Int. J. Inf. Secur. (2017) 16:227–248
DOI 10.1007/s10207-016-0330-4

REGULAR CONTRIBUTION

Entropy analysis to classify unknown packing algorithms for malware detection

Munkhbayar Bat-Erdene[1] · **Hyundo Park**[1] · **Hongzhe Li**[1] · **Heejo Lee**[1] · **Mahn-Soo Choi**[2]

Published online: 4 May 2016

Abstract The proportion of packed malware has been growing rapidly and now comprises more than 80 % of all existing malware. In this paper, we propose a method for classifying the packing algorithms of given unknown packed executables, regardless of whether they are malware or benign programs. First, we scale the entropy values of a given executable and convert the entropy values of a particular location of memory into symbolic representations. Our proposed method uses symbolic aggregate approximation (SAX), which is known to be effective for large data conversions. Second, we classify the distribution of symbols using supervised learning classification methods, i.e., naive Bayes and support vector machines for detecting packing algorithms. The results of our experiments involving a collection of 324 packed benign programs and 326 packed malware programs with 19 packing algorithms demonstrate that our method can identify packing algorithms of given executables with a high accuracy of 95.35 %, a recall of 95.83 %, and a precision of 94.13 %. We propose four similarity measurements for detecting packing algorithms based on SAX representations of the entropy values and an incremental aggregate analysis. Among these four metrics, the fidelity similarity measurement demonstrates the best matching result, i.e., a rate of accuracy ranging from 95.0 to 99.9 %, which is from 2 to 13 higher than that of the other three metrics. Our study confirms that packing algorithms can be identified through an entropy analysis based on a measure of the uncertainty of the running processes and without prior knowledge of the executables.

Keywords Entropy analysis · Original entry point (OEP) · Symbolic aggregate approximation (SAX) · Piecewise aggregate approximation (PAA)

1 Introduction

1.1 Background on packed malware

Malicious software, also known as malware (e.g., viruses, worms, and Trojan horses), not only violates the security and privacy of computer users, but also incurs a considerable amount of financial loss. Malware has developed from a tool used primarily in data theft, botnets (both of which are still very common), and fake antivirus scams, to one used in ransomware attacks, as revealed by Symantec's 2014 "Internet Security Threat Report" [1]. According to Choi et al. [2], attackers continually make their malware harder to detect and analyze. Although anti-malware and other malware analysis and removal tools can mitigate this situation to some extent, the evolution of malware, such as in the booming growth of obfuscation techniques including polymorphic and metamorphic malware, is making the fight against its use more difficult.

One popular and widely used obfuscation technique is packing. Through the use of different packing algorithms, attackers possess a great number of malware options, including hiding its original behavior used. This makes it harder for security software engineers to detect and analyze malware using common anti-malware software. A packer is basically a program that produces a number of data blocks forming a compressed or encrypted version of the original executable,

✉ Heejo Lee
heejo@korea.ac.kr

[1] Department of Computer Science and Engineering, Korea University, 145 Anam-ro Seongbuk-gu, Seoul 02841, South Korea

[2] Department of Physics, Korea University, 145 Anam-ro Seongbuk-gu, Seoul 02841, South Korea

as described by Yan et al. [3]. Some packers use more sophisticated techniques to evade detection. More than 80 % of malware is now packed, as demonstrated by Yan et al. [3] and Lyda et al. [4]. Popular packers include Aspack, FSG, MPRESS, UPXN, NsPack, and Themida, many of which are readily available on the Internet. Because packers are making executables harder to analyze, the identification and classification of packing techniques are becoming critical to uncover the malicious intentions of the packed malicious program. Identification and classification not only help in the detection and analysis of malware, but also allow the functions and characteristics of benign packed programs to be analyzed. In addition, identifying packers enables security software engineers to efficiently analyze a packed executable file and retrieve the original payload for further malware analysis. Generally, packers can be categorized into mainly three types.

1. Benign packer (used for benign programs only)
2. General packer (used for benign and malicious programs)
3. Custom packer (made by malware authors and used for malware only)

If a program is packed with a custom packer, it indicates the program. The malware can be detected at an earlier stage without further analysis of the contained payload by using our classification method.

1.2 Motivation

Malware attempts to evade anti-malware software using various obfuscation techniques. Packing is one technique that is widely used to bypass anti-malware software. Malware, mostly in the form of packed executables, is a growing problem in modern computer systems and thus represents a major challenge. According to Lyda et al. [4] and Symantec Research Laboratories [5], more than 80 % of malware uses packing algorithms to circumvent anti-malware software.

Owing to the prevalence of packing tools on the Internet, packing can be easily utilized by attackers who do not possess in-depth programming knowledge. In addition, the increasing number of variants and unknown packers present a great threat to defenders. Few studies on detecting packing algorithms have been conducted during the last two decades. If a malware executable is packed, its infectious properties become nearly impossible to detect through signature matching. Another important issue demanding consideration is the fact that malware writers often use sophisticated code obfuscation techniques to evade signatures. In fact, simply repacking existing malware in most cases is sufficient to evade signature-based techniques. According to Zuber et al. [6,7], Cesare et al. [8], and Liu et al. [9], 50 % of new malware is a repacked version of existing known malware.

Many different packers are used within a single malware family to avoid detection systems. Although packing algorithms are widely used for packing malware, they can also be used to protect legitimate software from reverse engineering. At runtime, a packed executable contains a section of uncertain data. From the entropy measurements of a running process, we can determine whether a given executable is packed. Many packing algorithms are being developed every year; however, no complete databases exist to detect all packing algorithms. Detecting packing algorithms is necessary for recognizing hidden malware and preventing such malware from evading anti-malware software. Anti-malware software must cope with old and new packers daily.

An automatic system that detects packing algorithms is indispensable to ensure effective security. However, the immense variety of existing packing algorithms can cause a packer detection process to become time-consuming. Moreover, current automatic systems mainly concentrate on detecting malware and not on the methods used in producing it. Today, packing algorithms are used extensively in malware development to help malware become undetectable. We devised a new technique for the identification and classification of packing algorithms by creating simple patterns of packers. In addition, we studied the use of entropy-based detection, which is dynamic in nature and more reliable than signature-based techniques.

We propose a method for identifying packing algorithms of given packed executables that are either known or unknown, and classify them by employing four similarity measurements and popular classification techniques such as naive Bayes (NB) and a support vector machine (SVM). In an experiment, we detected the use of a different packers and distinguished unknown packers from 19 existing packers. The results of this experiment revealed the high accuracy of the proposed method in detecting packing algorithms.

1.3 Main contributions

The main contributions of our work are threefold and are.

1. We developed a holistic method that identifies packing algorithms immediately before the payload of a packed malware begins to be executed. Because malware programmers use a variety of encryption and compressions techniques, identifying infections within files prior to unpacking is nearly impossible. Therefore, we can draw entropy patterns first by unpacking each packed executable. Then, to detect and classify the packing algorithms of each packed executable, we extract unique symbolic patterns through the use of entropy variables.

These patterns enable us to either classify each packing algorithm based on existing classifications, or assign it a new one.

2. Without the use of ready-made unpacking tools, we unpack known and unknown packed executables using an entropy analysis by finding an original entry point (OEP). If an OEP is found, we can detect any packing algorithm in a packed executable even if obfuscation techniques have been applied. Therefore, our proposed method is robust in unpacking any kind of packed executable and identifying packing algorithms even when they have been altered.
3. To the best our knowledge, our proposed method is the first in identifying packing algorithms of unknown packed executables. Therefore, in this paper, we present a detection method that can be employed in identifying any kind of packing algorithm. The proposed method is dynamic and does not depend on the known signatures of the packing algorithms. We assert that our proposed method can extract the entropy patterns of packed executables, thereby increasing the efficiency of detection.

The remainder of this paper is organized as follows. In Sect. 2, we describe previous studies related to packer classification and pattern-recognition techniques. Section 3 briefly discusses how packing is operated and examines the challenges that software security engineers face when coping with packers. Section 4 describes the structure of our proposed method, the classification of symbolic aggregate approximation, and classifier and similarity measurements. Section 5 examines the effectiveness of classifying packing algorithms and describes the results of experiments conducted using the proposed method with packed executables and different classification techniques. Finally, Sect. 6 provides some concluding remarks regarding this research.

2 Related works

During the last decade, researchers have adopted a variety of solutions to the control of malware. A manual analysis of packers is an early solution to the exposure of malware and is mainly used to reduce the false negatives of a signature-based technique. However, it is infeasible in terms of efficiency. Therefore, we conducted a study on previous related works in the following categories: signatures, machine learning, pattern recognition, and control-flow graphs, all of which can be employed to detect packing algorithms and packed malware.

Signature-based detection [10,11] involves a search for special patterns of known malware in an executable code. The signature patterns can possess a simple binary sequence, a binary sequence with mask bytes, or a specially designed checksum [10,12]. Signature-based detection [13] is one of the best ways to identify known malware, but it is ineffective in detecting new malware type and their variants, particularly when malware mutates, thereby making signature-based detection difficult to achieve. Such metamorphisms were previously witnessed by Marinescu et al. [14]. Malware authors have evaded signature-based detection using custom packers to pack the original malware. Signature-based packer detection has a weakness regarding the measurement distance. For instance, it cannot measure some packed executables because certain packers use multilayer encryption and compression on the packed section, as observed by Guo et al. [5] and Jacob et al. [15].

An effective packer classification framework that applies pattern-recognition techniques to automatically extract the randomness profiles of different packers was proposed by Li et al. [13]. As a substitute for signature-matching approaches, the researchers presented a packer classification approach by analyzing the performance of various statistical classifiers. In addition, they tested various statistical classification algorithms, including k-nearest neighbor (KNN), best-first decision tree, sequential minimal optimization, and NB. All four classifiers were found to be extremely effective, with three of the four achieving an average true-positive rate of approximately 99 %; however, NB was the least effective with a true-positive rate of approximately 94 %. The k-nearest neighbor classifier, where k = 1, achieved the best overall performance, with a true-positive rate of 99.6 % and a false-positive rate of 0.1 %. The system also reveals that the low randomness profile of a packed file normally produced by the PE header and an unpacking stub contains important packer information, and is extremely useful in distinguishing among families of packers. Li et al. [13] did not propose a technique for detecting unknown packers, and thus, their method is not useful in detecting non-signature-based packers and packed files. In contrast, our classification method extracts the entropy pattern of packing algorithms without any prior knowledge of the packed executables.

A significant amount of research has been conducted on developing automatic malware classification systems using data mining and machine-learning approaches [16–22]. All classification approaches examined from the literature can be categorized into two main types: those employing features drawn from an unpacked static version of the executable, and those employing dynamic features of the packed executable. Several approaches for detecting unknown malware based on its binary code have been discussed.

Kolter et al. [19] presented a vector of n-grams to represent malicious and benign files, and a comprehensive evaluation of classifiers instance-based (IBk), term frequency–inverse document frequency (TFIDF), NB, SVM, decision-tree, boosted NB, boosted SVM, and boosted decision-tree classifiers. The researchers showed that the results of their n-gram study were better than those presented by Schultz et al. [20] and that the boosted decision tree classifier outperformed the others. A static analysis consists of examining program codes, without running them, to determine the properties related to the dynamic execution of these programs. A behavior-based approach monitors the execution of a program to detect malicious behavior. A common way to represent a program behavior is the use of a system call sequence, which is a representation widely used in anomaly detection systems. Other methods based on a dynamic analysis for malicious code classification include those proposed by Bayer et al. [23] and Christodorescu et al. [24]. However their methods are dedicated only to detect known malware. Moreover, Kolbitsch et al. [25] proposed to classify an instance of a given malicious code to a predefined class, whereas the method cannot be used in detection of new packed or non-packed malicious codes in the real time.

A method for the automatic specification of malware behavior was proposed by Christodorescu et al. [24], who introduced a malware specification concept known as malicious specification (Malspec). Malspec contrasts the execution behaviors of known malware and a given set of benign programs. The researchers showed that a Malspec can be converted into templates or signatures that are used by malware detectors to detect variations of certain malware. Kolbitsch et al. [25] revealed that a Malspec does not encode data flow dependencies between system call parameters and that using a Malspec for detection without verifying such dependencies may lead to many false alarms. The authors proposed an approach that builds a behavioral graph for analyzing malicious programs. This graph employs nodes, which are system calls, and edges, which represent the data dependency between system calls. Researchers then extracted the program slices responsible for such dependencies. For detection, they matched the extracted program slices with the runtime behavior of an unknown program. Both Malspec and behavioral graph methods are suitable for detecting variations of existing malware, but not for detecting new or packed malware.

Malware developers have generated a wide range of approaches to minimize the susceptibility of their products to heuristic detection [26]. Suspicious OEP, section characteristics, application programming interface-API calling, multiple PE headers, code graphs [27], control-flow graphs [18], pattern recognition, and other features can be synthetically used to detect malware [16,17]).

A pattern-recognition technique [28–30] for the fast detection of packed executables was proposed by Roberto et al. [16], who applied various pattern-recognition techniques to classify executables into two categories, i.e., packed and non-packed. These techniques employ publicly available unpacking tools and signature-based anti-malware systems to distinguish specific kinds of malware and benign executables. If an executable is classified as packed, it is sent to a universal unpacker for hidden code extraction, and the hidden code is then sent to an anti-malware software.

However, if the executable is classified as non-packed, it is sent directly to an anti-malware scanner. This system achieves a high accuracy of greater than 95 % by using classifiers such as NB, J48 decision tree, bagged-J48, k-nearest neighbors, and a multilayer perceptron (MLP). The best results were obtained when using the MLP classifier, which showed an accuracy of 98.91 % for the test dataset. However, this approach did not classify different packers into different families. The weakness of this technique [16] is that unknown packed files cannot be unpacked because the researchers used universal unpacking tools as unpacking mechanisms. Moreover, the authors used a static analysis in their approach and did not propose a technique for packer identification. In contrast, our classification system extracts a specific type of packing algorithm from any packed PE file.

A malware classification method that constructs the control-flow graph-based signatures was designed by Silvio et al. [18]. In their classification method, the similarity between structured graphs can be quickly determined and malware can be effectively classified using string-edit distances.

Similar to what was employed in our research, they also detected the OEP of packed executables through the use of an entropy analysis when unpacking packed malware. However, their classification method is based solely on a similarity distance and predefined thresholds, which we consider to be somewhat vague and therefore insufficient for effective classification. Moreover, although the aforementioned research shows relatively favorable results in terms of identifying and classifying malware and its variants, it does not classify packing algorithms, which is essential in the analysis of packed malware.

A mechanism to determine OEP through the use of an entropy analysis was proposed by Jeong et al. [31]. However, they did not detect or classify the packing algorithms. As previously mentioned, several approaches for detecting packed malware and packing algorithms exist. However, these existing approaches have been proven to be ineffective in detecting known and unknown packing algorithms.

Martignoni et al. [32] proposed a new technique, OmniUnpack, which is generic and largely effective, but neither safe nor portable. OmniUnpack monitors the program execution and tracks written, as well as written-then-executed memory pages. When the program makes a potentially damaging system call, OmniUnpack invokes a malware detector on the written memory pages. If the detection result is negative, execution is resumed. A study by Renovo et al. [33] used a virtual machine, and they showed that many packed executables contain several layers. From an analytical perspective, the decompression module of a packed executable is only one part of the executable. No explicit signatures exist that mark a boundary between the decompressing and decompressed instructions. As a result, previous research has only focused on recording immediate instructions. However, our approach can locate the boundary, or the OEP, through the use of an entropy analysis.

3 How packing works

In this section, we provide academic insight into a mechanism that classifies unknown packing algorithms. We then briefly describe the operations involved in both the packing and unpacking process of packed executables when an entropy analysis is employed.

3.1 Packer's stub

A packer is a program that converts an executable into a compressed executable using an any available algorithm.

Packers are software programs that compress and encrypt other executable files in a disk and restore the original executable images when the packed files are loaded into memory. Packers do exactly what their name suggests, "pack" (i.e., compress) a program in the same way that a compressor, such as Pkzip, compresses files. Packers then use decryption or loading stubs to "unpack" the program before resuming normal execution at the program's OEP.

Packing is thus a conversion operation of a packing algorithm, as shown in Fig. 1. Currently, many types of malware use packing algorithms. Therefore, the goal of this research is the systematic detection of packing algorithms. The packed files described in this paper are in portable executable (PE) format (Guo et al. [5] and Pietrek et al. [34]). Packing algorithms have been devised to allow authors of malicious software to extend their expected lifetime.

Many authors of valid commercial software have used packers to make it difficult and costly to reverse-engineer their programs. Packers have thus evolved into sophisticated operations involving complex routines designed to encrypt the executables they want to protect. A packer takes an original program and compresses it. The compressed executable can then be moved to the data section of the newly created file.

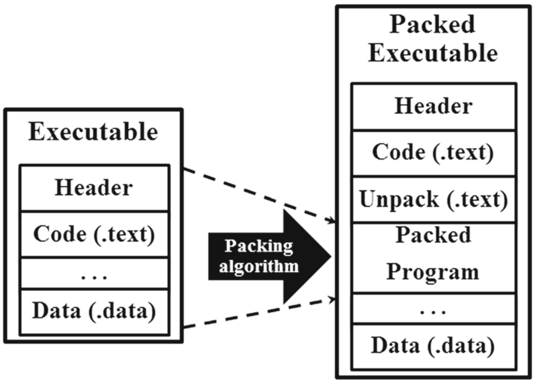

Fig. 1 Packing process of a PE file

The executable portion of the program is essentially a simple routine designed to decompress an original file into memory and to resume the execution at the OEP of the uncompressed program. Packing utilizes compression techniques to change the size of an executable. Because compression is applied, the executables are transformed after being packed. Malware takes advantage of this transformation for the purpose of obfuscation.

In addition, packing is easily abused by writers of malicious code, particulary because many packers such as UPXN, Aspack, NsPack, nPack, Molebox, and MPRESS are freely available on the Internet. In addition, malware writers can create their own packers at a low cost. Packers generally create an executable of a different size (either smaller or bigger) than the original file and change the signature of the file as well as any hash that can be used to conduct simple pattern matching. As the packing algorithms evolve, creating anti-malware software becomes more costly. For instance, Fig. 2 illustrates how file calc.exe, which is packed with UPX, operates during its execution. The packed executable contains different section structures from the original program, including a section called UPX0, which is used to write unpacked code and data. It is initialized as 0. Although well-known compression algorithms can be applied to pack an executable, malware writers can also create a private packer by developing and applying a new compression algorithm, which makes it difficult for analyzers to determine how a packed executable is compressed. Another element of UPX1 includes the unpacking routine. When it finishes unpacking, the control flow jumps to one of the unpacked instructions. Analyzing packing algorithms becomes increasingly difficult, because instead of employing only a single packing technique, different compression algorithms can be used multiple times.

Packers sometimes have anti-debugging functions in their unpacking process, and their unpacking instructions can be obfuscated. It is very difficult to manually unpack such executables. If the packing algorithm were classified based on the packer identification, the classification will be helpful for analysts because they provide useful information about anti-debugging and obfuscation methods.

3.2 Packed executables

To fully understand our approach, packed executables and the method used to build and execute them must be described. Describing the fundamental characteristics of a packer is sufficient for an understanding of our mechanism, although

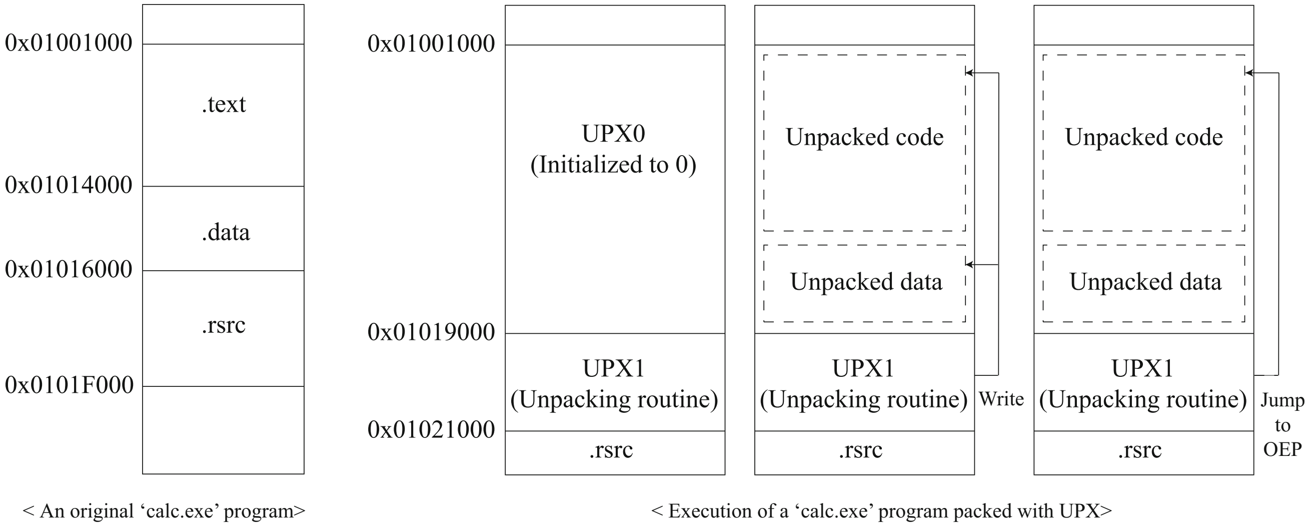

Fig. 2 Sequence running an executable packed using a UPX packing algorithm

each packer can have its own algorithm. A packed file comprises two essential components: data blocks that form the compressed and/or encrypted original executable file, and an unpacking stub that can dynamically recover the original executable file on the fly.

Packing indicates the compression of the execution file, which is unpacked when the executable file is executed. A packed executable is built with two main components during a two-phase packing process. First, the original executable is compressed and stored as data in a packed executable. Second, a decompression module is added to the packed executable and used to restore the original executable. After the packing procedure is completed, a packed executable is both transformed and packed, as demonstrated in Fig. 3.

3.3 Unpacking mechanism

When the packed file is run, the unpacking stub is first executed to unpack the code section whose entropy of memory is measured and then transfers the control to the original sections that are restored in memory. When the unpacking process has finished, the execution of the original file, which remains mostly unchanged, starts from its OEP with no penalties in the runtime performance. Unpacking employs the reverse process of a packing algorithm. Figure 4 shows that unpacking is a restoration process, because an original executable is restored in memory when the unpacking is completed. Decompression is first conducted and the execution flow then progresses or “jumps” to the first instruction of the unpacked code. After restoring the original executable, the execution flow jumps from the end point of the decompression module to the entry point of the original executable.

A jump (JMP) instruction is one of the commonly used ways to change the execution flow. We let the process run until one of the instructions, such as JMP, JCC, CALL, or RET, is encountered. If this happens, the execution is paused, and an entropy analysis of that instant of the process is conducted. When the unpacking is completed, these kinds of instructions should be used for changing the execution flow from the end of the decompression module to the beginning of the original unpacked code, i.e., the so-called OEP. Another issue regarding execution-flow-changing instructions is the number of instructions in a program.

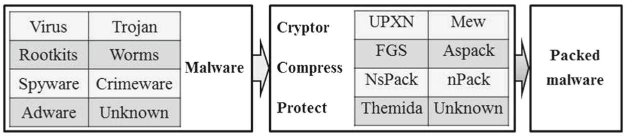

Fig. 3 Sequence of the packing process

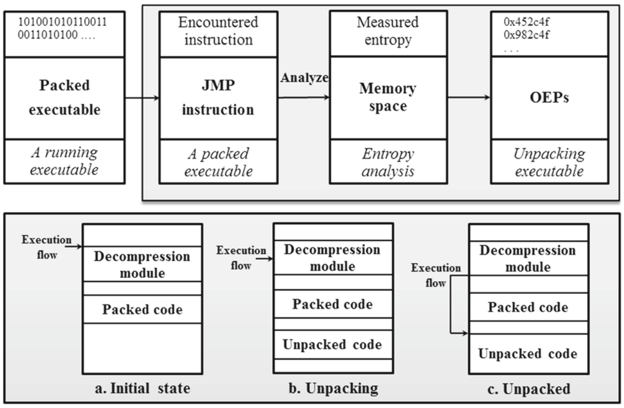

Fig. 4 Sequence of the unpacking process

The remainder of this section details the unpacking mechanism. First, the executable is executed and continues to

Algorithm 1: Finding the OEP during unpacking

```
Input: The input is a packed executable. The output is an entropy
    sequence. The packed executable, an instruction pointer, and
    entropy of unpacked code are represented by P, IP and E,
    respectively. We assume that an executable is categorized as
    either packed or native.
Output: Locate OEP of P.
// Initialization runs the executable
Find an entry point and the first section of P.
Set a break point to the entry point.
Set R to the range of the first section.
// Start analysis
while the PROCESS is not terminated do
  IP ← a current instruction pointer
  // Measure entropy in only this condition
  if IP is for a JMP instruction then
    Measure entropy of R.
  else
    Continue this loop.
  end if
  // Check if unpacking is complete
  if $E_{min} \leq$ Measured entropy $\leq E_{max}$ then
    // Check if it jumps onto the unpacked code
    if Jump into R from outside of R is true then
      OEP ← The next instruction address
      Break this loop.
    else
      Continue this loop.
    end if
    Continue this loop.
  end if
end while
```

run until a JMP instruction is encountered. When a JMP instruction is encountered, the execution is paused and an entropy analysis for that instant of the process is conducted. JMPs are the most important instructions in our approach because the OEP is always followed by a JMP instruction. When the unpacking is completed, a JMP instruction is used to alter the execution flow from the end of the decompression module to the beginning of the unpacked original code, or the OEP. In our study, whenever an instruction pointer encounters a JMP instruction, this instruction is considered an entropy analysis point. However, JMP instructions in APIs are ignored because the address of the OEP is determined by the packer. Therefore, even though the unpacking mechanism can feasibly check the APIs, JMP instructions from APIs are ignored.

In addition, several JMP instructions may be required in a program, because certain instructions are repeated for iterations (e.g., for() or while() in C language). An unpacking module, for instance, consists of several iterations, which increases the analysis time. These iterations are not helpful in locating the OEP and merely delay the analysis. To solve this problem, we cached a number of addresses of JMP instructions that are encountered during an analysis, and when a repeat cached address is reached, the entropy analysis at that point is skipped.

4 Entropy analysis for detecting packing algorithms

We determine that a considerable overlap exists between non-packed and packed executables. This indicates that all sections of packed executable can be measured for the purpose of analyzing the unpacking process. Based on these observations, we only measure changes in all sections of packed executables. Additionally, some data in the first section should be eliminated prior to the analysis if it is found to contain garbage values, which can affect the entropy analysis and cause errors. Therefore, data to be measured are instructions in the first sections of packed executables after the garbage values have been removed. Using a formula that will be explained in the entropy analysis section, the entropy of the filtered data can be measured. We use a natural logarithm, and the unit of data is a byte. During the unpacking process, instructions are unpacked in all sections. The value of the measured entropy during the analysis indicates the data state (e.g., packed, unpacked, or being unpacked) in all sections at that instant.

Our experimental results show that the entropy values change, while a packed executable is being unpacked. Therefore, we can determine whether an unpacking has been completed through the use of an entropy analysis. Because we discovered that most benign executables contain instructions in the first section, when the unpacking process is completed, the execution flow should change to the first instruction of the original executable to continue its functionalities. The measured entropy of an unpacked code is detailed in the next section.

Our approach involves the unpacking of packed executables and identifying their OEPs. The algorithm used in this approach is discribed as a pseudo-code in Algorithm 1.

The main purpose of detecting packing algorithms through an entropy analysis is to achieve a high classification accuracy of the packing algorithms for an effective performance in practical anti-malware software. First, we assess and classify packed executable files based on such characteristics as their entropy and complexity, rather than on their signatures. Because known and unknown packed executables are compressed or encrypted, their high complexity simplifies the process of assessing whether or not they are packed. We then detect and classify packing algorithms through our proposed method of entropy analysis. We create classes based on a similarity of entropy patterns and define the packing algorithms of each class. Ultimately, our method detects packing algorithms efficiently and makes the analysis of packed malware much easier to achieve.

4.1 Steps of packing algorithm detection

The main characteristic of our proposed method is to measure the entropy values while unpacking packed executables. Detecting packing algorithms directly through their entropy pattern is problematic. Because of the extensive amount of data and the number of errors incurred, the process is both time-consuming and difficult to analyze.

Therefore, as shown in Fig. 5, we extracted simple patterns from entropy patterns through a symbolic representation. Figure 5 shows what happens during an unpacking process in memory space in terms of entropy. The order of execution-flow-changing instructions is the instant of each entropy analysis.

We classified packing algorithms in four class based on their graphically visualized patterns.

1. *Increasing class*
   Packing algorithms of the increasing class initialize memory space, where unpacked code will be written, as zeros; it starts with zero entropy values. As packed code is unpacked, written code causes the increase. Finally, it stops changing when unpacking is complete.
   Reasons of such increase in patterns are, first, uniqueness of each packing algorithm and, second, a usage of formula 1 in summation of entropy values. For instance, packing algorithms such as FSG, nSpack, Alternate_EXE, UPXN, and UPX-iT are from a same class, the Increasing, even though they are different packing algorithms, as illustrated in Fig. 5 (1).
2. *Decreasing class*
   On the other side, packing algorithms of the decreasing class does not initialize memory space before unpacking packed executables. Therefore, entropy values start decreasing from higher value down to the end point of unpacking. Reasons of such decrease in patterns are, first, uniqueness of packing algorithms and, second, a usage of formula 1 in summation of entropy values.
   The (h) pattern of the class seems to be constant; however, with closer look it is evident that the entropy value is slightly deteriorating as shown in Fig. 5 (2). Although entropy patterns of packing algorithms in the decreasing class are similar, they are different products.
3. *Combination class*
   The combination class is divided in two classes: the increasing-to-constant and the decreasing-to-constant patterns. Packing algorithms of the combination class do not initialize memory before unpacking packed executables. Their entropy values start from higher value and either increase or decrease till stabilizing at certain value and continue to process at that level.
   Combination class encloses packing algorithms patterns of which show that their entropy value initially start at higher level and, then, sharply fall down till stabilizing at a certain value. Packing algorithms such as Themida and VMportect are classified in the increasing-to-constant pattern, Aspack, and Molebox are classified in the decreasing-to-constant pattern as described in Fig. 5 (3).
4. *Constant class*
   Constant class encloses patterns of packing algorithms for benign packed executables. Entropy patterns of benign packed executables have constant values.

Per our observation, a packing algorithm in malware can be one of the three classes: the increasing class, the decreasing class, and combination class.

We then classified the simple patterns through NB and SVM classification algorithms. In general, we classified symbolic representations using supervised and unsupervised learning classification techniques. Figure 6 shows the three primary components of our method. The first part involves measuring the entropy patterns, the second part is converting them into a symbolic representation, and the final part is classifying the packing algorithm.

4.2 Measuring the entropy pattern and entropy analysis

Measuring the entropy pattern enables us to determine the entropy value of the packed executables during the unpacking process. Therefore, we first execute a given packed executable and allow the unpacking process to complete, as shown in Fig. 7. During the unpacking process, packed instructions are unpacked by a decompression module, and the measured entropy of the memory space continuously changes automatically. The final stage of the unpacking process can be detected by identifying when the entropy stops changing. The executable is executed and continues to run until a JMP instruction is encountered. An entropy analysis is conducted by measuring the specific memory space.

A mathematical tool for measuring the entropy of information is required to apply the concept of entropy measurements, and Shannon's formula was devised as follows:

$$H(x) = -\sum_{i=1}^{n} p(i) \cdot \log_b p(i) \tag{1}$$

where $H(x)$ is the measured entropy value and *p(i)* is the probability of the ith unit of information in the series of n symbols of event x. The base number of the logarithm (b) can be any real number greater than 1. However, a value of 2, 10, or e is generally used, including by Yeung et al. [35], Costa et al. [36–38], Nikulin et al. [39], Pincus et al. [40], [41], Richman et al. [42], and Lake et al. [43]. Using an entropy analysis, Lyda et al. [4] proposed that binary files with a

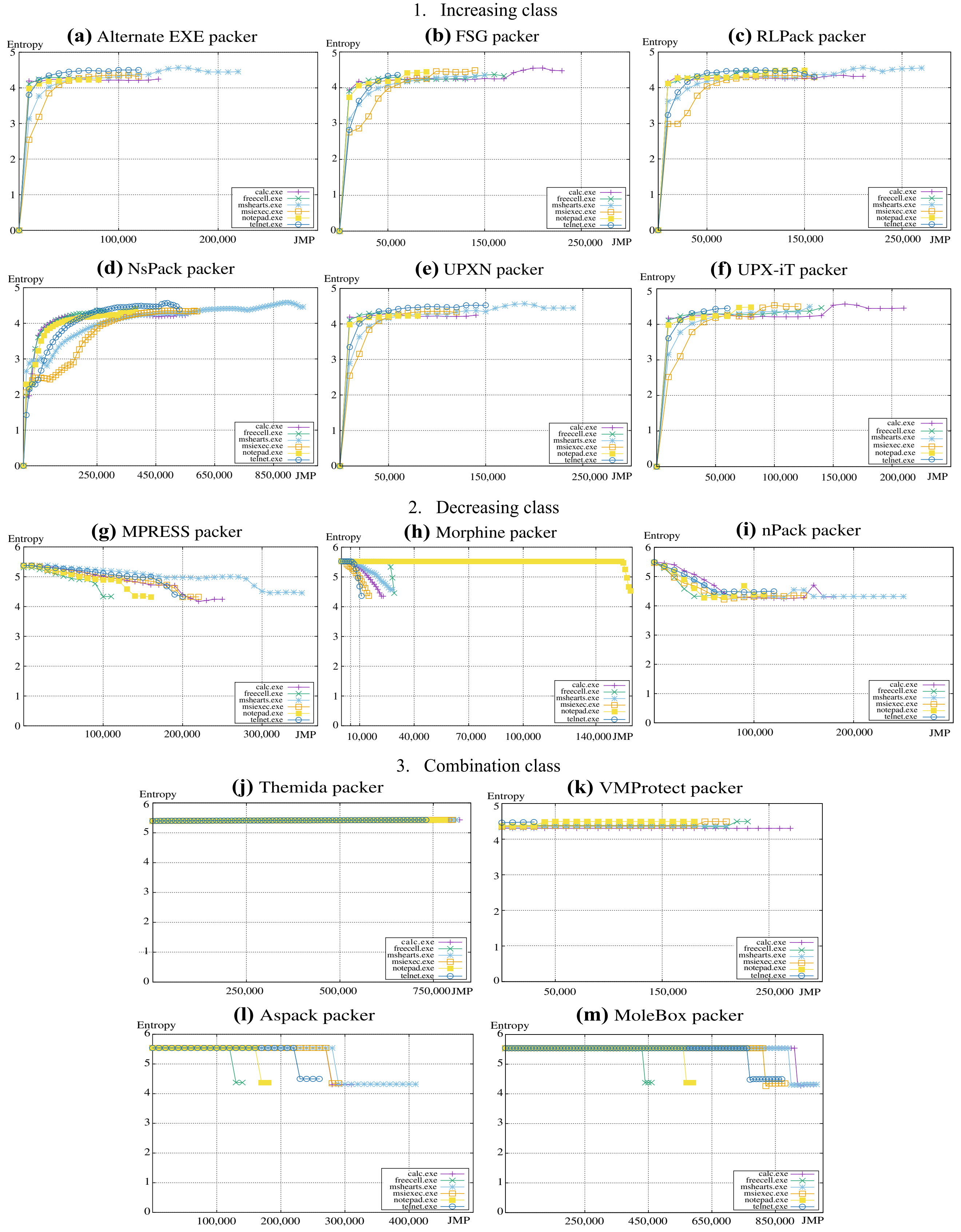

Fig. 5 Training entropy patterns of thirteen packing algorithms selected from 13 packing algorithms

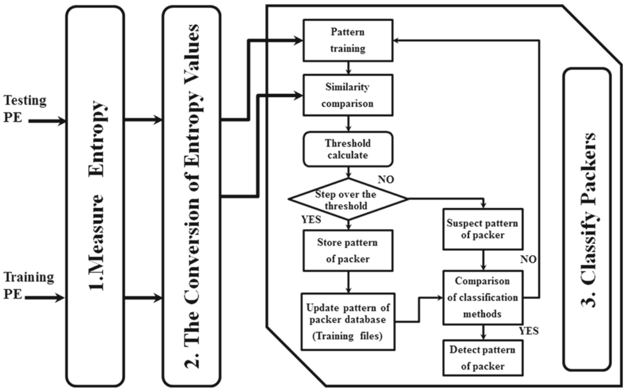

Fig. 6 The packing algorithm detection method for a given packed executable

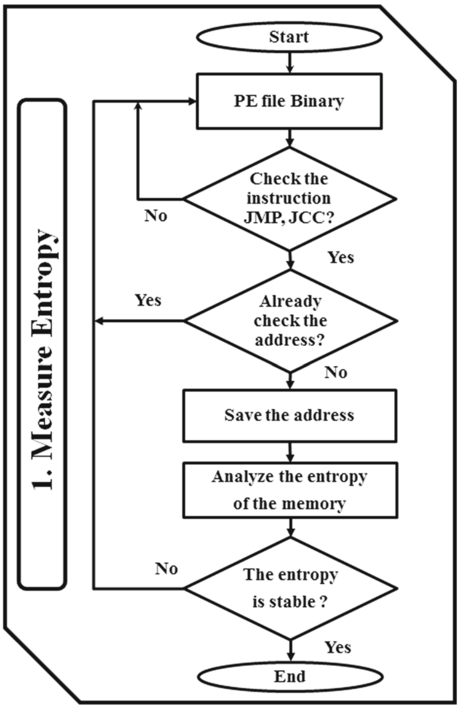

Fig. 7 Structure of an entropy analyzer used to locate the OEP

high entropy score tend to correlate with the presence of encryption or compression. Accordingly, binary files can be classified into four types: plain text files, benign executables, packed executables, and encrypted executables. Researchers have suggested that boundaries among the four categories exist and can be determined through the entropy. Based on such research, the present paper proposes an additional useful application of an entropy analysis. While a packed code is being unpacked, the measured entropy is changed and can be monitored by employing an entropy analysis.

An entropy analysis is an appropriate tool for unpacking because it can detect the end point of the unpacking by observing changes in entropy. A study by Lyda et al. [4] describes a static analysis because of its focus on binary files. In contrast, our approach is concerned with dynamic processes.

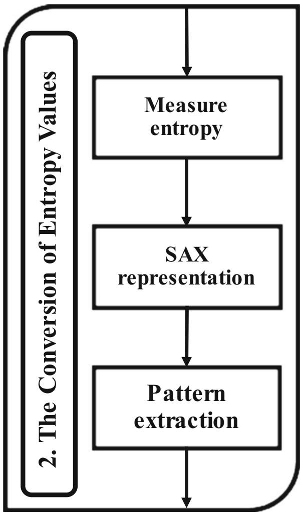

Fig. 8 Structure of the symbolic pattern-extraction method using SAX

4.3 Conversion into symbolic representation using SAX

A similarity search is a fundamental aspect of computer science and can be applied to many areas such as multimedia databases, bioinformatics, pattern recognition, text mining, computer vision, data mining, and machine learning.

Time-series data mining involves many tasks such as classification, clustering, similarity searches, motif discovery, and anomaly detection. A critical factor in the successful performance of these tasks is the use of representation methods that can represent a time series in an efficient and effective manner, as described by Chakrabarti et al. [44]. Of all the symbolic representation methods in the literature focusing on time-series data mining, the symbolic aggregate approximation (SAX) method stands out as one of the most powerful techniques, as revealed by Jessica et al. [45].

An orderless entropy pattern is then converted into a simple symbolic pattern. To extract a packing algorithm and represent its pattern through a symbolic representation, we use the entropy analysis shown in Fig. 8. A symbolic representation allows for a dimensionality reduction and indexes using a lower-bounding distance measure of the true distance. A dimensionality reduction occurs when data of extremely high dimensionality are converted into data of much lower dimensionality such that each of the lower dimensions conveys more information than before. Thus, the dimensionality reduction in a piecewise aggregate approximation (PAA) is automatically carried over to a symbolic representation, as shown by Chakrabarti et al. [44] and Yi et al. [46].

The lower-bound distance between two symbolic strings can be proven by simply pointing to existing proofs for a PAA

Algorithm 2: Conversion into symbolic representation

```
Input: $E$, $\overline{E}$, $\overline{E}_{norm}$, $\beta$ and $\phi(\beta)$.
// Extract symbolic unique pattern, which will be used in the detection of any packing algorithms.
Output: $G$ and $\overline{S}$.
$\overline{E} \leftarrow$ scale $(E)$
// Scale the entropy values for symbolic representation.
$\overline{E}_{norm} \leftarrow$ Normalize $(\overline{E})$
Loop $i = 0; i < \phi(\beta); i < i + 1$
If $\beta_{i-1} < \beta_i$ then
$\phi(\beta) \leftarrow$ Divide $(\overline{E}_{norm})$
// Divide normalized entropy values using number of symbols.
$G \leftarrow$ Convert $(\overline{E}_{norm})$
// Convert normalized entropy values into the sequence of symbols.
$\overline{S} \leftarrow$ Extract New Symbolic Pattern $(G)$
// Extract new unique symbolic pattern from entropy values.
End If
End Loop
```

representation itself. Researchers can take advantage of the generic time-series data mining model as well as a host of other algorithms, definitions, and data structures that are only defined for discrete data including hashing, Markov models, and suffix trees. SAX is one of the most competitive methods in the literature and utilizes a similarity measurement which is easy to compute because it is based on precomputed distances obtained from lookup tables.

We present an improved similarity measurement for using SAX. This measurement has the same advantages as the original similarity measurement used in SAX. SAX is based on the fact that a normalized time series has a high Gaussian distribution. Therefore, according to Jessica et al. [45], by determining the breakpoints that correspond to the size of the alphabet, one can obtain equally sized areas in a Gaussian curve. We used a Gaussian distribution for converting scaled entropy values into a symbolic representation. SAX is applied as follows.

- First, the time series are scaled and normalized.
- Second, the dimensionality of the time series is reduced using PAA by Keogh et al. [47].
- Third, the PAA representation of the time series is discretized, which is achieved by determining the number and location of the breakpoints.

The SAX method approximates time series X of length n into vector $\overline{X} = (\overline{x_1}, \overline{x_2}, \ldots, \overline{x_M})$ of any arbitrary length M ($M < n$, typically $M << n$), where each $\overline{x_i}$ is calculated through (2) below,

$$\overline{x_i} = \frac{1}{r}\left[\sum_{j=r(i-1)+1}^{ri} (x_j)\right], \tag{2}$$

where r is a ratio defined as

$$r = \frac{n}{M}. \tag{3}$$

Here, M is the length of the original time service (original entropy values), and n is the length of the string (i.e., the number of frames or symbols). Simply stated, to reduce the time series from n dimensions to M dimensions, the data are divided into M equally sized frames. SAX is the first symbolic representation of a time series having an approximate distance function that lower bounds the Euclidean distance. In Algorithm 2, we presented a process for converting entropy values into SAX.

An algorithm is required for converting the entropy values into a symbolic representation. We unpack the given packed executables, regardless of whether they are benign or malware programs.

Entropy values from unpacking a packed executable, the scaling values of the original entropy pattern, the normalized scaling entropy values, the breakpoints, the number of symbols, the sequence of symbols, and the SAX pattern are abbreviated as E, $\overline{E}$, $\overline{E}_{norm}$, β, $\phi(\beta)$, G, and $\overline{S}$, respectively. In SAX, the data are first transformed into a PAA representation, and the transformed PAA representation is then symbolized into a sequence of discrete strings, as indicated by Jessica et al. [45]. The breakpoint locations are determined using statistical lookup tables so that these breakpoints produce equally sized areas in the Gaussian curve. The interval between two successive breakpoints is assigned to a symbol of the alphabet, and each segment of the PAA that lies within that interval is discretized by this symbol. As a final step, a similarity measurement of SAX is taken.

Breakpoints (β) are a sorted list of numbers, where $\beta = \beta_1, \beta_2, \ldots, \beta_{a-1}$ such that $\beta_{i-1} < \beta_i$ divides the area in an N(0,1) Gaussian curve into equally sized areas. In addition, the size of the alphabet is an arbitrary integer a greater than 2 (e.g., the letters $= (a, b, c)$ when $a = 3$). According to Jessica et al. [45], these $breakpointsaredeterminedby$ locating them in a statistical table. The range of β is

$$\text{from} \quad \beta_i = 0 \quad \text{to} \quad \beta_{i+1} = \frac{1}{a}, \tag{4}$$

where the variables β_0 and β_a are defined as $-\infty$ and ∞, respectively.

Given that the normalized dataset (scaled entropy values) has a high Gaussian distribution, we can simply determine the breakpoints that will produce a, where the breakpoints are equally sized areas on the Gaussian curve. The breakpoints divide a Gaussian distribution into a number of equiprobable regions, as illustrated in Fig. 9. If we transform the original entropy values into PAA representations, $\overline{Q}$ and $\overline{S}$ using formula (2), we can then obtain a lower-bounding approximation using the Euclidean distance ($\mathscr{D}$) [47] between the

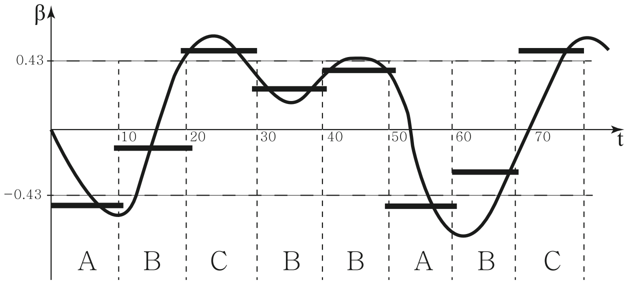

Fig. 9 The entropy pattern is discretized by first obtaining a PAA approximation and then by using predetermined breakpoints (β) to map the PAA coefficients into SAX symbols

original entropy values through the following formula for the function($\mathscr{D}$):

$$\mathscr{D}(\overline{Q}, \overline{S}) = \sqrt{\frac{n}{N}} \cdot \sqrt{\sum_{i=1}^{M} (\overline{Q}_i - \overline{S}_i)^2}, \tag{5}$$

where n, N, and M follow the definitions for formula (2). The $\mathscr{D}$ function measures the "as the crow flies" distance.

4.4 Classifier

Our proposed method includes two types of classification. The first type of classification includes commonly used similarity classification methods such as Cosine, Dice, Jaccard, and fidelity. Figure 10 illustrates our use of the similarity classification process. Figure 10 shows the three primary components of our similarity classification process. The first part involves measuring the original entropy values, the second part is scaling the entropy values, and the last part is similarity measurements. The second type of classification incorporates commonly used classification methods such as the NB and SVM classifiers. Through our method, we generate patterns with a high accuracy in detecting known and unknown packing algorithms. The classification of known and unknown packing algorithms turned out to be a critical result in our research. A classifier can be designed using various approaches for classification. According to Meijer et al. [48], roughly speaking, three different methods can be used. We conducted supervised and unsupervised learning classification based on entropy patterns. The main part of this experiment is to measure the effectiveness in detecting unknown packers. We arranged available unknown packers by determining their nearest similar entropy patterns and placing them in families of analogous patterns. When a family of similar patterns did not exist, we created a new database of families. Figure 11 illustrates our use of two types of classification methods. For the first, we used similarity measurements. For the second, we used NB and SVM, which are both supervised learning classifiers, to classify the known/unknown packing algorithms.

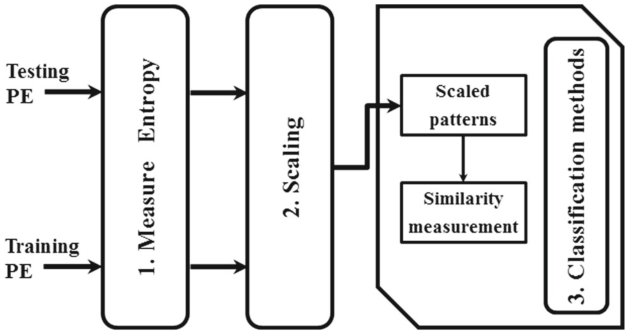

Fig. 10 A similarity classification process using a scaling method for detecting packing algorithms

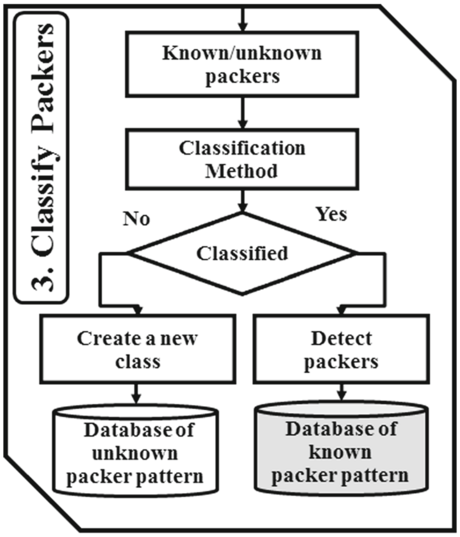

Fig. 11 The structure of a classifier

4.5 Similarity measurements

The similarity is fundamentally important in nearly every scientific field. In addition, the concept of a similarity measurement plays a crucial but less direct role in the modeling of many other psychological tasks. This is especially true in theories on the recognition, identification, and categorization of objects. A common assumption is that the greater the similarity between a pair of objects, the more likely one will be confused with the other. We use the similarity measures to characterize the similarity between the entropy values of each packed executable for the detection of packing algorithms. In this experiment, we used a simplified approach and determined the similarity based exclusively on distance-based similarity measurements.

We used the four metrics, Cosine, Dice, Jaccard, and fidelity, for measuring the similarity between packed executables. The fidelity coefficient similarity shows a much

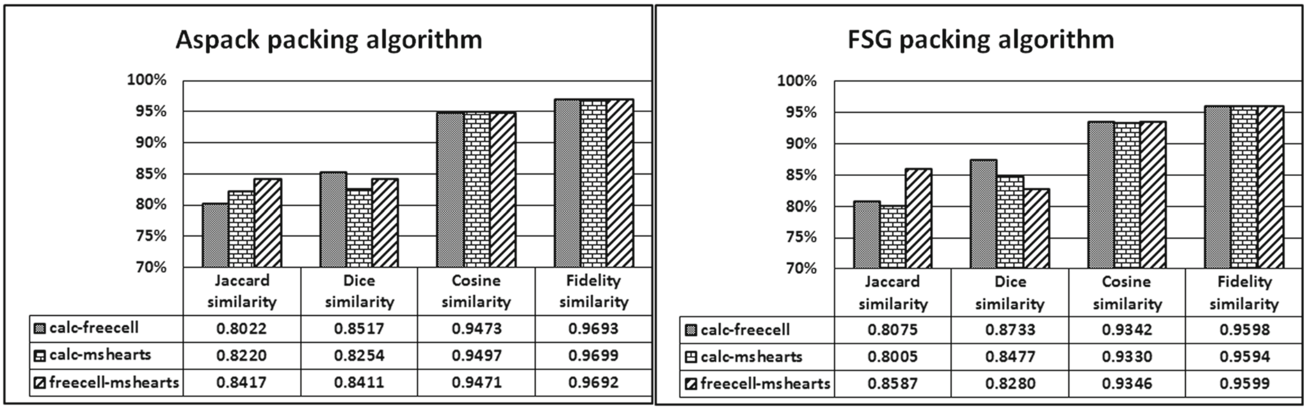

Fig. 12 Four similarity measurement results using the Aspack and FSG packing algorithms individually

Table 1 Similarity measurements of the Aspack and FSG packing algorithms

COSINE similarity	Aspack calc	Aspack freecell	Aspack mshearts	DICE similarity	Aspack calc	Aspack freecell	Aspack mshearts
FSG calc	0.7482	0.7343	0.7433	FSG calc	0.4032	0.6220	0.7233
FSG freecell	0.7413	0.6728	0.7569	FSG freecell	0.4167	0.6281	0.3153
FSG mshearts	0.7115	0.7491	0.7205	FSG mshearts	0.5083	0.5600	0.5609
JACCARD similarity	Aspack calc	Aspack freecell	Aspack mshearts	FIDELITY similarity	Aspack calc	Aspack freecell	Aspack mshearts
FSG calc	0.7740	0.6255	0.5666	FSG calc	0.7695	0.7684	0.7661
FSG freecell	0.2632	0.6368	0.1871	FSG freecell	0.7689	0.7680	0.7644
FSG mshearts	0.7200	0.7545	0.7572	FSG mshearts	0.7690	0.7676	0.7670

higher accuracy result than Cosine, Dice, and Jaccard similarity measurements, as shown in Fig. 12.

Fidelity coefficient similarity: The fidelity is a measurement of the closeness between two distributions. It is widely used in classical and quantum information theories because of its simplicity, its invariance during unitary transformations, and its relation to a trace measurement. In its original form, the fidelity is defined based on two distributions, but herein we extend its definition to include unnormalized sequences of random values. For the given sequences $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ of random positive values $(x_j, y_j > 0)$, fidelity $F(x, y)$ is defined through the following formula.

$$F(x, y) = \frac{\sum_{i=1}^{n} \sqrt{x_i \cdot y_i}}{\sqrt{\sum_{i=1}^{n} x_i} \sqrt{\sum_{i=1}^{n} y_i}} \tag{6}$$

Note that the normalization of sequences is explicitly included and that $F(x, y) = 1$ if and only if $x = y$. In general, $0 \leqslant F \leqslant 1$. All packers are measured based on their similarity using Cosine, Dice, Jaccard, and fidelity. Figure 12 demonstrates the results of three benign executables packed using the same packing algorithms with four similarity measurements. Table 1 shows the Cosine, Dice, Jaccard, and fidelity coefficient similarity measurements in the experimental results of three different benign executables packed using the Aspack and FSG packing algorithms. In our experiment, we calculated the average similarity variables after employing four similarity metrics of benign packed executables using different packing algorithms.

The average similarity variables of the Cosine, Jaccard, Dice, and fidelity similarities are 0.73087, 0.58387, 0.52643, and 0.76766, respectively. Thus, we calculated the average similarity variables of the results shown in Fig. 12 for three benign executables packed using the same packing algorithms. For example, if the same executables are packed using an Aspack packing algorithm, the average similarity variables of the Cosine, Jaccard, Dice, and fidelity similarities are 0.94803, 0.82199, 0.83941, and 0.96946, respectively. Among the four similarity metrics, the fidelity similarity measurement yielded the highest results compared to the other three metrics.

5 Evaluation of the effectiveness of Classification

In this section, we describe the experimental results of our analysis. As previously stated, we proposed a method for detecting the use of packing algorithms. The dataset used in this experiment contained 650 packed executables, 326 of which were packed malware files [49],with the remaining being 324 packed benign executables. The malware samples were collected from the malicious Web sites, *VX heavens* [49], and *Offensive computing* [50]. A data sample of the 326 packed malware executables include a collection of viruses, Trojans, adware, spyware, and few options [50] and [51] as shown in Table 2.

Table 2 demonstrates type of packing algorithms by each family of malware. Surprisingly, a family of packed malware, Packed.Win32.Mondera, demonstrated no sing of using any packing algorithms; however, it might be a false presumption. The family probably used different types of packing algorithm other than demonstrated in Table 2.

Unknown family includes 25 types of different kind of families of packed malware, name and type of some are not recognized, but does not have a significant proportion. This family is only 8 % of the dataset. The data samples of the 324 packed benign executables included a collection of Windows system files and normal executables.

We used 19 popular packers in our experiments and conducted 8100 measurements on packing algorithm detection. In these experiments, the methods of similarity measurements, symbolic representations, and popular forms of classification were used on each packed executable. The data samples were divided into training and testing samples at a ratio of 1:1. The training dataset consists of 162 data samples from 324 packed benign executables, with the remainder being packed malware. Similarly, the testing dataset consists of 163 samples from 326 packed malware, with the remainder being packed benign executables.

When comparing the performances of different classification techniques, assessing how correctly they predict the actual classification of the packers is critical. To introduce the metrics, let us first define the classification of the packers. A packer is positive if it is predicted to be in class A and negative if it is not. Let $\mathscr{A}$ denote the accuracy of classification based on the percentage of test set packers that are correctly identified by the classifier.

$$\mathscr{A} = \frac{(TP + TN)}{n} = \frac{(TP + TN)}{(P + N)} \tag{7}$$

Accuracy $\mathscr{A}$ indicates the overall effectiveness. However, this measure has a certain limitation. Suppose that a test set contains many negative, and a few positive, packers and that we use a classifier to label every class as negative, regardless of the input data. In this case, TN is very high and TP is very low. Despite the primitive nature of the classifier, it will achieve an extremely high classification accuracy on this dataset. Driven by this, the true ($\mathscr{T}_r$) and false ($\mathscr{F}_r$) positive rates are introduced to measure the proportion of positive packers that are correctly identified and the proportion of negative packers that are incorrectly identified, respectively.

5.1 Results of experiments using SAX entropy analysis

We extracted packing algorithm patterns using the SAX representation method. The experimental results of thirteen popular packing algorithms selected from 19 possible packing algorithms are shown in Fig. 5. First, we present the benign "calc. exe" files packed using the 19 packing algorithms. Second, we assign four types of $\phi(\beta)$ values to the packed "calc.exe" executables converted using SAX. In this example, $\phi(\beta) = 10$, $\phi(\beta) = 100$, $\phi(\beta) = 1000$, and $\phi(\beta) = 10,000$ where n=100,000, $\phi(\beta)$ is the number of symbols, and the entropy value is mapped to the character symbols, "abcdefghijklmnopqrstuvwxyz."

The values of M in formula (3) and $\phi(\beta)$ have a reverse relationship. A lower value of M results in a higher beta value. When the value of M is low, the accuracy of converting the entropy values into symbolic representations is high, which can be seen in formula (7).

$$\phi(\beta) = \frac{\text{Entropies}}{M} \tag{8}$$

The number of symbols is expressed as $\phi(\beta)$, as presented in Fig. 13. As can be seen, the accuracy is independent of the number of symbols, $\phi(\beta)$, as is also indicated in Table 3. We show the accuracy results after applying four values of $\phi(\beta)$. For a packed executable using the MPRESS packer, Fig. 13 shows the value of $\phi(\beta)$ when converting the entropy patterns into a symbolic representation.

In other words, $\phi(\beta)$ represents the number of symbols used to extract the packing algorithm patterns. For instance, we packed the calc.exe file using the MPRESS packer. Next, we extracted the entropy pattern by unpacking the file through a measurement and analysis of its entropy value. We then converted the entropy pattern into a character symbol pattern through a symbolic representation. The entropy pattern is expressed by using symbols. In Table 4, the patterns of 19 packing algorithms converted using SAX are shown. We detected packing algorithms from packed executables using the fidelity similarity method. The average accuracy using the different packers is 95.35 %. Table 4 shows the detailed accuracy of the sample dataset.

The detection results of the samples are presented as a confusion matrix. As shown in Table 4, the accuracy of the MPRESS and Mew packing algorithms are both 100 %, whereas the minimum accuracy is 88.3 %, which relates to the PELock packing algorithm. Table 4 shows an average true-positive rate ($\mathscr{T}_r$) of 95.77 %, a false-positive rate ($\mathscr{F}_r$)

Table 2 Result of experiment on packing algorithms (compared by family names)

Family name of	Sample	Percent	Name of Packing Algorithms										
packed malware	data	%	Aspack	MoleBox	Alternate_EXE	FSG	NsPack	RLPack	UPXN	MPRESS	nPack	Morphine	UPX-iT
Packed.Win32.CPEX	42	13			+	+			+				+
Packed.Win32.Klone	19	6								+	+	+	
Packed.Win32.PolyCrypt	8	2	+	+									
Packed.Win32.NSAnti	17	5			+	+	+	+	+				+
Packed.Win32.TDSS	10	3	+	+									
Packed.Win32.PePatch	82	25											
Packed.Win32.Tibs	67	21					+	+	+				+
Packed.Win32.VBCrypt	6	2		+									
Packed.Win32.Katusha	12	4	+										
Packed.Win32.Krap	21	6				+	+	+	+		+	+	+
Packed.Win32.Palevo	11	3			+				+				+
Packed.Win32.Mondera	6	2											
Unknown	25	8											
Total	326	100											

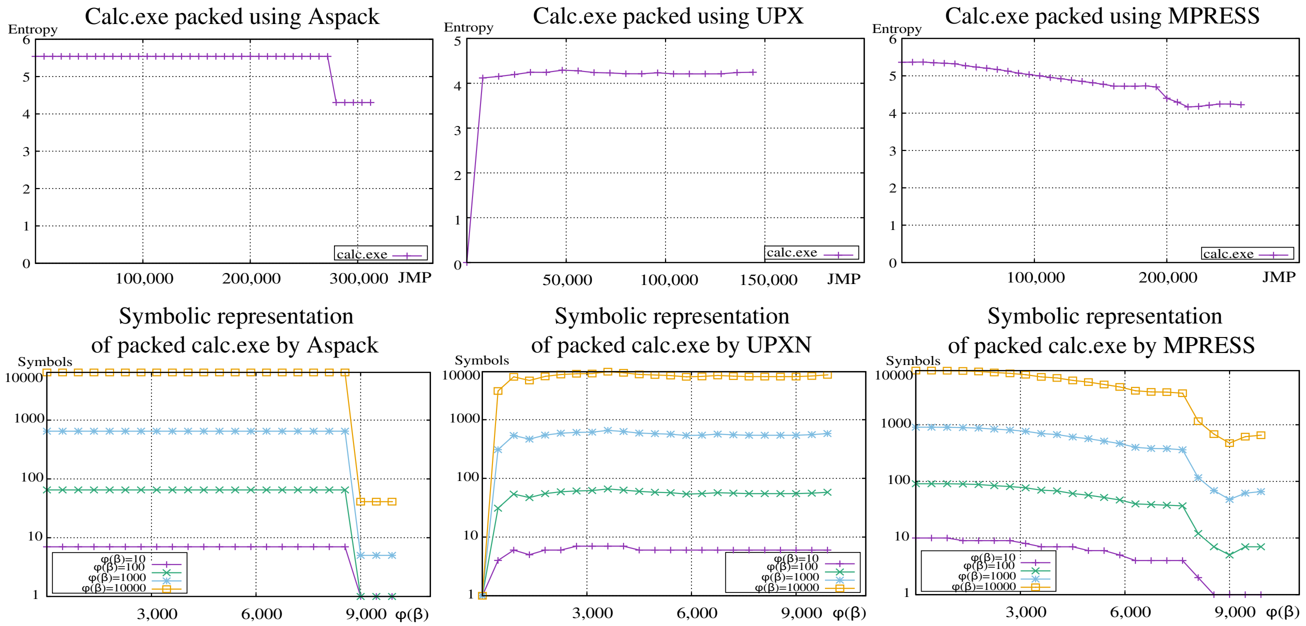

Fig. 13 Entropy patterns converted into symbolic representations using different values of $\phi(\beta)$ for the Aspack, UPXN, and MPRESS packers

Table 3 Results of MPRESS packer converted using SAX

MPRESS $\phi(\beta)$	$\mathscr{T}_r$ (%)	$\mathscr{F}_r$ (%)	$\mathscr{A}$ (%)	$\mathscr{P}$ (%)	$\mathscr{R}$ (%)
10	100.0	3.3	98.0	95.2	100.0
100	100.0	0.0	100.0	100.0	100.0
1000	100.0	0.0	100.0	100.0	100.0
10000	100.0	0.0	100.0	100.0	100.0

of 4.78 %, a precision ($\mathscr{P}$) of 94.13 %, and a recall ($\mathscr{R}$) of 95.83 %.

5.2 Supervised learning classifications of the SAX pattern

Table 5 presents two classifications of packing algorithms based on symbolic-representation data. As the table shows, the accuracy of the NB classification is 98.0 %, which is higher than that of the SVM classification. We can thus classify packing algorithms using an entropy analysis and symbolic representation with a high rate of accuracy. Figure 14 shows three packing algorithms selected from the 19 packing algorithms, i.e., UPXN, Aspack, and MPRESS. The graphs on the right show the conversion of each packing algorithm into a symbolic representation.

The MPRESS packing algorithm reveals a totally different pattern than UPXN and Aspack. However, we easily extracted a new symbolic pattern from each packing algorithm. The example graphs in Fig. 14 show that each packing algorithm is converted into a symbolic pattern with $\phi(\beta) = 10,000$. The experimental results for these notorious packing algorithms suggest that our proposed method is useful for identifying different packing algorithms. In addition, the results show that the proposed method is also applicable to packed malware.

5.3 Incremental aggregate analysis

When some scaled entropy patterns of packing algorithms are similar to each other in one class, detecting packing algorithms using the similarity measurement is very difficult. Therefore, we used an incremental aggregate analysis for each scaled entropy pattern.

Figure 15 shows new sequences of each scaled entropy value created through an incremental aggregate analysis. We then compared the packing algorithm patterns using training dataset with fidelity similarity measurement for the packer detection. In the incremental aggregate analysis, the sequence of the original entropy values is transformed into a new sequence of Σ.

$$\Sigma \equiv (S_0, S_{\max}, S_{\min}, \sigma_1, \ldots, \sigma_m), \tag{9}$$

where $S_0 = s_1$ is the initial entropy value; $S_{\max} \equiv \max\{s_1, \ldots, s_n\}$ and $S_{\min} \equiv \min\{s_1, \ldots, s_n\}$ are the maximum and minimum values, respectively, and $m \equiv n/K$,

Table 4 Detailed accuracy of each packer using the fidelity similarity classification dataset

Num.	PACKERS	$\mathscr{T}_r$ (%)	$\mathscr{F}_r$ (%)	$\mathscr{A}$ (%)	$\mathscr{P}$ (%)	$\mathscr{R}$ (%)
1.	Alternate_EXE	100.0	0.0	100.0	100.0	100.0
2.	FSG	100.0	3.1	98.2	96.0	100.0
3.	RLPack	90.5	8.1	90.2	86.4	90.5
4.	NsPack	95.2	4.8	96.1	95.2	95.2
5.	UPXN	90.5	5.4	92.2	90.5	90.5
6.	UPX-iT	94.5	1.8	94.6	94.8	94.8
7.	MPRESS	100.0	0.0	100.0	100.0	100.0
8.	Morphine	100.0	0.0	100.0	100.0	100.0
9.	nPack	100.0	10.0	94.1	91.7	100.0
10.	Themida	96.3	5.9	96.0	92.3	96.3
11.	VMProtect	96.2	2.9	95.0	92.3	96.2
12.	Aspack	95.2	4.8	96.1	95.2	95.2
13.	MoleBox	100.0	0.0	100.0	100.0	100.0
14.	Petite	92.3	11.5	90.9	91.7	92.3
15.	ASProtect	91.9	9.4	91.9	92.6	92.7
16.	Mew	100.0	0.0	100.0	100.0	100.0
17.	Yoda's Crypter	92.3	4.7	93.5	88.9	92.3
18.	PELock	88.5	9.7	88.3	88.5	88.5
19.	tELock	96.2	8.8	93.3	89.3	96.2
	Average	95.77	4.78	95.35	94.13	95.83

Table 5 Accuracy rates of each classifier

Classification	$\mathscr{T}_r$ (%)	$\mathscr{F}_r$ (%)	$\mathscr{A}$ (%)	$\mathscr{P}$ (%)	$\mathscr{R}$ (%)
Naive Bayes	98.0	1.5	90.4	91.8	98.0
Support vector machine	95.7	2.3	95.5	90.0	95.7
Average	96.8	1.9	92.9	90.9	96.8

with K being the coarse-graining parameter ($K > 1$). In addition,

$$\sigma_j \equiv \begin{cases} 1 & (s_{(j+1)K} - s_{jK} > 0) \\ 0 & (s_{(j+1)K} - s_{jK} = 0) \\ -1 & (s_{(j+1)K} - s_{jK} < 0) \end{cases} \quad (j = 1, \ldots, m) \qquad (10)$$

represents the incremental change in the coarse-grained entropy values. While the transformation focuses on the incremental changes in entropy, it excludes details of the entropy values, as shown in Fig. 16.

In other words, since entropy patterns have discrete form and may fluctuate up and down, we created a new frequency with a newly developed formula (9) (10). Therefore, the sequence Σ is insufficient for classifying all packing algorithms. However, it can be used to classify packing algorithms into several categories quickly and without excessive computational costs. Within each category, different packing algorithms can then be discriminated further by means of various similarity coefficient analysis methods, which may be computationally expensive.

Table 6 shows the results of a new sequence of an incremental aggregate analysis, as shown in formula (9), using the FSG and MPRESS packing algorithms selected from the 19 packing algorithms. For example, we found that the thirteen packing algorithms shown in Fig. 5 are classified through an incremental aggregate analysis into three classes, which we call the increasing, the decreasing, and a combination, respectively. After detecting, the packing algorithm patterns within the class by using the new sequence of incremental aggregate analysis through the fidelity similarity measurement.

- **Increasing class; (A class)** includes Alternate_EXE, FSG, UPXN, NsPack, RLPack, and UPX-iT;
- **Decreasing class; (B class)** consists three packing algorithms, morphine, MPRESS, and nPack;
- **Combination class; (C class)** consists four packing algorithms, VMProtect, Themida, Aspack and MoleBox;

Therefore, FSG and MPRESS are listed in Table 6 as representative packing algorithms of the corresponding classes.

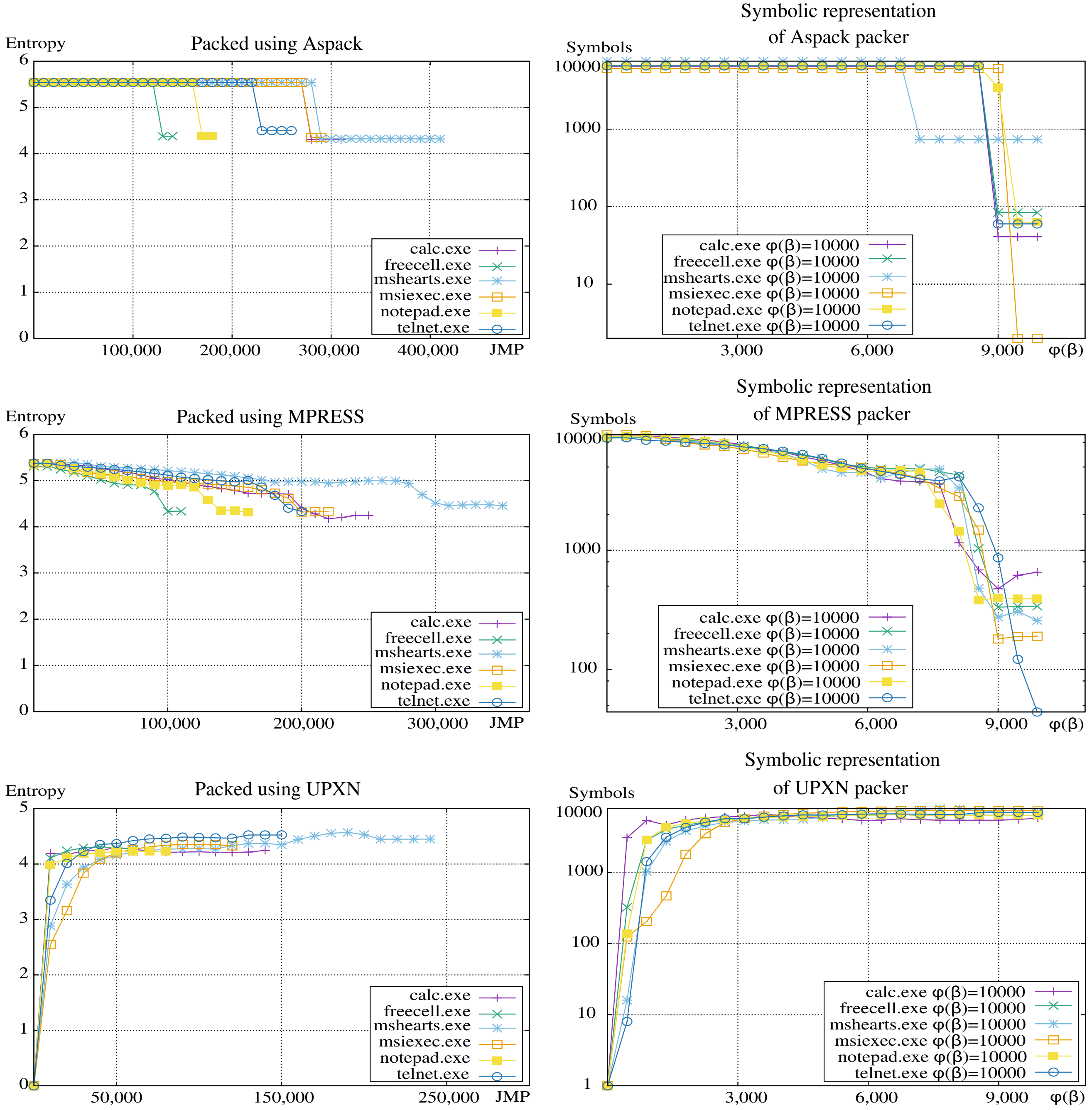

Fig. 14 Experimental results of entropy patterns of three popular packers converted into symbolic representations

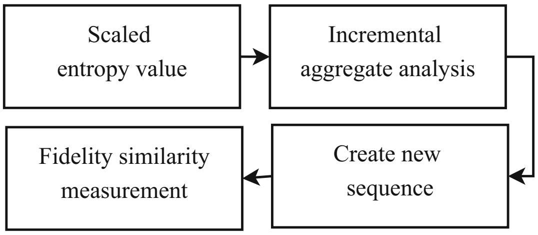
Scaled entropy value
Incremental aggregate analysis
Create new sequence
Fidelity similarity measurement

Fig. 15 Structure of incremental aggregate analysis

5.4 Results of packed malware detection

We analyzed the unpacking process of packed malware using our proposed approach and classified the packed malware through a similarity classification. The packed malware samples were collected from the malicious Web site *VX heavens* [49]. We conducted the experiments using 326 packed malware executables classified into four classes, as shown in Fig. 18. Thus, we can classify 89 % of the packed malware

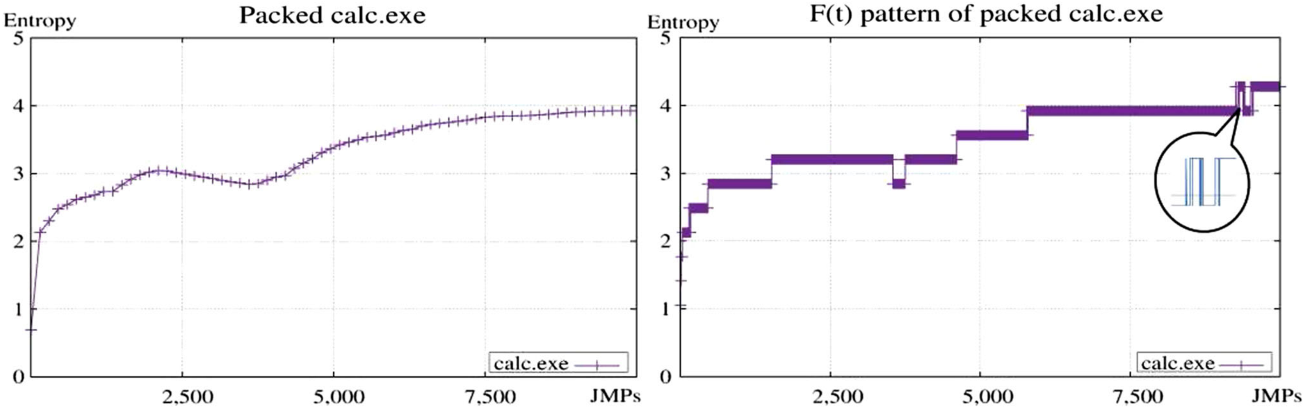

Fig. 16 Σ pattern sequence of the entropy values for classifying packing algorithms

Table 6 Experiment results of Σ function using FSG and MPRESS packing algorithms

FSG	S_{max}	S_{min}	S_0	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7	σ_8	σ_9	σ_{10}
calc	4.2790	0.6931	0.6931	0	1	1	1	1	−1	1	1	1	−1
Freecell	4.3334	0.6937	0.6931	0	1	1	1	1	−1	1	−1	1	1
Mshearts	4.5637	0.6931	0.6931	0	1	1	1	1	−1	−1	1	1	1
Msiexec	4.3274	0.6931	0.6931	0	1	1	1	1	−1	−1	1	−1	1
Notepad	4.3611	0.6931	0.6931	0	1	1	1	1	1	−1	1	−1	1
telnet	4.5246	0.6931	0.6931	0	1	1	−1	1	1	−1	1	1	−1
MPRESS	S_{max}	S_{min}	S_0	σ_{11}	σ_{12}	σ_{13}	σ_{14}	σ_{15}	σ_{16}	σ_{17}	σ_{18}	σ_{19}	σ_{20}
Calc	5.3701	4.1650	5.3619	−1	−1	−1	0	0	0	0	0	0	0
Freecell	5.3205	4.2935	5.3095	1	−1	1	−1	1	−1	1	−1	1	−1
Mshearts	5.3872	4.4526	5.3815	0	0	0	0	0	0	0	0	0	0
Msiexec	5.3710	4.3181	5.3673	1	−1	1	−1	−1	−1	−1	−1	0	0
Notepad	5.3592	4.3113	5.3490	1	−1	1	−1	−1	−1	−1	−1	−1	0
Telnet	5.3863	4.3083	5.3766	−1	−1	0	0	0	0	0	0	0	0

into classes of known packing algorithms (classes A, B, and C) and the remaining 11 % into the class of unknown packing algorithms. For the packed malware to be analyzed in this experiment, Klone, NSanti, Tdss, and PolyCrypt were chosen from *VX heavens*.

We used four variants of Klone, NSanti, Tdss, and PolyCrypt for the selected packed malware collections, i.e., Klone. bg, NSanti.ak, Tdss.c, and PolyCrypt.n, respectively.

Experimental results of the entropy patterns of packed malware are illustrated in Fig. 17. The graphs for Klone.bg and Tdss.c look similar. However, they have three and five sections, respectively. Both use the first section to unpack the instructions. However, the entropy value of the first sections differs from each other. Tdss.c shows a totally different pattern than the former variants and Klone.bg. It is quite obvious that the first section is initialized and is used to unpack the instructions.

NSanti.ak can be classified into Nspack packers of class A, such as those shown in Figs. 5, and 17. Klone.bg and Tdss.c can be classified into MPRESS and Molebox packers of classes B and C, respectively. We conducted a few experimental steps for the detection of packing algorithms used for packed malware. For the first step, we unpacked the packed malware through an entropy analysis. In the second step, we used a scaling process for each pattern of unpacked malware. For the last step, we classified each packed malware based on a similarity classification of the packing algorithms used. The packed malware pattern of Klone.bg looks very similar to the packer pattern of MPRESS (99.98 %) among class B. We therefore used an incremental aggregate analysis for the detection of packing algorithm patterns from each class. The packed malware pattern of Tdss.c has a similarity with the packer pattern of Molebox (99.98 %) among class C. The packed malware pattern of NSanti.ak looks very similar with

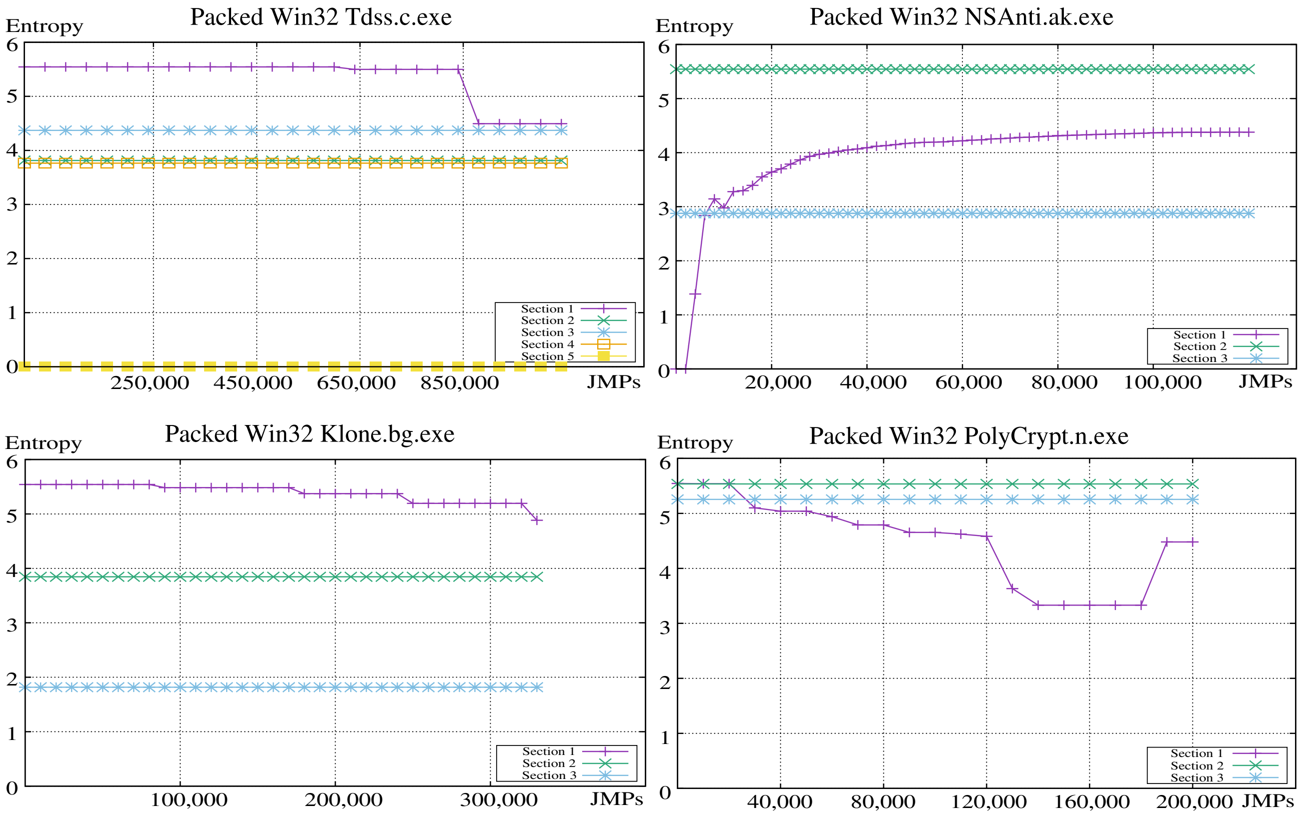

Fig. 17 Real packed malware samples from *VX Heavens*

Table 7 Detection of packing algorithms from packed malware

Class D: patterns of packing algorithms	Pattern of Tdss.c (%)
Aspack	84.95
Molebox	99.98
Class A: patterns of packing algorithms	**Pattern of NSanti.ak (%)**
Alternate_EXE	83.57
FSG	86.54
NsPack	98.60
RLPack	83.93
UPXN	81.36
Class B: patterns of packing algorithms	**Pattern of Klone.bg (%)**
MPRESS	99.98
nPack	80.93
Morphine	75.78

the packer patterns of NsPack (98.6 %) among class A, as shown in Table 7.

The packed malware pattern of PolyCrypt.n differs from the patterns of Tdss.c, NSanti.ak, and Klone.bg, and the training packer patterns (Fig. 5).

In our case, PolyCrypt.n packed malware is detected through the use of an unknown packing algorithm. Therefore, we classified PolyCrypt.n packed malware into the unknown class, as shown in Fig. 18. We confirmed these patterns based on the experimental results, as illustrated in Table 7. These experimental results for these notorious packed malware types imply that the proposed packing algorithm detection methods are useful for analyzing packed malware. The proposed approach is applicable to both known and unknown packed malware, which is a meaningful result. We performed manual verification and compared with our

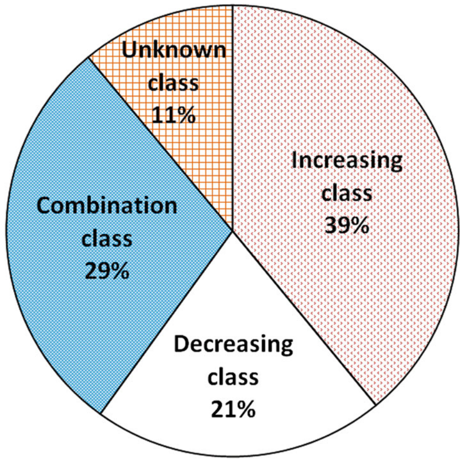

Fig. 18 Classification of packed malware

automatic detection results. Manual unpacking is being done by using debugging, disassembling, and other tools. Results indicated that both results of manual and automatic detections are well matched.

6 Conclusion

This paper examined the use of entropy values of unpacking processes of packed executables for detecting unknown packers. We presented a novel technique for the detection of packing algorithms using SAX representations of the entropy values, and the similarities in the sequence of SAX symbols in each packer. For this method, the low randomness profile of the packing algorithm is extracted and then passed to a pattern classifier.

We also proposed a data conversion method to transform numbers into symbols to greatly reduce the space complexity while preserving the accuracy of detection. Our method proves that the data size (entropy values) can be reduced by one-half to 1/100,000 times. The method presents a packer classification algorithm, which converts entropy patterns (containing numeric values) in large datasets into symbols using a symbolic representation.

In addition, we introduced an incremental aggregate analysis for classify unknown packed benign and malware executables, which then uses four similarity measurements for the classification and detection of packing algorithms based on entropy analysis.

Among the four metrics, the fidelity similarity measurement demonstrates the best matching result, i.e., an accuracy rate of 95.0–99.9 %, which is from 2 to 13 % higher than the accuracy rates of the other three metrics. Our work demonstrates that the randomness profile, when combined with strong pattern-recognition algorithms, produces a highly accurate packer classification system for real data.

The proposed system was tested on a large dataset that includes 324 benign packed files and more than 326 packed malware types. Our method classifies packing algorithms using NB and SVM classifiers using symbolic-representation patterns. In future research, we will extract symbolic patterns from new packed malware, examine multiple packing algorithms, and use additional classification methods for packer classification and detection.

Acknowledgements A preliminary version of this paper was presented at the 8th IEEE International Conference on Malware 2013 [52]. M.-S.Choi acknowledges the support by the National Research Foundation of Korea (Grant No. 2015-003689).

References

1. Symantec Corporation.: Internet Security Threat Report (2014)
2. Choi, H., Zhu, B.B., Lee, H.: Detecting Malicious Web Links and Identifying Their Attack Types. In: WebApps (2011)
3. Yan, W., Zhang, Z., Ansari, N.: Revealing packed malware. IEEE Secur. Priv. **6**(5), 65–69 (2008)
4. Lyda, R., Hamrock, J.: Using entropy analysis to find encrypted and packed malware. IEEE Secur. Priv. **2**, 40–45 (2007)
5. Guo, F., Ferrie, P., Chiueh, T.C.: A study of the packer problem and its solutions. In: Recent Advances in Intrusion Detection, pp. 98–115. Springer, Berlin, Heidelberg, Cambridge (2008)
6. Shafiq, M.Z., Tabish, S.M., Mirza, F., Farooq, M.: Pe-miner: Mining structural information to detect malicious executables in realtime. In: Recent advances in Intrusion Detection, pp. 121–141. (2009)
7. Shafiq, M.Z., Tabish, S., Farooq, M.: PE-probe: leveraging packer detection and structural information to detect malicious portable executables. In: Proceedings of the Virus Bulletin Conference (VB), pp. 29–33. (2009)
8. Saichand, G., Kumar, T.V., Tech, M.: Malwise-An Effective and Efficient Classification System for Packed and Polymorphic Malware, IEEE Transactions on Computer, pp. 1193–1206. (2013)
9. Liu, L., Ming, J., Wang, Z., Gao, D., Jia, C.: Denial-of-service attacks on host-based generic unpackers. In: Information and Communications Security, pp. 241–253. (2009)
10. GitHub.: PEID ser db 2 Yara Conversion. https://github.com/ocean1/peid2yara, (2014)
11. Pasha, M.M.R., Prathima, M.Y., Thirupati, M.L., Malwise System for Packed and Polymorphic Malware, pp. 167–172. (2014)
12. Briones, I., Gomez, A.: Graphs, entropy and grid computing: automatic comparison of malware. In: Virus Bulletin Conference, pp. 1–12. (2014)
13. Sun, L., Versteeg, S., Bozta, S., Yann, T.: Pattern recognition techniques for the classification of malware packers. In: Information Security and Privacy, pp. 370–390. (2010)
14. Adrian, M.: An Analysis of Simile. http://www.securityfocus.com/infocus/1671 (2003)
15. Jacob, G., Comparetti, P.M., Neugschwandtner, M., Kruegel, C., Vigna, G.: A static, packer-agnostic filter to detect similar malware samples. In: Detection of intrusions and Malware, and vulnerability assessment, pp. 102–122. (2012)
16. Perdisci, R., Lanzi, A., Lee, W.: Classification of packed executables for accurate computer virus detection. Pattern Recognit. Lett. **29**(14), 1941–1946 (2008)
17. Santos, I., Ugarte-Pedrero, X., Sanz, B., Laorden, C., Bringas, P.G.: Collective classification for packed executable identification. In: Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference, pp. 23–30. ACM (2011)
18. Cesare, S. and Xiang, Y.: Classification of malware using structured control flow. In: Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-vol. 107, pp. 61–70. (2010)
19. Kolter, J.Z., Maloof, M.A.: Learning to detect malicious executables in the wild. In Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 470–478. ACM (2004)
20. Schultz, M.G., Eskin, E., Zadok, E., Stolfo, S.J.: Data mining methods for detection of new malicious executables. In: IEEE Symposium on Security and Privacy, Proceedings, pp. 38–49. IEEE (2001)
21. Stolfo, S.J., Wang, K., Li, W.J.: Towards stealthy malware detection. In: Malware Detection, pp. 231–249. Springer, US (2007)
22. Tian, R., Batten, L., Islam, R., Versteeg, S.: An automated classification system based on the strings of trojan and virus families. In: MALWARE International Conference on, pp. 23–30. IEEE (2009)
23. Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable. Behavior-Based Malware Clustering. In: NDSS **9**, 8–11 (2009)

24. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Proceedings of the 1st India software engineering conference, pp. 5–14. ACM (2008)
25. Kolbitsch, C., Comparetti, P.M., Kruegel, C., Kirda, E., Zhou, X.Y., Wang, X.: Effective and efficient malware detection at the end host. In: USENIX Security Symposium, pp. 351–366. (2009)
26. Szor, P.: The Art of Computer Virus Research and Defense. Pearson Education, New York (2005)
27. Lee, J., Jeong, K., Lee, H.: Detecting metamorphic malwares using code graphs. In: Proceedings of the ACM Symposium on Applied Computing, pp. 1970–1977. (2010)
28. Vapnik, V.N., Chervonenkis, A.J.: Theory of pattern Recognition: Statistical Problems of Learning, Nauka (1974)
29. Vapnik, V.: The Nature of Statistical Learning Theory. Springer Science & Business Media, New York (2013)
30. Burges, C.J.: A tutorial on support vector machines for pattern recognition. Data Min. Knowl. Discov. **2**(2), 121–167 (1998)
31. Jeong, G., Choo, E., Lee, J., Bat-Erdene, M., Lee, H.: Generic unpacking using entropy analysis. In: Malicious and Unwanted Software (MALWARE), pp. 98–105. IEEE (2010)
32. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: Computer Security Applications Conference, ACSAC, pp. 431–441. IEEE (2007)
33. Kang, M.G., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: Proceedings of the ACM workshop on Recurring malcode, pp. 46–53. ACM (2007)
34. Pietrek, M.: An In-depth Look into the Win32 Portable Executable File Format (2002)
35. Yeung, R.W.: A First Course in Information Theory. Springer Science & Business Media, New York (2012)
36. Costa, M., Goldberger, A.L., Peng, C.K.: Multiscale entropy analysis of biological signals. Phys. Rev. E **71**(2), 1–18 (2005)
37. Costa, M., Healey, J.A.: Multiscale entropy analysis of complex heart rate dynamics: discrimination of age and heart failure effects. In: Computers in Cardiology, pp. 705–708. IEEE (2003)
38. Costa, M., Goldberger, A.L., Peng, C.K.: Multiscale entropy analysis of complex physiologic time series. Phys. Rev. Lett. **89**(6), 21–24 (2002)
39. Nikulin, V.V., Brismar, T.: Comment on multiscale entropy analysis of complex physiologic time series. Phys. Rev. Lett. **92**(8), 804–812 (2004)
40. Pincus, S.M.: Approximate entropy as a measure of system complexity. Proc. Natl. Acad. Sci. **88**(6), 2297–2301 (1991)
41. Pincus, S.M.: Assessing serial irregularity and its implications for health. Ann. NY Acad. Sci. **954**(1), 245–267 (2001)
42. Richman, J.S., Moorman, J.R.: Physiological time-series analysis using approximate entropy and sample entropy. Am. J. Physiol. Heart. Circ. Physiol. **278**(6), H2039–H2049 (2000)
43. Lake, D.E., Richman, J.S., Griffin, M.P., Moorman, J.R.: Sample entropy analysis of neonatal heart rate variability. Am. J. Physiol. Regul. Integ. Comp. Physiol. **283**(3), R789–R797 (2002)
44. Chakrabarti, K., Keogh, E., Mehrotra, S., Pazzani, M.: Locally adaptive dimensionality reduction for indexing large time series databases. ACM Trans. Database Syst. (TODS) **27**(2), 188–228 (2002)
45. Lin, J., Keogh, E., Lonardi, S., Chiu, B.: A symbolic representation of time series, with implications for streaming algorithms. In: Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery, pp. 2–11. ACM (2003)
46. Yi, B.K., Faloutsos, C.: Fast time sequence indexing for arbitrary Lp norms. VLDB, In: Proceedings of the 26th International Conference on Very Large Data Bases, pp. 385–394. (2000)
47. Keogh, E., Kasetty, S.: On the need for time series data mining benchmarks: a survey and empirical demonstration. Data Min. Knowl. Discov. **7**(4), 349–371 (2003)
48. Meijer, B.R.: Rules and algorithms for the design of templates for template matching. In: Pattern Recognition, Conference A: Computer Vision and Applications, In: Proceedings of the 11th IAPR International Conference on, pp. 760–763. IEEE (1992)
49. Baranovich, A.: VX heavens. http://vx.netlux.org
50. Georgia Tech Information Security Center.: Offensive computing (2005)
51. Han, K.S., Lim, J.H., Kang, B., Im, E.G.: Malware analysis using visualized images and entropy graphs. Int. J. Inf. Secur. **14**(1), 1–14 (2015)
52. Bat-Erdene, M., Kim, T., Li, H., Lee, H.: Dynamic classification of packing algorithms for inspecting executables using entropy analysis. In: MALWARE, 8th International Conference on, pp. 19–26. IEEE (2013)