection mechanism. Advanced static analysis consists of reverse engineering the malware's internals by disassembling the executable and analyzing the program instructions executed by the CPU in order to discover and reconstruct the program's behavior. The static approach is also used to explore malware similarity (e.g., n-gram distribution [56]), but it requires that the suspected executable file be unpacked and disassembled. In the context of packer identification and classification, packer signatures or its entropy may be used to determine whether the file being analyzed might be runtime-packed [45].

The fact that packing methods transform the original bytes into a series of random-looking data bytes challenges researchers who use static analysis methods. In order to analyze large malware collections, researchers must quickly and efficiently classify the samples and unpack or decrypt them before analysis can begin [21]. The difficulty to recognize these transformed bytes depends on the transformation scheme's strength and the original bytes' statistical nature. Stronger transformation schemes produce more randomness and less predictable sequences. This principle underlies some of the static analysis methods, such as entropy, that we elaborate on later in this section.

Next, we present four subcategories of static analysis for the detection and classification of packed files and programs. Since the most common security solution among **personal computer (PC)** users is an antivirus engine, this is presented first. It is important to note that each of the

following methods relies upon different approaches, however all of them are aimed at extracting informative data from the file.

4.1.1.1 Signature Analysis. Signature-based detection involves a search for special patterns of a known sample in executable code. It is a common method employed by antivirus engines to detect malware by matching the malware signature to a signature of an existing threat saved in the antivirus known threat signature database. A signature is a fingerprint, i.e., a byte string or pattern that should match a certain instance or family of malware while not matching any legitimate software [45]. A fingerprint can also be defined as a distinctive set of bytes which occurs at the entry point or in sections of a PE file. Usually, scanning the whole signature repository is time-consuming. The signature repository is maintained by the vendors by first capturing the sample and then analyzing it. Note that the database needs to be updated constantly in order to decrease the window of vulnerability between a new piece of malware appearing in the wild and antivirus software being able to detect it.

The fact that signature-based solutions for malware detection are able to detect only known malware is the primary disadvantage; moreover, sophisticated malware can evade them [19]. During packing, the byte sequence of the original file is changed, paving the path for easy evasion of signature-based detection solutions. In addition, different packing techniques which pack the malicious code in layers of compression or encryption also make it difficult to detect the malicious content.

As mentioned before, since most packers are available online and are open-source, a large number of packing algorithms are created every year and their complexity is increasing. This variety of packing algorithms makes the packer detection process more complicated and time-consuming [1]. The main goal of packer classification is to quickly detect and identify the packer, allowing antivirus researchers to unpack the file and retrieve the original payload for further malware detection and analysis. However, there is no complete database of signatures to detect all malware. In addition, the signature repository must be updated continuously in order to keep pace with the rapid creation of malware.

To summarize this subcategory, signature-based detectors for packed executables are available on the Internet. The most common approach for packer classification is based on matching the packer's byte signature (a distinctive set of bytes which occurs at the entry point or in sections in a PE file). This approach is effective at detecting known packers, however it is not suitable for detecting unknown ones. Although signature-based detectors are efficient and result in relatively fewer false positives, in the case of packed executables, static signature analysis alone is ineffective. In this case, static signature analysis suffers from a high number of false negatives, since packer writers use advanced hiding techniques that enable them to avoid signature-based detection [10].

4.1.1.2 Analysis Using Hash. Analysis methods which use hash functions, such as the two methods below, might be effective for packer identification, although such methods have not yet been used for this purpose.

A system that attempts to detect similarity between malware samples without running them [22], peHash [57] uses structural information extracted from malware samples, such as the number, size, permission setting, and Kolmogorov complexity of the sections of PE executables. While this approach makes peHash efficient, its reliance on ephemeral features is not robust, and it can be easily confused. This method has been implemented by researchers attempting to find similarity patterns between malware samples and particularly to detect similarity between packed PE files [22]. MinHash [58] is a technique that allows a rapid assessment of the similarity between two samples. This method generates an N-size signature for a given file (regardless of its size and type)

based on N simple hash functions (fuzzy hashing). The similarity of two objects can be calculated using the Hamming distance between their signatures. The signature created by the MinHash technique can be used as an input feature for lazy machine learning algorithms which are based on a distance function (e.g., K-Nearest Neighbors). The MinHash method can save a lot of time and space. The MinHash method has previously been used for malware detection [59–62] and malware clustering [63], however MinHash may not be effective for the detection of packed PE malware files, since the malicious content is compressed or encrypted; thus, the MinHash signature represents the packed content and not the actual content. However, MinHash might be effective for detecting similarities between files that were packed with the same packing algorithm.

4.1.1.3 Entropy Analysis. In information theory, entropy is the measure of the amount of information that is missing before reception and is sometimes referred to as Shannon entropy [64]. Shannon entropy was originally devised by Claude Shannon in 1948 to study the amount of information in a transmitted message. Shannon's formula devised to measure the information entropy is as follows:

$$H(X) = -\sum_{i} P(i) * \log_b (P_i)$$

The definition of the information entropy is expressed in terms of a discrete set of probabilities p(i). P(i) is the probability of the ith unit of information in the series of n variables of event x, where H(x) is the value of the measured entropy value; two, 10, or Euler's number is usually used as the base number of the logarithm [6]. In the context of packed executables, P(i) might be derived from a string byte.

Packing algorithms can be identified using entropy analysis (information density). As mentioned, entropy is a term from information theory describing the amount of information that each symbol within a series of symbols carries [45]. Entropy analysis measures the uncertainty in an executable's data, without *a priori* knowledge of the executable. The PE file format can be represented as a byte string sequence. Both encryption and compression transform a one-byte sequence into another, causing an increase in the entropy of their input, i.e., high entropy may be indicative of the presence of packed malware. It is also worth mentioning that packers that use virtualization, (e.g., which generate a virtual machine to execute the malicious behavior) do not always increase the file's entropy [65]. The main disadvantage of using entropy is the problem of false positives.

There are several approaches that use static analysis in order to analyze packed files, ranging from signature matching, statistic calculation, and feature extraction in conjunction with machine learning. However, based on the prior research [13], basic static analysis is not effective for detecting packed programs; packed malware must be unpacked before it can be analyzed statically, which makes analysis more complicated and challenging. Due to the variety of packed malware, it is inefficient to use static analysis. If we ignore the need to detect and classify the packing algorithms used for packing a file and instead directly try to classify a packed executable file (without unpacking it), it will be difficult to detect whether an executable is malicious due to the encryption or compression of the executable, as this prevents detection solutions from accessing the original file content, especially in the case of static analysis [23]. Al-Zanei et al. [2] proposed a method aimed at quickly distinguishing between packed and non-packed executables in 2014. The method relies on the assumption that a compressed or encrypted packed executable file usually has high complexity and aims to measure information quantity in a more exact way than the entropy. The complexity level C(X) of a finite string X is defined as the length of the shortest string of X, i.e., the length of the shortest computer program that represents X and then pauses. The complexity function is defined by the equation C(X) = min{X} which uses binary static analysis to measure the file's complexity according to data extracted from the PE file. Complexity assessment enables

the authors to distinguish between packed and non-packed files, so only executables detected as packed are sent to a general unpacker for hidden code extraction, while non-packed executables were sent directly to an antivirus scanner. This study used a collection consisting of 250 benign unpacked programs that were randomly gathered from Windows XP OS system files; these files were then packed using the UPX packer. The result shows high accuracy with a TPR of 96% and a low FPR of 4%.

In 2013, Jacob et al. [22] presented a robust technique aimed at quickly determining whether malware is similar to another sample that was previously analyzed, using code signal properties statically extracted directly from packed code. The distance between code signals is measured in order to compute the similarity between the two programs. A code signal is a bigram distribution over the raw bytes of the code section. The authors used a code signal repository of previously analyzed malware samples. This database is updated, since whenever a new sample arrives, its code signal is extracted. The method is based on the fact that packed versions involve compression or weak encryption schemes that do not change among versions. Moreover, the proposed method can distinguish between different levels of protection, such as unpacked, compressed, encrypted, and multi-layer encrypted code. The dataset included 795,000 samples that were divided into 91,522 different groups with similar runtime activity. In order to evaluate the precision metric, they used a trade-off between reducing unnecessary analysis runs, the **True Hit (TH)**, and the risk of discarding potentially interesting samples, the **False Hit (FH)**. Filtering was measured by three thresholds. The highest precision was achieved by the first threshold with a TH of 90%, i.e., more than 90% of similar samples were correctly discarded. The results showed that the proposed similarity measure was highly effective in detecting malware variants, even if they were repacked, and therefore reduces the number of samples that need to be analyzed by a factor of three to five. Lyda et al. [21] developed a method named Bintropy in 2007. Bintropy is a prototype tool for binary file entropy analysis aimed at helping analysts identify encrypted or packed malware quickly. Bintropy is applicable on any file type, particularly the PE format. Bintropy divides the PE file into 256 byte blocks. Then it computes the entropy of each block, as well as the average and maximum entropy. Using a large database collection of packed files, they employed simple statistical inference to compute threshold values of the average and maximum block entropy. In their evaluation, if a PE file has entropy values above the thresholds, it will be labeled as packed [10, 21]. The main drawback of this method is that it uses simple statistical inference for computing confidence intervals on the entropy values, instead of applying statistical learning. However, work by Predisci et al. in 2008 [10] overcame this limitation.

4.1.1.4 Tools. This sub section briefly describes the few publicly available trusted tools that employ static analysis techniques in order to identify the packer or detect type of packer used to pack PE files; some of these tools are widely used. One common tool used to detect packed files is the PEiD program. This program can detect the type of packer employed (Figure 8 illustrates PEiD giving a scan result of a PE file that was packed using UPX) using static analysis methods. Currently, it is capable of detecting over 600 different signatures in PE files. This tool has several advantages compared to other signature-based identifiers. First and foremost, its detection rates are high. In addition, it has a plugin interface that supports plugins which are used to find the original OEP and open and the detected encryption algorithms. It has three different scanning modes allowing even better detection. Moreover, it is free and user-friendly [83].

Exeinfo PE [84] is another tool for the detection of packed PE files. It also provides valuable information which is extracted from the PE header (e.g., it shows the binary entry point, offset, compiler information, etc.). Exeinfo PE is a program that allows users to verify executable files and assess their properties. It is also used as a packer and compressor detector and provides packing

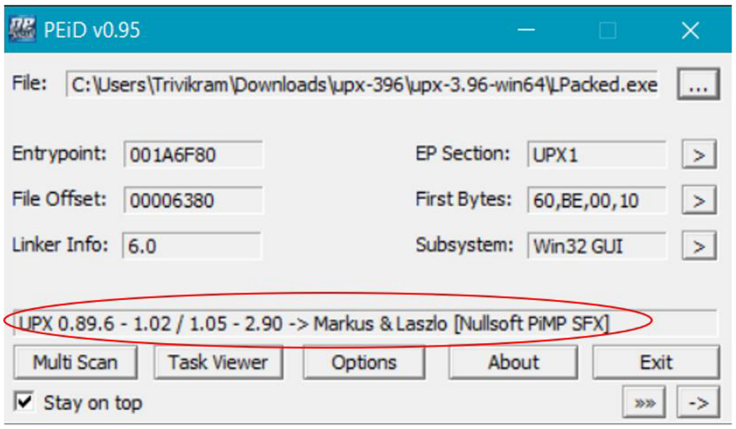


Fig. 8. PEiD program, an example of a UPX packer [13].

information. This tool can detect packers for various operating systems such as Windows, Symbian, Android, Linux, and Mac OS [85, 86].

FUU (Faster Universal Unpacker) [87] is a GUI Windows tool with a set of plugins that can identify different packers and apply custom unpacking routines specifically coded for different packers.

4.1.2 Dynamic Analysis. Dynamic analysis involves executing a suspicious program in order to reveal its behavior during runtime. Elements that can be gathered using dynamic analysis include the volatile memory dump of the suspicious process, all of the processes spawned by the suspicious process, threads, services, DDLs, and hard drive and network activity. In order to avoid a potential attack by hostile code while executing the program, the malware is typically executed inside an isolated environment, e.g., a sandbox. A common approach is to use a **virtual machine (VM)** as a sandbox, because VMs are easy to maintain and restore, and they provide a flexible architecture. However, the malware can detect that it is being run in an emulated environment and consequently delay or disable its malicious activity, in order to avoid detection.

Dynamic analysis can be performed at various degrees of abstraction. The simplest case is to record the system states during different stages of execution while the researcher controls the system's inputs and outputs. Other cases involve tracing the program's behavior in order to better understand the interaction of the packed file with the operating system during transitions from user mode to kernel mode code. For example, [23] used system call captures in order to trace the program's behaviors. Several commonly used dynamic analysis tools are: Cuckoo Sandbox [111], CWSandbox [112], and Falcon Sandbox [113]. In addition, there are online sandbox services such as Hybrid-Analysis [114] for PE files (*.exe) and Linux EML files, and online Cuckoo Sandbox for Linux EML files. The main drawback of a sandbox is its long execution time.

Many times, dynamic analysis can be time consuming and requires expensive resources, is high in computational complexity, while covering only one/few possible execution paths through a program (out of all possible execution paths). Dynamic analysis challenges include obfuscation from crackers who write private packers that make the packed file very hard to unpack [23]. Although dynamic analysis can reveal the behavior of a running packed executable, this behavior includes not only the original code's behavior but also the behavior of the packer, which may obfuscate the original behavior.

Dynamic analysis is an approach which can detect packed malware variants without manually unpacking the malware (as opposed to static approaches). Dynamic analysis has several

advantages. First, it can provide a researcher with an overview of what a certain program does relatively quickly. It also allows researchers tracing dynamic data flow to identify malicious activity in a generic way [66]. In addition, it is largely robust to obfuscation techniques that hinder static analysis methods.

The research performed by Zhang et al. [23] in 2018 aimed to detect packed malware variants using system call classification. The authors proposed a dynamic analysis method, which first extracts a series of system calls that are sensitive to malicious behavior to reduce obfuscation. In order to obtain the system calls, they performed dynamic analysis on the executable files using Cuckoo Sandbox. Following this, they used principal component analysis to extract features of these sensitive system calls and used deep neural networks to efficiently and effectively train and detect malicious instances. The training dataset included 3,167 unpacked malware executables and 2,894 unpacked benign executables. The test set included 2,083 packed malware variants and 1,986 packed benign files. Using their method, the authors were able to achieve 95.6% detection accuracy.

Another recent method was proposed by Cheng et al. [100] which proposed the use of a bare metal framework for the purpose of malware unpacking. The reason for a bare metal framework is that many malwares have evasion tactics built into their structure that allows them to search for system artifacts and indicators that would highlight that the environment that it is being executed in is an analysis environment (and if that was the case, to halt deployment of the malicious payload to evade detection). The use of a bare metal framework prevented such evasion tactics. In order to analyze the execution of the malware on the bare metal system, an **Import Address Table (IAT)** Hooking method was proposed. This allowed for the monitoring of the system calls that were being performed by the malware without leaving any artifacts that would compromise the presence of the analysis system.

4.1.3 Hybrid Analysis. Both static and dynamic analysis methods are applied to detect packed malware. Static techniques usually focus on a file's structure or its code byte string arrangement analysis, in order to find the signature of an already known suspicious function. In contrast, dynamic techniques focus on capturing the behavior of the packed code, for example, analyzing whether the code tries to reconstruct the IAT, which implies whether the file is packed and whether an attack may occur.

In order to overcome the drawbacks of each method, in some cases, it is preferable to combine both of those techniques, especially while dealing with packer classification [2]. For example, [67] combined static and dynamic analysis for the detection of malicious PDF files. Other hybrid detection models for PDF and Office files that conducted both static and dynamic analysis were implemented by [68–70]. PolyUnpack, a combination of static and dynamic analysis for automating the extraction of hidden malware code was proposed by [71].

In 2013, Cesare et al. [30] proposed Malwise, a prototype system for malware classification which uses automated unpacking. This approach applies both dynamic and static analysis methods. Initially, entropy analysis statically determines whether the binary has undergone a code packing transformation. If the binary is classified as packed, dynamic analysis is employed for application-level emulation to perform automated unpacking. Then, static analysis is used to identify features and build signatures for control flow graphs. The signature is represented as a string of characters. In order to create a signature, the algorithm orders the nodes in the control flow graph using depth-first order. The control flow graphs are structured to produce signatures that can be compared and used to approximate matching using edit distances. The signature is also correlated with a weight which symbolizes the importance of the signature when used to determine program similarity. The weight is relative to the sum of all weighted signatures in the program and is defined as the length of the signature normalized by the sum of the lengths of all of the signatures that appear in

the binary. Pairs of two graphs are not isomorphic if the signatures of the two graphs are different. This comparison is based on the assumption that a pair of similar malware variants shares similar high level structured flow of control. If the source code of the variant is a modified version of the original malware, this assumption would be proven. A measure of similarity is established by a comparison between the set of control flow graphs and those in a malware database. To classify a binary, the analysis makes use of this database which contains sets of flow graph string signatures of already known malware. In order to evaluate Malwise, the authors tested the prototype against 14 publicly available packing tools. The results show that UPX, rplack, mew, fsg, and npack were 100% successfully unpacked, in contrast to pepsin which was not successfully unpacked. To evaluate exact matching in Malwise on a larger scale, 15,409 malware samples were used. The results showed that 94.4% of the malware samples were found to have a similarity of more than 95% to previously classified malware in the set. A total of 863 representative malware signatures were stored in the database, and none of them were more than 95% similar to other signatures in the database. The authors also found that 88.26% of malware was detected as variants of previously classified malware. Moreover, it was reported that 34.24% of malware was 100% similar to existing malware, once unpacked. It is worth mentioning that the authors did not mention the false positive rate.

In 2018, Cheng et al. [26] designed BinUnpack, an efficient unpacking approach aimed at dealing with multi-layer packing algorithms. BinUnpack is based on the fact the IAT of malware is usually erased by packers but will be reconstructed before the original code resumes execution. When an API is invoked through rebuilding the IAT during the execution of packed malware, it indicates that the original payload has been restored. According to this approach, the unpacking routine of the IAT is extracted via static analysis, while BinUnpack monitors the dynamic execution of packed malware. Also, BinUnpack applies advanced bioinformatics inspired system call sequence alignment to bypass fake API calls. BinUnpack does not need to heavily monitor memory address accesses, and instead it requires just a small semantic gap in the runtime. It can also defeat evasion tricks via a novel kernel level DLL hijacking technique. The evaluation database included 238K packed malware and multiple Windows utilities.

In 2018, Cozzi et al. [36] conducted an experiment that relies on a set of heuristics to flag samples that might be packed. The heuristics are based on the file section's entropy and the static analysis results, such as the number of imported symbols, the percentage of the code sections correctly disassembled, and the total number of functions recognized. Since UPX customization is very prevalent, they added a set of custom analysis routines to their pipeline to identify possible UPX variants and generic multi-architecture unpackers. The UPX modification is aimed at changing its standard unpacking process. Among the most common techniques are changing the magic number, so that the file does not appear to be packed with UPX, changing UPX strings, insertion of junk bytes, and even a combination of all of these methods. However, these examples use the same packing mechanism and compression algorithms. In their experiment, the authors created and implemented an analysis infrastructure for Linux-based malware using three techniques: file and metadata analysis, static analysis, and dynamic analysis. In the static analysis phase, they combined information extracted from the ELF headers and the binary code analysis in order to identify packed applications. Binaries that could be statically unpacked were then processed and the results were fed back so they could be statically analyzed again. Samples that could not be unpacked statically were analyzed through dynamic analysis. In the dynamic analysis phase, the system recorded network traffic into PCAP files. Eventually, to dynamically unpack unknown UPX variants, the authors developed a tool based on the Unicorn [115] emulator. The system emulates instructions on multiple architectures and behaves like a tiny kernel that exports the limited set of system calls used by UPX during unpacking. This allowed the authors to automatically unpack 100% of the samples that were packed with UPX or its variants (377 files), however they were

Table 8. Comparison of Non-Machine Learning Based Packer Identification/Classification Techniques

Type	Ref #	Year of Pub	Key Topics	Dataset Used	Dataset Size	Number of packers	Feature sets	Results obtained/ Measurement Metric used
Dynamic	[97]	2017	Using signatures extracted from CFGs of files to identify packers	Custom	140 packed files (20 for each packer)	7	CFG signatures	100% accuracy
	[100]	2018	Bare Metal unpacking framework that unpacks the packed binary and monitors the execution through Import Address Table Hooking	Hupigon Malware sample	Hupigon malware sample packed using 27 packers. (27 packed malicious files)	27	NA	BareUnpack successfully unpacked the packed files of 26/27 packers. Outperformed PolyUnpack, Renovo, OmniUnpack, CoDisasm, PINDemonium
Static	[21]	2007	Entropy analysis to detect packed malware	Custom	100 Windows PE files	3	Entropy Levels - Highest and Average for each file	99.99% confidence intervals for average and highest entropy levels calculated for each group of files (plain text, native executables, packed executables and encrypted executables
	[22]	2012	Similarity measures applied to packed malware without the need to unpack	Anubis	795,000 malware files	7	Similarity between bi-grams of the code section (stripped of unpacking noise)	Metrics - TPR, FPR Family Granularity thresholds - 95.2%, 0.3% Version Granularity thresholds - 91.3%, 0.2%
Hybrid	[30]	2012	Application level emulator for unpacking and CFG matching for classification of malware	Custom	15000 malicious files	14	String signatures for identifying flowgraphs	Unpacking method was able to detect OEP perfectly in almost all cases (with the exception of aspack)
	[26]	2018	Kernel level DLL hijacking and system call monitoring for binary unpacking	VX Heaven, VirusShare, VirusTotal	271,095 files	29	System API calls obtained through kernel level DLL hijacking	Demonstrated capabilities to unpack a large set of packed malware in an efficient manner

unable to unpack three samples within their database. Table 8 compares various non-Machine Learning based techniques that are used for the purpose of detecting/classifying packing methods.

4.2 Machine Learning based Packer Identification/Classification Techniques

Machine learning (ML) is a branch of artificial intelligence aimed at the development of computer algorithms which automatically learn (e.g., construct a mathematical model) from given data (a.k.a. "training data") and are able to make decisions or predictions on new unseen data. ML-based computer systems progressively enhance their performance on a specific task over time (not always, but that's the goal).

ML algorithms have already been used in many research fields [3, 30, 72]. For example, in the domain of malware detection, ML was found to be effective at detecting malicious executable files [72–75] and non-executable files of various types [76–81] when using static and dynamic analysis.

ML methods can be particularly effective in the classification of packed files and in distinguishing between the levels of complexity of the packed program. Fleshman et al. [55] found that

machine learning classifiers are less effective at classifying packed benign files, while antivirus products are less effective at classifying packed malware. However, this disparity is most likely a byproduct of imbalanced distributions of packed versus unpacked files in the training set containing benign and malicious files.

Advanced methods for the detection of new packed files are based mainly on classifiers induced by ML algorithms that leverage data from features extracted from the program or application either employing static or dynamic analysis. For instance, in order to detect packed PE files, domain expert defines a set of features which might distinguish between packed and unpacked files statistically. These features are used to induce an ML classification model which is trained on labeled data and then is able to predict new unseen data. The output of the model can be calibrated to achieve any desired false positive ratio [55].

4.2.1 Static Analysis. In 2008, Perdisci et al. [10] applied pattern recognition techniques in order to classify single-layer packed and non-packed executables files by combining nine features, such as: number of standard and non-standard sections, number of executable sections, number of entries in the PE file's IAT, PE header entropy, and code/data/file section entropy. The dataset included 5,498 executables in PE format. The data contained 2,598 packed viruses from the Malfease Project dataset, 2,231 non-packed benign executables, and 669 generated packed benign executables of 17 different executable packing tools freely available on the Internet. This method achieved high accuracy (above 95%) using five classifiers, including the NB, J48 decision tree, bagged, KNN, or **multilayer perceptron (MLP)**. Nevertheless, this approach has several limitations, the first of which is its inability to determine what family of packers a packed file belongs to. In addition, this technique cannot unpack unknown packed files, because the researchers used universal-unpacking tools as an unpacking mechanism. Furthermore, this approach is unable to identify the packer used.

In 2010, Sun et al. [3] presented a simple, fast, and effective packer classification framework which utilizes pattern recognition techniques on randomness profiles of packers. A randomness profile is a set of unique features of the packers which is mainly based on the packer's randomness signature. In order to classify packers, the authors first evaluated the randomness signature's scanning techniques using the sliding windows randomness test (involves building a byte frequency histogram and a Huffman tree which enables the calculation of the length of the corresponding data's code) that was first introduced by [82]. In a randomness signature-based packer classification system, a packer signature is simply calculated as the average value of a set of randomness profiles of training files. The distance between the tested file and each packer is calculated, where a shorter distance indicates file structure similarity to the packer structure. The framework was tested on a large dataset, including 583 benign packed files and 17,336 real malware samples. The samples were packed by different versions of packers, such as UPX, FSG, PETITE, and NSPACK, with different levels of complexity. The authors used four machine learning classification algorithms: **K-Nearest Neighbor (KNN), Best-First Decision Tree (BDT), Sequential Minimal Optimization (SMO)**, and **Naive Bayes (NB)**. Experimental results show that three of the four algorithms achieved a TPR of 99% or above. The Naive Bayes algorithm had the lowest TPR (around 94%), while the K-Nearest Neighbor classifier (with k = 1) had the best overall performance (a TPR of 99.6% with a FPR of 0.1%). The experiments also revealed that the randomness profiles used in the system are effective features for packer classification. It can be applied with high accuracy on real malware samples. This framework also showed that a low randomness profile of a packed file which is usually produced by the PE header and the unpacking stub contain important packer information. It is therefore very useful for distinguishing between families of packers. Since this is a signature packer detection approach, it is unable to measure every packed file. Furthermore, because some packers protect the packed section after packing the given files, this framework cannot be used to detect unknown packers.

In 2011, Santos et al. [65] proposed a method for the detection of packed executables. This method applies machine learning approaches trained on features extracted statically from the executables. Collective classification was applied using the following collective classifiers: Collective IBK, Collective Forest, Collective Woods & Collective Tree, and Random Woods. The extracted features can be divided into four main categories: PE file header characteristics, section characteristics, characteristics of the section containing the entry point, and entropy values. The authors also measured each characteristic's relevance based on the **information gain (IG)**, which provides a score for each feature that measures its importance for the classification task (packed/not packed). The dataset collection contained 1,000 non-packed executables and 1,000 packed executables. The experimental results reveal that the Collective Forest classifier achieved the highest accuracy, obtaining a TPR of 99.9%, FPR of 0.3%, and AUC of 99.9% while using 10% of the training dataset.

In spite of developments in deep learning, there are still some classical ML based solutions for packer identification and classification such as SVMs [93, 94] and Random Forests [95] that have been effective.

In 2018, Fleshman et al. [55] applied n-gram and neural networks to process the file byte in order to show that machine learning classifiers are more robust to evasion techniques (such as non-destructive automated binary modification, obfuscation through packing, and malicious injection) than existing antivirus software. A large dataset, consisting of two million training samples and a test set of 80,000 samples was used. Their results show that for the task of detecting obfuscation through packing, n-gram achieved a TPR of 98.7%, while the average TPR of the four AV engines examined was 84.2%, demonstrating that packed files are the major challenge for pure static analysis systems, such as antivirus packages.

Also in 2018, with the advent of the attention-based models [104], newer models in deep learning that took into account the importance weights of the input (i.e., the contextual information that highlights the important parts of the input to the model) led to derivative solutions in related fields. Notably, Yakura et al. [103] made use of an attention-based CNN in order to classify image representations of executable binaries. Considering that this paper dealt with a multi-class classification task that had over 500 independent classes of malware families and packers, a Top-5 error rate (Error in classification when the actual class is not present in the top five predictions) of 31% is commendable.

Among the latest works that have applied deep learning to develop static analysis systems capable of identifying a classifier include the use of a CNN by Gibert et al. [98] and the use of LSTMs by Bergenholtz [96] et al. All these applications, while indicative of progress and interdisciplinary application of concepts to solve problems, also highlight the fact that it is possible to find representation methods that are unique and add greater value to the classification score. It behooves every academician in the field to begin to not only target achieving higher results by mere margins, but rather to come up with newer ways of creating representations of these files that may shed new light on distinctions between one packer and another/one malware and another/a packed file and a non-packed file.

4.2.2 Dynamic Analysis. In 2017, Bat-Erdene et al. [6] described a method designed to classify packing algorithms of a given executable file into three categories: single-layer packing, repacking, or multi-layer packing. In order to classify between these three categories, entropy values of the executable file which were loaded into volatile memory are converted to symbolic representations, using **symbolic aggregate approximation (SAX)**. The entropy value of each section was dynamically calculated. During the unpacking process, they utilize a decompression module to unpack the packed instructions. The executable is executed and keeps running until a JMP instruction is encountered. In other words, the unpacking process is finished when the OEP is found. Since

detecting packing algorithms directly through the entropy values is difficult due to the large size of the data, a symbolic representation pattern based on entropy patterns was used, and a comparison threshold was used to compare patterns. The experiment dataset included 2,196 programs and 19 packing algorithms. The results demonstrated the method's ability to identify packing algorithms with high precision (97.7%), and a TPR of 96.8%.

A very recent study in 2020 by Hua et al. [102] dealt with the use of Deep Graph Convolutional Neural Networks for the purpose of classifying Control Flow Graphs of packed malware (and not the packing methods). The method involved executing the packed malware in a sandbox and extracting its execution stack of function calls. From this, the section of the commands that pertains to the unpacking routines is stripped and only the sub-graph that pertains to the functions calls of the packed malware itself is used as input to the DGCNN.

A similar approach at reducing the noise generated by obfuscators and their associated system calls can be seen in a study by Zhang et al. [101] where Information Gain was used to select "sensitive" system calls from the full set of system calls that was extracted from the execution stack of the packed malware in a sandbox. These system calls were then used as the basis to generate bi-gram representations which were then fed to a **Deep Belief Network (DBN)** that functioned as the detector which differentiated between packed and unpacked malware. Table 9 compares various Machine Learning based techniques that are used for the purpose of detecting/classifying packing methods.

4.2.3 Dynamic Analysis. In 2013, Bat-Erdene et al. [1] proposed a method to identify the packing algorithms of given known or unknown packed executables. The first phase, as we saw in the previous study [6], is to convert the entropy values of the packed executables loaded in memory into symbolic representations, using SAX. Then, the distribution of symbols are classified using popular machine learning classifications, such as Naive Bayes and **Support Vector Machines (SVMs)**. The dataset included a diverse collection of 466 programs and 15 packing algorithms. The results show a high accuracy of 92.7%. This approach enriches the method presented in [10], since this classification system extracts the types of packing algorithms from packed PE files and unpacks any inner packed executable files using entropy analysis.

In 2017, Bat-Erdene et al. [19] proposed a method for classifying unknown packing algorithms of given packed executables (benign or malware programs). Using SAX, they converted entropy values to symbolic representation, and then, classified the distribution of the symbols using learning classification methods, such as Naive Bayes. The dataset included a collection of 326 packed malware programs with 19 packing algorithms and 324 packed benign programs. The results show high accuracy of 95.35% and a TPR of 95.83%.

5 SUMMARY OF PACKING IDENTIFICATION AND CLASSIFICATION TECHNIQUES

Figure 9 provides a comparison of the capabilities of existing academic methods and tools. Our comparison can answer some interesting questions regarding the proposed methods and tools. For example, which of the methods provide unpacking methods and which of the methods perform packer identification, etc. The figure also indicates whether the suggested method is effective with both known and unknown files, i.e., whether the system can deal with files that are unknown (i.e., files and file variants that were not handled by the technique during training or in the deterministic solution). Usually, methods that apply a machine learning approach are able to deal with new unknown files, while methods that are signature-based are unable to do so. The table also indicates the methods' ability to perform automated unpacking of the executable before analyzing it and their ability to distinguish between levels of protection, such as unpacked, compressed, encrypted, and multi-layer encrypted code. For each technique or tool, we also specify its dedicated operating

Table 9. Comparison of Machine Learning Based Packer Identification/Classification Techniques

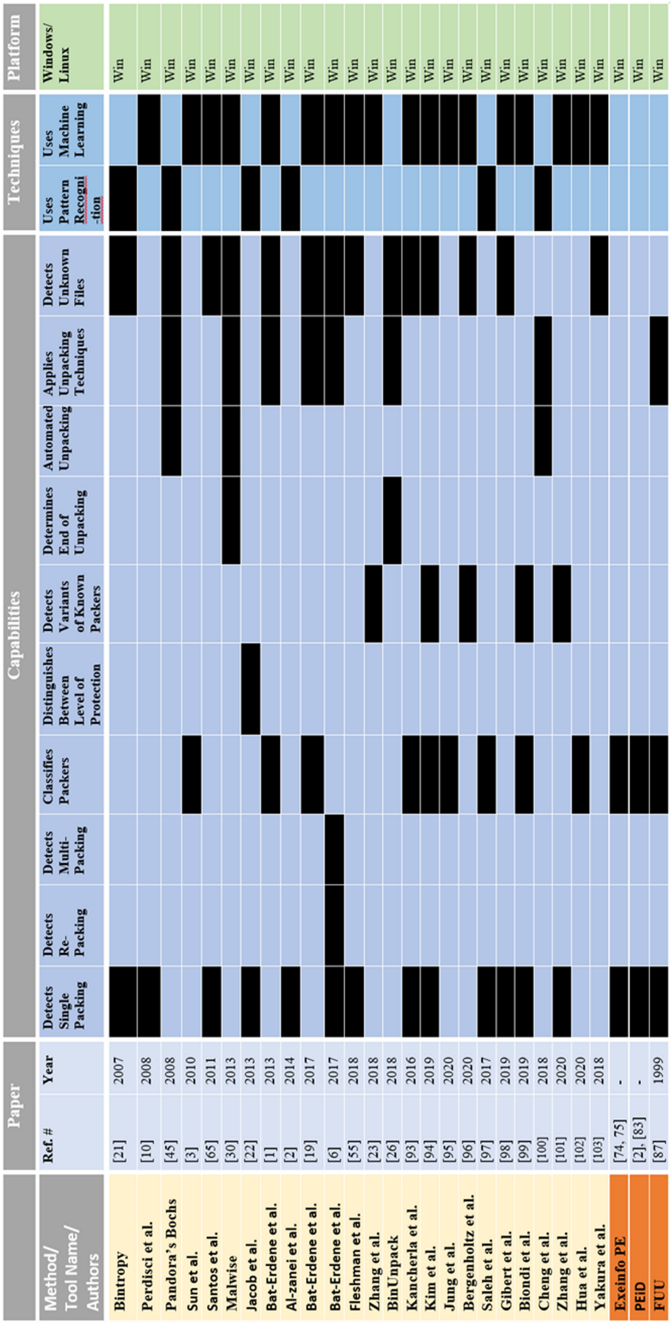
Type	Ref #	Year of Pub	Key Topics	Classifier	Dataset Used	Dataset Size	Number of packers	Feature sets	Results obtained/ Measurement Metric Used
Static	[93]	2016	The malware is first converted to byte plot and Markov plot. Out of this representation, gabor and wavelet based features are extracted and are used with SVMs for detecting the packing method.	SVM, RF	Offensive Computing Open Malware	5,000 files each for 9 packers	9	Intensity Based (6 features), Wavelet based (16 features) and Gabor based (512 features) features extracted from Byte Plot and Markov plot of each file	Metric - Accuracy MultiClass classification For Markov Plot: RF - 78.25% SVM - 81.34%
	[3]	2010	Packer classification using randomness profiles	Naïve Bayes, SMO, BFTree, KNN	Custom	17,802 packed malware 117 packed benign files	9	Randomness Profiles	Metric - TPR, FPR Naïve Bayes - 93.9%, 1.1% SMO - 98.9%, 0.8% BFTree - 99.3%, 0.5% KNN - 99.6%, 0.1%
	[94]	2019	Using an SVM having kernel lifting with binary diffing measures to classify packers	SVM	Custom	15,865 packed benign windows PE files	9	Longest Common Substring (LCS), n-gram Similarity, NH Similarity - all generated from the first 15 bytes after the entry point of the binary	Metric - Accuracy SVM (RBF kernel + LCS) - 98.41% SVM (RBF kernel + n-gram) - 98.20%
	[95]	2020	Encrypted sections of a file are selected based on entropy scores and are utilized for generating byte frequency and entropy sequence feature vectors	RF, Gradient Boosting	Custom	648 unpacked files and 3,587 packed files	6	Entropy Sequence analysis, Byte frequency analysis	Metric - Accuracy RF - 83.1% GB - 81.1%
	[96]	2020	Using RNNs to classify packers	LSTM	Custom	74,084 files (42 packers applied to 1,904 Windows executables)	42	Representation learning using LSTMs	Metric - Accuracy Exp 1 - Training on one kind of packer and evaluating on the rest of the set - 89.36% Exp 2 - Training on N-1 packers and evaluating on the excluded packer - 99.69%
	[98]	2019	Using CNNs for packer identification	CNN	MalImg, Microsoft Malware Classification Challenge	MalImg – 9,342 files Microsoft – 21,741 files	1	Gray scale image representations of malware binaries	97.5% accuracy
	[99]	2019	Feature selection process to improve efficiency of the packer detection and classification process while maintaining high accuracy and relevance in terms of trend shift	Decision Tree	Custom	281,344 binaries	22	Byte Entropy, Entry bytes of the file, Import Functions, Metadata, Resource (.rdata and .rsrc) , Section based data.	Decision tree achieved best cost to F-score ratio

(Continued)

Table 9. Continued

Type	Ref #	Year of Pub	Key Topics	Classifier	Dataset Used	Dataset Size	Number of packers	Feature sets	Results obtained/ Measurement Metric Used
Static	[103]	2018	Using Attention mechanism with CNNs to classify image representations of binaries	CNN with attention	VX Heaven	147,803 malware binaries	542 families of malware (not necessarily packed)	Binary image representations of binaries	Top 5 error rate of 31.34%
	[10]	2008	Creating a 9 feature pattern vector to identify packed executables	Naïve Bayes, J48, Bagging ensemble of J48, KNN, MLP, Entropy Threshold Classifier	Malfease	5,498 executables : 2,598 packed malicious 669 packed benign 2,231 non-packed benign	17	1. # standard and non-standard sections 2. # executable sections 3. # readable/writable/sections 4. # entries in the IAT 5. PE header, code, data and file entropy	Metric – Accuracy Naïve Bayes – 97.11% J48 – 97.01% Bagged J48 – 96.82% KNN – 95.62% MLP – 98.91% Entropy Threshold – 91.74%
	[65]	2011	Collective classification for packed executable identification	Collective Forest	VX Heavens	1,000 packed executables 1,000 non packed executables	10	DOS Header, File header block, Optional Header block, Section characteristics, Section of entry point characteristics, Entropy values (44 features in total)	Accuracy - 99.8%
Dynamic	[6]	2017	Classifying a packing algorithm into single layer packing, repacking and multi-layer packing	Fidelity and similarity measurement based classifier	Custom	2196 programs	19	SAX representations	Metric - Accuracy Repacking - 98.5% Multi-layer Packing - 97.5%
	[101]	2020	Packed malware variants detection using Deep Belief Networks	DBN	VX Heaven	3,100 malicious, 2,895 Benign binaries	4	Bi-gram representations of system calls. Information gain used to select "sensitive" system calls	Accuracy - 92.50%
	[102]	2020	Classifying CFGs of packed malware using DGCNNs	DGCNN	Microsoft Malware Classification Challenge (MSKCFG)	10,868 malicious binaries	9 malware families	Function call graph of the PE files	Accuracy - 99.25%
Hybrid	[1]	2013	Symbolic Aggregate Approximation (SAX) representation of entropy values for packer classification	Naïve Bayes, SVM	Custom	466 files	15	SAX representations	Accuracy -94.2%
	[19]	2017	Four new similarity measures based on SAX representations of file entropy values	Naïve Bayes, SVM	Custom	324 benign executables, 326 malicious executables	19	SAX representations and incremental aggregate analysis	Accuracy - 92.9%

system (Windows or Linux). Each black cell in the table indicates that the attribute specified in the column header applies for the particular method specified in the row. In the first column, academic studies appear in yellow, while tools appear in orange. Table 10 provides insight into the packers that some of the main studies surveyed countered; note that the documentation for the tools does not mention specifically which packers they are capable of dealing with, and thus tools are not presented in the table. In Table 10, cells marked with a check mark "v" and highlighted in green



Method/ Tool Name/ Authors	Paper: Ref. #	Paper: Year	Capabilities: Detects Single Packing	Capabilities: Detects Re-Packing	Capabilities: Detects Multi-Packing	Capabilities: Classifies Packers	Capabilities: Distinguishes Between Level of Protection	Capabilities: Detects Variants of Known Packers	Capabilities: Determines End of Unpacking	Capabilities: Automated Unpacking	Capabilities: Applies Unpacking Techniques	Capabilities: Detects Unknown Files	Techniques: Uses Pattern Recognition	Techniques: Uses Machine Learning	Platform: Windows/ Linux
Bintropy	[21]	2007	■									■	■		Win
Perdisci et al.	[10]	2008	■											■	Win
Pandora's Bochs	[45]	2008									■	■	■		Win
Sun et al.	[3]	2010				■								■	Win
Santos et al.	[65]	2011	■									■		■	Win
Malwise	[30]	2013							■	■	■	■		■	Win
Jacob et al.	[22]	2013	■				■						■		Win
Bat-Erdene et al.	[1]	2013				■					■	■		■	Win
Al-zanei et al.	[2]	2014	■										■		Win
Bat-Erdene et al.	[19]	2017				■	■				■	■		■	Win
Bat-Erdene et al.	[6]	2017	■	■	■						■	■		■	Win
Fleshman et al.	[55]	2018	■	■	■							■		■	Win
Zhang et al.	[23]	2018						■						■	Win
BinUnpack	[26]	2018							■		■	■			Win
Kancherla et al.	[93]	2016	■			■						■		■	Win
Kim et al.	[94]	2019	■			■		■				■		■	Win
Jung et al.	[95]	2020				■								■	Win
Bergeuholtz et al.	[96]	2020						■				■		■	Win
Saleh et al.	[97]	2017	■			■							■		Win
Gibert et al.	[98]	2019	■									■		■	Win
Biondi et al.	[99]	2019	■			■		■						■	Win
Cheng et al.	[100]	2018								■	■		■		Win
Zhang et al.	[101]	2020	■					■						■	Win
Hua et al.	[102]	2020				■								■	Win
Yakura et al.	[103]	2018										■		■	Win
Exeinfo PE	[74, 75]	-	■			■									Win
PEiD	[2], [83]	-	■			■									Win
FUU	[87]	1999	■			■					■				Win

Fig. 9. Comparison of methods and tools aimed at the detection of packed PE file.

Table 10. Packers vs. Leading Studies

Study / **Packer**	[93]	[94]	[95]	[96]	[97]	[99]	[100]	[101]	[26]	[21]	[3]	[65]	[22]	[1]	[19]	[6]	% of Studies that counter the packer
ACPROTECT							v		v								12.5%
ALLAPLE													v				6.25%
ALTERNATE_EXE															v	v	12.5%
ARMADILLO							v		v								12.5%
ARMPROTECTOR				v													6.25%
ASPACK	v	v	v			v	v	v	v					v	v	v	62.5%
ASPROTECT		v				v	v		v		v	v		v			43.75%
ENIGMA		v		v			v		v								25%
EXECRYPTOR				v	v												12.5%
EXPRESSOR					v	v	v		v								25%
FISHPACKER									v								6.25%
FSG	v				v	v	v		v		v	v	v	v	v		62.5%
KBYS									v								6.25%
MEW	v					v	v		v	v	v			v	v	v	56.25%
MOLEBOX							v		v								12.5%
MORPHINE				v						v					v		18.75%
MPRESS		v	v											v	v		25%
NPACK				v			v		v					v	v	v	37.5%
NSPACK						v	v		v		v		v	v	v	v	50%
OBSIDIUM		v		v			v		v								25%
ORIEN							v		v								12.5%
PACKMAN						v						v					12.5%
PECOMPACT						v	v		v		v						25%
PELOCK							v		v						v		18.75%
PENINJA				v													6.25%
PETITE			v	v			v		v		v			v			37.5%
PEP									v								6.25%
PESPIN							v		v								12.5%
POLYENE				v									v				12.5%
RLPACK							v		v			v		v	v	v	37.5%
SLV												v					6.25%
SOFTWAREPASSPORT							v		v								12.5%
TELOCK							v		v			v	v	v	v		37.5%
THEMIDA		v		v	v	v	v		v		v	v		v	v	v	68.75%
UPACK			v			v					v						18.75%
UPX	v	v	v		v	v	v	v		v	v	v	v				68.75%
UPX-iT															v		6.25%
UPXN									v						v		12.5%
VMPROTECT		v		v				v						v	v	v	37.5%
WINUPACK							v		v				v				18.75%
YODACRYPTOR				v	v		v		v				v	v	v		43.75%
YODAPROTECTOR							v		v								12.5%
ZPROTECT							v	v	v								18.75%
% of packers covered by the Study	9.3%	18.6%	11.6%	27.9%	13.9%	25.5%	60.4%	9.3%	67.4%	6.9%	20.9%	18.6%	18.6%	30.23%	37.2%	18.6%	

indicate that the corresponding packing detection method successfully detects the corresponding packer. The % of studies that counter the packer, which appears in the last column, is highlighted in red for values that are under 20%, in yellow for values between 20% and 50%, and in green for values over 50%. Similarly, in the last row in the table where the % of packers covered by the study is mentioned, red highlighting is used for values under 20%, yellow is used for values between 20% and 50%, and green is used for values over 50%. Some interesting insights derived from the information are presented in the table.

The studies considered in the table used at least three packers in their research. The packers included are some of the most commonly used packers for packing malware in the wild. While this list is certainly not exhaustive in terms of the packers that are available in the wild, it

Table 11. Comparison of the Requirements and Capabilities of the Surveyed Methods

	Requirements									Capabilities						
Study	CFG generation	File disassembly	Signature generation	Bytewise analysis of file	Entropy analysis	Dynamic Analysis	Application-Level Emulation	API Hooking	No Requirement(works as is)	Handling Junk Code insertion	Handling Code Virtualization	OEP Detection	Detection of Polymorphism	Combats API Redirection	Combats DLL Hijacking	Classifies Unknown Packer
[97]	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N
[100]	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N
[21]	N	N	N	N	Y	N	N	N	N	N	N	N	N	N	N	N
[22]	N	N	N	N	N	N	N	N	Y	Y	N	N	N	N	N	N
[30]	Y	N	Y	N	Y	Y	Y	N	N	N	N	Y	Y	N	N	N
[26]	N	N	N	N	N	Y	N	Y	N	Y	Y	Y	Y	Y	Y	N
[93]	N	N	N	Y	N	N	N	N	N	M	Y	N	M	-	-	N
[3]	N	N	N	Y	N	N	N	N	N	M	Y	N	M	-	-	N
[94]	N	N	N	Y	N	N	N	N	N	M	N	N	M	-	-	N
[95]	N	N	N	Y	Y	N	N	N	N	N	N	N	N	-	-	N
[96]	N	Y	N	N	N	N	N	N	N	Y	M	N	N	-	-	N
[98]	N	N	N	Y	N	N	N	N	N	Y	N	N	N	-	-	N
[99]	N	N	N	Y	Y	N	N	N	N	Y	N	N	M	-	-	N
[103]	N	Y	N	Y	N	N	N	N	N	N	N	N	N	-	-	N
[10]	N	Y	N	N	Y	N	N	N	N	M	N	N	N	-	-	N
[65]	N	N	N	Y	Y	N	N	N	N	N	N	N	N	-	-	N
[1]	N	N	N	N	Y	Y	N	N	N	M	N	Y	N	N	N	Y
[101]	N	N	N	N	N	Y	N	Y	N	Y	N	N	N	M	M	N
[102]	Y	Y	N	N	N	Y	N	N	N	Y	N	Y	N	M	M	N

provides insight into the packers available to the research community. The percentage of studies that counter a specific packer indicates its popularity among malware datasets in the wild. The percentage of common packers that each of the studies cover, on the other hand, indicates the extent of coverage of different packer families in the datasets used in each of the studies. As can be seen, the method proposed by Cheng et al. [26] counters the maximum number of different packers, with coverage of 67%, while the UPX and Themida packers are those that most studies (68%) can counter.

Table 11 presents a comparison between the surveyed studies along the lines of the requirements that each of these studies have before they can either classify or detect packing and the capabilities that each of these studies have at defeating commonly used analysis evasion techniques by packers.

In the table, "Y" indicates that the study either requires the listed Requirement or has the listed Capability in the respective columns. "N" indicates the opposite. "-" indicates that the attribute of comparison does not apply to the corresponding study and "M" indicates that the study may have the corresponding Requirement/Capability.

From the table, it becomes clear that most of the methods combat some form of packing evasion, while [26] combats almost all the evasion methods and falls short only for being unable to classify unknown packers. The insight to be had from the requirements that each of these methods have is the overhead that each requirement has to analysis. So, the interpretation to be had is that the more the requirements that a method has, the longer it takes to analyze a file.

Table 12 that follows Table 11, presents the shortfalls of each of the surveyed methods. It can be seen that many of the works that implement dynamic analysis seem to work under the assumption that the packed file will not use runtime evasion to avoid detection and exposure of the packed content. Parallelly, several of the static analysis methods fail to consider the performance of the methods on encrypted files. Several methods also implement solutions that are susceptible to input file size bias.

6 DISCUSSION

Although packing is considered the most legitimate means of protecting proprietary code and is widely used for this purpose, it has also become a preferred obfuscation method of malware authors wishing to avoid malware payload detection. In this survey, we introduce the file packing phenomenon and provide the background needed to understand the domain of packed portable executable (PE) files. We provide a comprehensive description of the PE file structure along with a review of common packing and unpacking techniques and their effect on the PE file structure. We review research in the unpacking domain and present a comparative taxonomy and analysis that highlights the differences in the capabilities of each method.

We gained insight on the malware packing trends between 2010 to 2021 by scanning 2,000 randomly selected PE files from VirusTotal for each year. There was very little information available regarding these trends, and our analysis shed light on the actual trends seen in the wild. We found that on average, 21.35% of all files between 2010 and 2021 were packed.

In retrospect, it seems that while the use of malware packing has had upticks and downticks over the years, its use has increased in the recent past as accurate and machine learning-based detection mechanisms [124, 125] were developed to cope with malicious non-executable files, which are an alternative means of launching malware attacks. This points to the need to both better understand the extent of packing and its malicious use and develop solutions capable of preventing attackers from using packing to perform attacks.

Thus, the detection of packed files and the identification of the packer used can be a key capability in detecting both known and new unknown malware and result in enhanced computer security. In addition, the number of newly created packers and their complexity are constantly increasing, and organizational anti-malware solutions must keep pace and improve their ability to efficiently deal with new incoming, customized, and suspicious packed executables.

When considering the Windows OS, it is important to determine whether a PE has been packed as a first step when analyzing a large number of unknown PE files. Some characteristics of PE files, such as a small file size, a small number and/or names of the sections, and a small number of import functions may serve as good indicators in heuristic packer identification. Moreover, the PE header contains basic information about the file itself and more valuable information (e.g., metadata) for malware analysis; this information might be helpful in detecting packers. In addition, the PE entropy or packer code signatures can be used to determine whether the file is packed.

Based on our analysis, about 70% of existing academic research has applied different techniques to unpack the code using machine learning algorithms. The use of machine learning algorithms has paved the way for the detection of known and unknown packers, and they are far more dependable than traditional signature-based methods, which are far too unidimensional and inflexible when

Table 12. Limitations of Each of the Surveyed Methods

Work	General limitations	Method subject to file size bias	Method cannot handle encryption
[97]	The proposed method fails if indirect addressing is used near original entry point.	No	Yes
[100]	Data extraction fails if the file being analyzed checks for GetProcAddress API hooking.	No	No
[21]	Entropy scores become less dependable when the files being analyzed are larger than 500 Kilobytes.	Yes	No
[22]	The proposed method can only detect (and classify) packing if the packer is known and code similarities exist at the binary level.	No	No
[30]	-	No	No
[26]	The proposed method does not fully reconstruct the rebuilt IAT.	No	No
[93]	The proposed method works on the premise that the byteplot image generated from the file being analyzed sufficiently captures all data. No insight regarding file size vs performance of the detector was provided.	Yes	Unclear. Needs experimentation to see if an excessive number of false positives are generated.
[3]	-	No	No
[94]	The work does not elaborate on junk code handling which is essential for establishing the robustness of any static packed file analysis system. Also, the time taken for classification seems impractical.	No	Unclear. Needs experimentation.
[95]	Encrypted section selection is flawed – the section with the highest entropy is determined to be the encrypted section (while this may be true in many cases, having this as an underlying assumption of the method makes it vulnerable to trivial attacks where a crafted sample with higher entropy in a benign section is incorrectly analyzed, since the method analyzes only the section having the highest entropy).	No	Yes
[96]	The method is too heavy just for being able to determine whether the PE file is packed.	No	Yes
[98]	The proposed method fails on encrypted samples.	Yes	Yes
[99]	The proposed method fails on repacked malware and partially encoded malware.	No	Unclear. Needs experimentation.
[103]	Proposed 2D CNNs may be vulnerable to junk code insertion.	Yes	Yes
[10]	The proposed method is heavy for detecting whether a binary is packed.	Yes	No
[65]	The extracted features can be manipulated by adversarially constructed sample constructed by malware authors to bypass the proposed packing detection method.	No	No
[1]	The proposed method fails if the packer uses runtime evasion routines.	No	Yes
[101]	The proposed method fails if the packer uses runtime evasion routines.	No	Yes
[102]	Method fails if packer uses runtime evasion routines.	No	Yes

it comes to generalizing to malware variants. Machine learning techniques give generalizability to the classifiers trained to detect malware variants and packing, thus allowing them to function more dependably when encountering a new, unseen case.

One of the main disadvantages of packer identification schemes proposed in prior work is that very few studies have considered the case of custom packers. There are many malware samples in the wild that use custom packers to pack their payloads. This is problematic, because most packer identification studies use well-structured datasets comprised of malware packed using well-known and documented packers. This results in a knowledge gap in that the performance of most of these methods is questionable when it comes to the detection of malware that is packed using custom packers. Consequently, even if these detectors are effective at determining whether the executable is packed, they are face a challenge in using an unpacking scheme that will dependably unpack a file for further analysis in order to determine whether the file is malicious or not. This introduces a possibility for triggering higher rates of false positives among legitimate, packed executables.

Also, the survey shows that not many studies aimed at developing schemes to combat the anti-analysis techniques that packed malware employ. Thus, there is a need for research on methods that are robust against packers that try to evade detection. A database of tagged packed malware that employ a variety of anti-analysis techniques would be of great value to the research community.

Regarding solutions in the packer identification space that use machine and deep learning, we see that there are several methods that suggest features or feature representations that are highly sensitive to file size. For example, consider the case of converting a packed file to its corresponding byteplot image; here, if the file size is sufficiently large, and the malicious payload is relatively small and takes up a comparatively minuscule portion of the overall file, it will more than likely be missed or disregarded in the overall classification as noise. Byteplot image representations are also a lossy method of data representation. Consider the fact that most deep learning methods whose input is an image require an image of fixed size. When represented as an image, a file over a certain upper-bound of bytes in file size tends to be resized to fit the constraints of the dimensions of the input image to the classifier. This often results in the loss of valuable data (through either aggregation or the dropping of pixels).

Not many methods seem to have considered side-channel analysis for packer identification. This may be worth investigating provided the right features are chosen.

This information gathered in this survey reveals that none of the surveyed methods or tools can do all of the following: (1) identify a wide range of different packers of various complexity levels (described in Figure 4); (2) identify the packing chain (in the case of multi-packing) without unpacking the file first; and (3) detect packing when the PE file is packed using unknown or custom packers.

In 2020, Aghakhani et al. [116] explored the inherent disadvantages of approaching packer classification using static analysis-based features. They showed that the features and signals extracted statically are insufficient for ML models to detect previously unseen packers and adversarial examples. They also demonstrated that static analysis-based machine learning products on VirusTotal have high false positive rates for packed binaries. At the same time, there are several malware binaries that go undetected on VirusTotal because they were packed by custom packers.

Consequently, we propose establishing a temporal machine learning-based framework aimed at packed file identification and packer classification. A two-stage solution is suggested; in the first stage, entropy-based static analysis will be used to determine whether the examined PE file is packed. This is, of course, dependent on the assumption that an entropy-based method will be sufficient for identifying a packed PE file. A recent work by Mantovani et al. [128] showed that many packing methods have adopted low-entropy generating packing techniques in order to avoid

additional scrutiny and detection by packing detection methods. As much as 31.5 % of low-entropy malicious PE files were shown to be packed. Keeping this in mind, we also propose engineering novel additional features that can be used to more effectively and accurately determine whether a PE file is packed or not, in addition to using the file's entropy measurement as a part of the first stage of our proposed solution. If the file is packed, the file will be analyzed dynamically in the second stage, in order to identify the specific packer(s) used to pack the PE file.

In addressing similar packing issues, Cheng et al. [1] also used static analysis as a first step and then used sequence mining to dynamically analyze system calls. We propose a different approach, applying machine learning algorithms instead. The contribution of using a machine learning mechanism (such as the TPF method that was recently developed for early sepsis detection in ICU patients) will allow us to detect and profile unknown packers - something that can only be achieved using machine learning.

6.1 Future Work

The significance of packing classification compared to determining whether a file has been packed or not is that the latter only deals with the trivial case, i.e., it cannot be used to identify the packers used to pack the file. There are many organizations that have a security policy of completely prohibiting packed files, however such a policy is not without limitations – namely, the loss of all benign packed files (the blocking of packed files would also limit the benign use of packing within an organization). This is the main reason for the importance of packing classification. Determining the type of packing used is always the first step in reverse engineering a packed file. If the packing technique/tool is identified, unpacking a file becomes easier; moreover, the true unpacked behavior of the file can then be analyzed, thereby eliminating the need to reject all packed files. There is also a critical need to implement solutions with the capability of identifying all of the packers used to pack a file (in the case of a multi-packed file).

The use of dynamic analysis for the purpose of unpacking files has always been viewed as much more computationally expensive than using static methods to unpack files. The inherent advantage that generic dynamic unpacking methods provide is that regardless of the packing scheme used (assuming that the file being analyzed has not used packing schemes that implement dynamic analysis evasion techniques, such as conditional runtime unpacking), they have a high chance of returning the unpacked version of the file from memory. While this eliminates the need for packing classification, it is a computationally expensive procedure that can cause significant delays when there is a need to scan a large number of files. Static unpacking techniques (which currently are largely based on heuristics and signature-based methods) are efficient when it comes to unpacking packed files but tend to fail when the packing technique used is unknown. Also, in the event that a packed file has been iteratively packed, the iterative static unpacking process is not as reliable and efficient as it is when it is used to unpack a file that was packed only once.

Thus, as a future research direction, we plan to explore a potentially efficient method of classifying the packer in which the initial unpacking routines are detected dynamically and halted once the import address table is reconstructed (using the IAT hooking technique or another appropriate technique). This process may even result in the detection of the use of more than one packing method – in the case of multi-packed files, thus greatly reducing the overhead that dynamic analysis imposes. Once the packing method is accurately identified, it would become easier to statically unpack the file. We plan to develop the "Time-Unpack" framework, designed to examine packed PE files, i.e., determine that the file is packed and classify the specific packer(s) used. If the file is found to be packed, the file will be analyzed dynamically during runtime to identify the specific packer(s) used to pack the PE file. We suggest only conducting the dynamic analysis procedure for the initial stage of the packed file's execution (which would mainly pertain to the unpacking

procedure). At this stage, the dynamic analysis process would be performed in an isolated sandbox (such as the Cuckoo sandbox), and the invoked API calls would be recorded during the unpacking process. The novelty of the proposed framework lies in the feature extraction method, which will be able to mine temporal patterns that take into account the API calls' time dimension and the interactions between the different API calls over time; this method (the TPF method), which was originally presented for improving early sepsis detection [88], will reveal discriminative time interval temporal patterns that represent time-based behavior of the file while it is being unpacked, in order to identify the specific packer, or group of packers, used. The TPF method learns a system's behavior during runtime, utilizing the data provided by various side-channel sensors which are measured continuously over time; we believe that the combination of the properties of the data from individual sensors, along with the interrelations between this data over time, could represent the behavior of file unpacking and, more specifically, represent the behavior that identifies an individual packer. Such side-channel sensors can monitor system calls, which provide an essential interface between a process and the OS, and thus represent various operations performed by the system, including reading a file, writing a file, encryption, etc.

ACKNOWLEDGEMENTS

We would like to thank Shlomi Boutnaru for his meaningful discussions regarding packing techniques and how they are being exploited by malware authors. We would also like to thank Eitam Sheetrit for his inputs on how to improve our proposed packing classification solution using the TPF classification algorithm. Lastly, we would like to thank Tomer Panker for his devoted assistance in the packed files acquisition process.

REFERENCES

[1] M. Bat-Erdene, T. Kim, H. Li, and H. Lee. 2013. Dynamic classification of packing algorithms for inspecting executables using entropy analysis. *Proc. 2013 8th Int. Conf. Malicious Unwanted Softw. The Am. MALWARE 2013*. 19–26.
[2] M. M. K. Al-Zanei. 2014. Generic packing detection using several complexity analysis for accurate malware detection 5, 1 (2014), 7–14.
[3] L. Sun, S. Versteeg, S. Boztaş, and T. Yann. 2010. *Pattern Recognition Techniques for the Classification of Malware Packers*. Springer, Berlin, 2010, 370–390.
[4] D.-I. M. Morgenstern and H. Pilz. Useful and useless statistics about viruses and anti-virus programs.
[5] Peter Ferrie, Senior Anti-virus Researcher, and Microsoft Corporation. 2008. Anti-unpacker tricks. *Current* (2008).
[6] M. Bat-Erdene, T. Kim, H. Park, and H. Lee. 2017. Packer detection for multi-layer executables using entropy analysis. *Entropy* 19, 3 (2017), 1–18.
[7] W. Yan, Z. Zhang, and N. Ansari. 2008. Revealing packed malware. *IEEE Secur. Priv. Mag.* 6, 5 (2008), 65–69.
[8] M.-J. Kim et al. 2010. Design and performance evaluation of binary code packing for protecting embedded software against reverse engineering. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing* 2010, 80–86.
[9] Y. Choi, I. Kim, J. Oh, and J. Ryou. 2008. PE file header analysis-based packed PE file detection technique (PHAD). In *International Symposium on Computer Science and its Applications 2008*. 28–31.
[10] R. Perdisci, A. Lanzi, and W. Lee. 2008. Classification of packed executables for accurate computer virus detection. *Pattern Recognit. Lett.* 29, 14 (2008), 1941–1946.
[11] PE Format | Microsoft Docs. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format. [Accessed: 04-Feb-2021].
[12] D. Devi and S. Nandi. 2012. PE file features in detection of packed executables. *Entropy* 4, 3 (2012), 476–478.
[13] M. Sikorski and A. Honig. 2012. Practical malware analysis: The hands-on guide to dissecting malicious software. No Starch Press.
[14] L. Sun, S. Versteeg, S. Boztaş, and T. Yann. 2010, July. Pattern recognition techniques for the classification of malware packers. In *Australasian Conference on Information Security and Privacy*. Springer, Berlin, Heidelberg, 370–390.
[15] D. M. Abhi Gupta and Akshi S. Arya. 2018. Hashing Base Ed Encryption N And Anti-Deb Bugger Suppor Rt For Packing Multiple Fi Es Into Sing E Executable (2018), 96–99.
[16] "I Executable and Linkable Format (ELF)."
[17] K. Muhammad and H. Zahid. 2015. ITEE Journal. *ITEE J.* 4, 4 (2015), 1–5.

[18] M. Hassnain and A. Abbas. 2017. ITEE Journal. *Int. J. Inf. Technol. Electr. Eng.* 6, 1 (2017), 10–16.
[19] M. Bat-Erdene, H. Park, H. Li, H. Lee, and M.-S. Choi. 2017. Entropy analysis to classify unknown packing algorithms for malware detection. *Int. J. Inf. Secur.* 16, 3 (2017), 227–248.
[20] B. Li, Y. Zhang, J. Li, W. Yang, and D. Gu. 2018. AppSpear: Automating the hidden-code extraction and reassembling of packed Android malware. *J. Syst. Softw.* 140 (2018), 3–16.
[21] R. Lyda and J. Hamrock. 2007. Using entropy analysis to find encrypted and packed malware. *IEEE Secur. Priv. Mag.* 5, 2 (2007), 40–45.
[22] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna. 2013. A static, packer-agnostic filter to detect similar malware samples. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer-Verlag, 2013, 102–122.
[23] J. Zhang, K. Zhang, Z. Qin, H. Yin, and Q. Wu. 2018. Sensitive system calls based packed malware variants detection using principal component initialized multilayers neural networks. 1–13.
[24] Entropy and the distinctive signs of packed PE files. | NTinfo. [Online]. Available: http://n10info.blogspot.com/2014/06/entropy-and-distinctive-signs-of-packed.html. [Accessed: 04-Feb-2021].
[25] I. J. Good, T. N. Gover, and G. J. Mitchell. 1970. Exact distributions for X 2 and for the likelihood-ratio statistic for the equiprobable multinomial distribution. 1970.
[26] B. Cheng et al. 2018. Towards paving the way for large-scale windows malware analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18*. 395–411.
[27] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. 2015. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*. 659–673.
[28] G. Bonfante et al. 2015. CoDisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. 2015.
[29] D. Bueno, K. J. Compton, K. A. Sakallah, and M. Bailey. Detecting traditional packers, decisively.
[30] S. Cesare, Y. Xiang, and W. Zhou. 2013. Malwise—an effective and efficient classification system for packed and polymorphic malware. *IEEE Trans. Comput.* 62, 6 (2013), 1193–1206.
[31] Ang Li, Yue Zhang, Junxing Zhang, and Gang Zhu. 2015. A token strengthened encryption packer to prevent reverse engineering PE files. In *2015 International Conference on Estimation, Detection and Information Fusion (ICEDIF'15)*. 307–312.
[32] L. Bilge, A. Lanzi, and D. Balzarotti. Thwarting real-time dynamic unpacking.
[33] J. Křoustek, P. Matula, D. Kolář, and M. Zavoral. Advanced preprocessing of binary executable files and its usage in retargetable decompilation.
[34] L. Durfina, J. Kroustek, and P. Zemek. 2013. PsybOt malware: A step-by-step decompilation case study. In *2013 20th Working Conference on Reverse Engineering (WCRE'13)*. 449–456.
[35] J. R. Levine. 2000. *Linkers and Loaders*. Morgan Kaufmann.
[36] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti. 2018. Understanding Linux malware. *Proc. - IEEE Symp. Secur. Priv.* 2018-May, 161–175.
[37] K. A. Roundy and B. P. Miller. 2013. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.* 46, 1 (2013), 1–32.
[38] UPX: the Ultimate Packer for eXecutables - Homepage. [Online]. Available: https://upx.github.io/. [Accessed: 10-Feb-2021].
[39] H. D. Menéndez, S. Bhattacharya, D. Clark, and E. T. Barr. 2019. The arms race: Adversarial search defeats entropy used to detect malware. *Expert Syst. Appl.* 118 (2019), 246–260.
[40] Manual Unpacking of UPX Packed Binary File - www.SecurityXploded.com. [Online]. Available: https://securityxploded.com/unpackingupx.php. [Accessed: 10-Feb-2021].
[41] Unpacking, Reversing, Patching. [Online]. Available: https://resources.infosecinstitute.com/unpacking-reversing-patching/#gref. [Accessed: 11-Feb-2021].
[42] Oreans Technology: Software Security Defined. [Online]. Available: https://www.oreans.com/themida.php. [Accessed: 11-Feb-2021].
[43] K. Coogan, S. Debray, T. Kaochar, and G. Townsend. 2009. Automatic static unpacking of malware binaries. In *2009 16th Working Conference on Reverse Engineering* 2009, 167–176.
[44] E. O. Osaghae. 2016. *Classifying packed programs as malicious software detected* 2016.
[45] L. Bohne. 2009. *Pandora's Bochs: Automatic Unpacking of Malware* 121, 2009.
[46] F. Guo, P. Ferrie, and T. Chiueh. 2008. A study of the packer problem and its solutions. In *Recent Advances in Intrusion Detection*. Berlin, Springer, Berlin, 2008, 98–115.
[47] S.-C. Yu and Y.-C. Li. 2009. A unpacking and reconstruction system-AGUnpacker. In *2009 International Symposium on Computer Network and Multimedia Technology*. 1–4.

[48] M. N. Gagnon, S. Taylor, and A. K. Ghosh. 2007. Software protection through anti-debugging. *IEEE Secur. Priv. Mag.* 5, 3 (2007), 82–84.
[49] P. Ferrie, S. A. Researcher, and M. Corporation. 2008. Anti-unpacker tricks. *Current.* 1–25.
[50] C. V. Liţă, D. Cosovan, and D. Gavriluţ. 2018. Anti-emulation trends in modern packers: A survey on the evolution of anti-emulation techniques in UPA packers. *J. Comput. Virol. Hacking Tech.* 14, 2 (2018), 107–126.
[51] E. Carrera and G. Erdélyi. 2004. Digital genome mapping. 2004.
[52] X. Hu, T.-C. Chiueh, and K. G. Shin. 2009. *Large-Scale Malware Indexing Using Function-Call Graphs* * †. 2009.
[53] A. Karnik, S. Goswami, and R. Guha. 2007. Detecting obfuscated viruses using cosine similarity analysis. In *First Asia International Conference on Modelling & Simulation (AMS'07).* 165–170.
[54] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia. Exploiting similarity between variants to defeat malware "Vilo" method for comparing and searching binary programs.
[55] W. Fleshman, E. Raff, R. Zak, M. Mclean, and C. Nicholas. Static malware detection & subterfuge: Quantifying the robustness of machine learning and current anti-virus.
[56] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. 2004. N-gram-based detection of new malicious code. *Proc. 28th Annu. Int. Comput. Softw. Appl. Conf. 2004. COMPSAC 2004* 2 (2004), 41–42.
[57] G. Wicherski. 2009. peHash: A novel approach to fast malware clustering. *2nd USENIX Work. Large-Scale Exploit. Emergent Threat.* 2009.
[58] O. Chum, J. Philbin, and A. Zisserman. 2008. Near duplicate image detection: Min-Hash and tf-idf weighting. In *Proceedings of the British Machine Vision Conference 2008*, 50, 1–50.10.
[59] W. Jin et al. 2012. Binary function clustering using semantic hashes. In *Proceedings - 2012 11th International Conference on Machine Learning and Applications, ICMLA'2012*, 1, 386–391.
[60] J. Crussell, C. Gibler, and H. Chen. 2013. Scalable semantics-based detection of similar Android applications. In *Esorics* 2013, 182–199.
[61] A. Akusok, Y. Miche, J. Hegedus, R. Nian, and A. Lendasse. 2014. A two-stage methodology using K-NN and false-positive minimizing ELM for nominal data classification. *Cognit. Comput.* 6, 3 (2014), 432–445.
[62] A. Tamersoy, K. Roundy, and D. H. Chau. 2014. Guilt by association. *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discov. Data Min. - KDD'14.* 1524–1533.
[63] C. Oprisa, M. Checiches, and A. Nandrean. 2014. Locality-sensitive hashing optimizations for fast malware clustering. In *Proceedings - 2014 IEEE 10th International Conference on Intelligent Computer Communication and Processing, ICCP'2014.* 97–104.
[64] Statistical Mechanics – R. K. Pathria, Paul D. Beale - Google בירפס." [Online]. Available: https://books.google.co.il/books?id=KdbJJAXQ-RsC&printsec=frontcover&redir_esc=y&hl=iw#v=onepage&q&f=false. [Accessed: 11-Feb-2021].
[65] I. Santos, X. Ugarte-Pedrero, B. Sanz, C. Laorden, and P. G. Bringas. 2011. Collective classification for packed executable identification. In *Proceedings of the 8th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference on - CEAS'11.* 23–30.
[66] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host.
[67] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos. 2011. *Combining Static and Dynamic Analysis for the Detection of Malicious Documents.* 2011.
[68] D. Maiorca, I. Corona, and G. Giacinto. 2013. *Looking at the Bag is not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection.* 2013.
[69] F. Schmitt, J. Gassen, and E. Gerhards-Padilla. 2012. PDF scrutinizer: Detecting Javascript-based attacks in PDF documents. In *2012 Tenth Annual International Conference on Privacy, Security and Trust.* 104–111.
[70] X. Lu, J. Zhuge, R. Wang, Y. Cao, and Y. Chen. 2013. De-obfuscation and detection of malicious PDF files with high accuracy. In *2013 46th Hawaii International Conference on System Sciences.* 4890–4899.
[71] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. 2006. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06).* 289–300.
[72] A. Cohen and N. Nissim. 2018. Trusted detection of ransomware in a private cloud using machine learning methods leveraging meta-features from volatile memory. *Expert Syst. Appl.* 102, 158–178.
[73] N. Nissim, Y. Lapidot, A. Cohen, and Y. Elovici. 2018. Trusted system-calls analysis methodology aimed at detection of compromised virtual machines using sequential mining. *Knowledge-Based Syst.* 153, (2018), 147–175.
[74] N. Nissim, R. Moskovitch, L. Rokach, and Y. Elovici. 2014. Novel active learning methods for enhanced PC malware detection in Windows OS. *Expert Syst. Appl.* 41, 13 (2014), 5843–5857.
[75] N. Nissim, R. Moskovitch, L. Rokach, and Y. Elovici. 2012. Detecting unknown computer worm activity via support vector machines and active learning. *Pattern Anal. Appl.* 15, 4 (2012), 459–475.

[76] A. Cohen, N. Nissim, L. Rokach, and Y. Elovici. 2016. SFEM: Structural feature extraction methodology for the detection of malicious office documents using machine learning methods. *Expert Syst. Appl.* 63 (2016), 324–343.
[77] N. Nissim, A. Cohen, C. Glezer, and Y. Elovici. 2015. Detection of malicious PDF files and directions for enhancements: A state-of-the art survey. *Comput. Secur.* 48 (2015), 246–266.
[78] N. Nissim, A. Cohen, and Y. Elovici. 2017. ALDOCX: Detection of unknown malicious microsoft office documents using designated active learning methods based on new structural feature extraction methodology. *IEEE Trans. Inf. Forensics Secur.* 12, 3 (2017), 631–646.
[79] N. Nissim et al. 2016. Keeping pace with the creation of new malicious PDF files using an active-learning based detection framework. *Secur. Inform.* 5, 1 (2016) 1.
[80] N. Nissim, A. Cohen, and Y. Elovici. 2016. Boosting the detection of malicious documents using designated active learning methods. *Proc. - 2015 IEEE 14th Int. Conf. Mach. Learn. Appl. ICMLA 2015.* 760–765.
[81] A. Cohen, N. Nissim, and Y. Elovici. 2018. Novel set of general descriptive features for enhanced detection of malicious emails using machine learning methods. *Expert Syst. Appl.* 110 (2018), 143–169.
[82] T. Ebringer, L. Sun, and S. Boztas. 2008. A fast randomness test that preserves local detail. *Virus Bull. (2008)*, 34–42.
[83] "PE iDentifier (PEiD) 0.95 /Binary Analysis/Editing/Downloads - Tuts 4 You." [Online]. Available: https://tuts4you.com/e107_plugins/download/download.php?view.398. [Accessed: 11-Feb-2021].
[84] M. G. Kang, P. Poosankam, and H. Yin. 2007. Renovo. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode - WORM'07.* 46.
[85] "Exeinfo PE 0.0.5.1 - Download." [Online]. Available: https://exeinfo-pe.en.uptodown.com/windows. [Accessed: 11-Feb-2021].
[86] "Exeinfo PE by A.S.L - packer - compression detector and data detector." [Online]. Available: http://exeinfo.atwebpages.com/. [Accessed: 11-Feb-2021].
[87] Google Code Archive - Long-term storage for Google Code Project Hosting. [Online]. Available: https://code.google.com/archive/p/fuu/. [Accessed: 21-Feb-2021].
[88] E. Sheetrit, N. Nissim, D. Klimov, and Y. Shahar. 1983. Temporal probabilistic profiles for sepsis prediction in the ICU. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining - KDD'19.* 2961–2969.
[89] J. F. Allen. 1983. Maintaining knowledge about temporal intervals. 1983.
[90] F. Song and W. B. Croft. 1999. A general language model for information retrieval. In *Proceedings of the Eighth International Conference on Information and Knowledge Management.* 316–321.
[91] S. Kullback and R. A. Leibler. 1951. On information and sufficiency. *Ann. Math. Stat.* 22, 1 (1951), 79–86.
[92] Which are the Linux Executable Files, and How do We Create Them? [Online]. Available: https://www.webhostinghero.com/blog/which-are-the-linux-executable-files-and-how-do-we-create-them/. [Accessed: 11-Feb-2021].
[93] Kesav Kancherla, John Donahue, and Srinivas Mukkamala. 2016. Packer identification using byte plot and Markov plot. *J. Comput. Virol. Hacking Tech.* (2016). DOI: https://doi.org/10.1007/s11416-015-0249-8
[94] Yeongcheol Kim, Joon Young Paik, Seokwoo Choi, and Eun Sun Cho. 2019. Efficient SVM based packer identification with binary diffing measures. In *Proceedings - International Computer Software and Applications Conference.* DOI: https://doi.org/10.1109/COMPSAC.2019.00117
[95] Byeong Ho Jung, Seong Il Bae, Chang Choi, and Eul Gyu Im. 2020. Packer identification method based on byte sequences. In *Concurrency Computation.* DOI: https://doi.org/10.1002/cpe.5082
[96] Erik Bergenholtz, Emiliano Casalicchio, Dragos Ilie, and Andrew Moss. 2020. Detection of metamorphic malware packers using multilayered LSTM networks. In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* DOI: https://doi.org/10.1007/978-3-030-61078-4_3
[97] Moustafa Saleh, E. Paul Ratazzi, and Shouhuai Xu. 2017. A control flow graph-based signature for packer identification. In *Proceedings - IEEE Military Communications Conference MILCOM.* DOI: https://doi.org/10.1109/MILCOM.2017.8170793
[98] Daniel Gibert, Carles Mateu, Jordi Planes, and Ramon Vicens. 2019. Using convolutional neural networks for classification of malware represented as images. *J. Comput. Virol. Hacking Tech.* (2019). DOI: https://doi.org/10.1007/s11416-018-0323-0
[99] Fabrizio Biondi, Michael A. Enescu, Thomas Given-Wilson, Axel Legay, Lamine Noureddine, and Vivek Verma. 2019. Effective, efficient, and robust packing detection and classification. *Comput. Secur.* (2019). DOI: https://doi.org/10.1016/j.cose.2019.05.007
[100] Binlin Cheng and Pengwei Li. 2018. BareunPack: Generic unpacking on the bare-metal operating system. *IEICE Trans. Inf. Syst.* (2018). DOI: https://doi.org/10.1587/transinf.2017EDP7424
[101] Zhigang Zhang, Chaowen Chang, Peisheng Han, and Hongtao Zhang. 2020. Packed malware variants detection using deep belief networks. *MATEC Web Conf.* (2020). DOI: https://doi.org/10.1051/matecconf/202030902002

[102] Yakang Hua, Yuanzheng Du, and Dongzhi He. 2020. Classifying packed malware represented as control flow graphs using deep graph convolutional neural network. In *Proceedings - 2020 International Conference on Computer Engineering and Application ICCEA'2020*. DOI: https://doi.org/10.1109/ICCEA50009.2020.00062
[103] Hiromu Yakura, Shinnosuke Shinozaki, Reon Nishimura, Yoshihiro Oyama, and Jun Sakuma. 2018. Malware analysis of imaged binary samples by convolutional neural network with attention mechanism. In *CODASPY 2018 - Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*. DOI: https://doi.org/10.1145/3176258.3176335
[104] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*.
[105] Obfuscated Files or Information: Software Packing | MITRE. Retrieved February 21, 2021 from https://attack.mitre.org/techniques/T1027/002/.
[106] The WildList Organization International. Retrieved February 25, 2021 from http://www.wildlist.org/.
[107] Five ways Android malware is becoming more resilient | Broadcom. Retrieved February 21, 2021 from https://www.symantec.com/connect/blogs/five-ways-android-malware-becoming-more-resilient.
[108] Executable compression - Wikipedia. Retrieved February 21, 2021 from https://en.wikipedia.org/wiki/Executable_compression.
[109] ImpREC - aldeid. Retrieved February 25, 2021 from https://www.aldeid.com/wiki/ImpREC.
[110] LordPE - aldeid. Retrieved February 25, 2021 from https://www.aldeid.com/wiki/LordPE.
[111] Cuckoo Sandbox - Automated Malware Analysis. Retrieved February 21, 2021 from https://cuckoosandbox.org/.
[112] The Sandbox | Understanding CyberForensics. Retrieved February 25, 2021 from https://cwsandbox.org/.
[113] Automated Malware Analysis Tool | Falcon Sandbox | CrowdStrike. Retrieved February 21, 2021 from https://www.crowdstrike.com/endpoint-security-products/falcon-sandbox-malware-analysis/.
[114] Free Automated Malware Analysis Service - powered by Falcon Sandbox. Retrieved February 21, 2021 from https://www.hybrid-analysis.com/.
[115] unicorn/sample_arm.c at master · unicorn-engine/unicorn. Retrieved February 21, 2021 from https://github.com/unicorn-engine/unicorn/blob/master/samples/sample_arm.c.
[116] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. 2020. When malware is packin' heat; Limits of machine learning classifiers based on static analysis features. DOI: https://doi.org/10.14722/ndss.2020.24310
[117] W. Yan, Z. Zhang, and N. Ansari. 2008. Revealing packed malware. In *IEEE Security & Privacy* 6, 5 (2008), 65–69, DOI: 10.1109/MSP.2008.126
[118] Debra A. Lelewer and Daniel S. Hirschberg. 1987. Data compression. *ACM Comput. Surv.* 19, 3 (1987), 261–296. DOI: https://doi.org/10.1145/45072.45074
[119] David A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.
[120] Threat Actors Use Delphi Packer to Shield Binaries From Malware Classification. Retrieved November 11, 2021 from https://securityintelligence.com/news/threat-actors-use-delphi-packer-to-shield-binaries-from-malware-classification/.
[121] Nir Nissim et al. 2019. Sec-lib: Protecting scholarly digital libraries from infected papers using active machine learning framework. *IEEE Access* 7 (2019), 110050–110073.
[122] Aviad Cohen, Nir Nissim, and Yuval Elovici. 2020. MalJPEG: Machine learning based solution for the detection of malicious JPEG images. *IEEE Access* 8 (2020), 19997–20011.
[123] Nir Nissim et al. 2014. ALPD: Active learning framework for enhancing the detection of malicious pdf files. *2014 IEEE Joint Intelligence and Security Informatics Conference*. IEEE, 2014.
[124] E. M. Rudd, R. Harang, and J. Saxe. 2018. MEADE: Towards a malicious email attachment detection engine. *2018 IEEE Int. Symp. Technol. Homel. Secur. HST'2018*. 1–7. DOI: 10.1109/THS.2018.8574202
[125] S. Shukla, G. Kolhe, S. M. Pd, and S. Rafatirad. 2019. RNN-Based classifier to detect stealthy malware using localized features and complex symbolic sequence. *Proc. - 18th IEEE Int. Conf. Mach. Learn. Appl. ICMLA'2019*. 406–409. DOI: 10.1109/ICMLA.2019.00076
[126] M. Li, Y. Liu, M. Yu, G. Li, Y. Wang, and C. Liu. 2017. FEPDF: A robust feature extractor for malicious PDF detection. *2017 IEEE Trustcom/BigDataSE/ICESS*.
[127] GitHub - NtQuery/Scylla: Imports Reconstructor. Retrieved March 7, 2022 from https://github.com/NtQuery/Scylla.
[128] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, and Davide Balzarotti. Prevalence and impact of low-entropy packing schemes in the malware ecosystem. DOI: https://doi.org/10.14722/ndss.2020.24297

Received 22 March 2021; revised 12 March 2022; accepted 6 April 2022