Hindawi
Security and Communication Networks
Volume 2019, Article ID 5278137, 16 pages
https://doi.org/10.1155/2019/5278137

WILEY | Hindawi

Research Article

All-in-One Framework for Detection, Unpacking, and Verification for Malware Analysis

Mi-Jung Choi, Jiwon Bang, Jongwook Kim, Hajin Kim, and Yang-Sae Moon

Department of Computer Science, Kangwon National University, 1 Kangwondaehak-gil, Chuncheon-si, Gangwon 24341, Republic of Korea

Correspondence should be addressed to Yang-Sae Moon; ysmoon@kangwon.ac.kr

Received 10 April 2019; Revised 21 August 2019; Accepted 5 September 2019; Published 13 October 2019

Academic Editor: Jesús Díaz-Verdejo

Packing is the most common analysis avoidance technique for hiding malware. Also, packing can make it harder for the security researcher to identify the behaviour of malware and increase the analysis time. In order to analyze the packed malware, we need to perform *unpacking* first to release the packing. In this paper, we focus on unpacking and its related technologies to analyze the packed malware. Through extensive analysis on previous unpacking studies, we pay attention to four important drawbacks: *no phase integration*, *no detection combination*, *no real-restoration*, and *no unpacking verification*. To resolve these four drawbacks, in this paper, we present an *all-in-one* structure of the unpacking system that performs packing detection, unpacking (i.e., restoration), and verification phases in an integrated framework. For this, we first greatly increase the packing detection accuracy in the detection phase by combining four existing and new packing detection techniques. We then improve the unpacking phase by using the state-of-the-art static and dynamic unpacking techniques. We also present a verification algorithm evaluating the accuracy of unpacking results. Experimental results show that the proposed all-in-one unpacking system performs all of the three phases well in an integrated framework. In particular, the proposed hybrid detection method is superior to the existing methods, and the system performs unpacking very well up to 100% of restoration accuracy for most of the files except for a few packers.

1. Introduction

Recently, as the Internet usage has explosively increased, the risk of malware exposure is also rapidly increasing. According to the 2017 AV-Test security report [1], about six billion malwares are used annually in DDoS (distributed denial of service), spam mails, and APT (advanced persistent threat). In addition, due to the advent of new malwares exploiting analysis avoidance techniques, there have been many research efforts on personal information protection, malicious code detection, and malware analysis technology [2–10]. Among the analysis avoidance techniques, *packing* is the most common one used to hide malware. Packing, also known as "executable compression," is a technique for compressing an executable file to reduce the file size while preserving its format. Packing is originally developed to reduce storage space, but malicious users exploit it to hide malware in the executable file [11, 12]. According to the WildList's 2006 report, more than 92% of malwares are running compression technology [13]. Since packing mostly transforms the original code, we need to perform *unpacking* first before analyzing the packed files which may include malware. In this paper, we focus on such unpacking techniques used for malware detection and analysis.

In order to unpack the packed malware, we need a phase of detecting whether or not the file is packed. If we conclude the file is packed, we restore (i.e., unpack) the file and sometimes verify the unpacked file. However, the existing work has separately developed these three phases of packing detection, unpacking, and verification, and thus, the analyst has difficulty in using all these three phases in an integrated manner. Moreover, there are many detection methods [14–19], but there has been no attempt to combine these detection methods. We also note that the previous unpacking research focuses on finding OEP (original entry point), the first command address where the actual program

starts, but does not address actual restoration of packed files. In addition, even if unpacking is successful, there is no verification for the unpacked files to evaluate their unpacking accuracy or reliability. Based on a thorough survey on recent studies and products, we pay attention to four important observations: (1) no integration of necessary phases, (2) no combination of detection techniques, (3) no real-restoration of packed malware, and (4) no verification of unpacked images. We briefly call these observations *no phase integration*, *no detection combination*, *no real-restoration*, and *no unpacking verification*, respectively.

The goal of this paper is to propose an *all-in-one* unpacking system solving or improving four observations as shown in Table 1. We explain each observation and its solution in detail as follows. First, *no phase integration* is a problem in which analysts have to perform each phase separately since the packing detection, unpacking, and verification phases are separately developed. To resolve this problem, we present an all-in-one unpacking system that integrates all these three phases of packing detection, unpacking, and verification. As shown in Table 2, each existing method focuses on a particular phase. That is, many studies have been done in depth focusing on one of the three phases covered in this paper. In many real applications, however, we often need to apply all three phases at once or sequentially rather than just one phase. To satisfy this demand, the proposed all-in-one system supports all three phases together. Since the system supports all necessary phases in an integrated framework, we can easily analyze the malware and obtain the objective restoration rate through the actual unpacking and verification phases.

The second observation, *no detection combination*, is that there is no attempt to combine various packing detection methods. Based on this observation, we propose a hybrid approach to improve the packing detection accuracy by combining four existing and new packing detection methods. The third observation, *no real-restoration*, is that the previous work tries to find only OEP when it detects packing, but there is little discussion about restoring the actual executable file. We improve this problem by actually restoring the image of the unpacked file. The fourth observation, *no unpacking verification*, is that there is no previous work to measure the restoration accuracy of unpacked images. We resolve this problem by presenting a verification algorithm to quantitatively evaluate the restoration accuracy of unpacked file images.

In this paper, we implement the all-in-one unpacking system to reflect the solutions of Table 1 and empirically evaluate the proposed system. First, we verify each phase of the all-in-one system to confirm that the overall working mechanism works well. Next, we construct a dataset composed of 2,600 PE (portable executable) files and use the set in evaluation of detection, unpacking, and verification phases of the all-in-one system. In the detection phase, we need to determine the entropy range first, and through a preliminary experiment, we set it to be less than 6.00 or greater than 6.85. Experimental results of comparing the proposed hybrid method with individual or combined detection stage(s) show that the proposed method shows the highest detection accuracy up to 98.4% without any false positives. We also try to actually unpack all files of the dataset to verify the unpacking phase, and we confirm that all files including Yoda's Protector [22] are 100% unpacked. Finally, through the evaluation of the verification phase, we see that most files except those packed by some packers show up to 100% restoration accuracy.

The contribution of the paper is as follows. First, this is the first attempt to integrate detection, unpacking, and verification phases into a single unified framework. The proposed all-in-one concept, which integrates all three phases rather than focusing on only one phase, allows users to more easily detect and analyze malware. Second, based on empirical experience, we present a hybrid approach of packing detection that exploits four existing and new detection techniques. Third, we perform actual unpacking restoration beyond detecting OEPs only. Fourth, we propose a verification algorithm that measures the accuracy of unpacking results. Fifth, through various experiments, we demonstrate the superiority of the proposed all-in-one unpacking system.

The rest of the paper is organized as follows. Section 2 describes related work on packing detection and unpacking. Section 3 presents an overall architecture of the proposed all-in-one unpacking system. Section 4 explains detection, unpacking, and verification phases in detail to show how the all-in-one system works. Section 5 presents experimental results. We finally summarize and conclude the paper in Section 6.

2. Related Work

2.1. Packing Detection Techniques. It takes much time to detect and analyze the malware to which packing is applied, and thus, there have been many studies on packing detection and unpacking techniques. Choi et al. [15] propose PHAD that detects packed files by analyzing the header information of PE files. PHAD selects eight characteristic variables to distinguish between the general file and the packed file through heuristic analysis of the PE header, and based on these variables, it determines whether a file is packed or not. More specifically, it calculates the Euclidean distance of eight variables selected by the characteristic vector (CV) and confirms it to be packed if that distance exceeds the heuristically determined minimum threshold. PHAD shows the higher detection accuracy with lower false-negative rates as compared to commonly used software PEiD [23], but has a drawback that many false positives occur.

Lyda and Hamrock [17] use the entropy-based analysis to detect encrypted or packed malware. Entropy is a measure of the uncertainty of information, and packed files tend to have higher entropy than regular files because they compress the original executable sections or collapse those sections into a few new sections. We calculate the entropy as shown in equation (1), where $p(i)$ is the probability of the i-th unit of information (such as a number) in event x's series of n symbols. This equation generates entropy scores as real numbers [17].

Table 1: Observations and solutions for the proposed all-in-one unpacking system.

Observation	Explanation	Solution
No phase integration	Unpacking-related three phases are separately developed	Adopt an all-in-one approach integrating all three phases
No detection combination	There is no attempt to combine various existing methods for packing detection	Combine four packing detection methods to improve detection accuracy
No real-restoration	Main goal is to find OEP	Restore unpacked files by performing actual unpacking as well as finding OEP
No unpacking verification	There is no quantitative way to verify the restoration accuracy	Present a verification algorithm to evaluate the accuracy of unpacking results

Table 2: Comparison of analytical phases supported by existing and proposed systems.

Analytical phases		PHAD [15], REMINDer [16]	PEframe [20]	[11, 13, 21]	All-in-one system
Detection		○	×	×	○
Unpacking	Static	×	○	×	○
	Dynamic	×	×	○	○
Verification		×	×	×	○

$$H(x) = -\sum_{i=1}^{n} p(i)\log_2 p(i). \quad (1)$$

The entropy-based method has an advantage of being easily applied to various packers. However, it causes a false positive if the entropy of a normal file is low or a false negative if that of a packed file is high.

Han and Lee [16] propose REMINDer that detects the packing based on the entropy value of the entry point section and the WRITE attribute. They note that a large number of false positives occur because the entropy calculation range is too wide, and thus, REMINDer uses the EP (entry point) section only in computing the entropy value. In addition to the entropy-based detection method of the EP section, they also use the WRITE attribute test, which is an essential feature of the packed file, to reduce false positives. This is because, unlike ordinary files, packed PE files require WRITE permission to perform unpacking before the file is executed.

Arora proposed by Rohit et al. [19] is a packing detection technique for analyzing the PE information in a packed file using a heuristic algorithm. It defines various parameters for packing analysis and presents a heuristic algorithm for assigning weights and risk factors to those parameters. Based on the training set to which weights and risk factors are assigned, it determines whether or not a given input file is packed.

2.2. Unpacking Techniques. Unpacking techniques are roughly classified into four types. The first is a direct analysis method in which a person unpacks directly using an analysis tool. This method has been popular in the early days, and typical analysis tools include OllyDbg PlugIn [24], Immunity Debugger [25], and IDA PlugIn [26]. This manual unpacking is relatively accurate, but it takes too much time to manually analyze all the commands of the packed file.

The second is a *static unpacking* method based on the characteristics of the specific packing algorithm(s). Since this method is based on the characteristics of the packing algorithm, it is very useful when the prior knowledge of the algorithm is known. Static unpacking is commonly used in antivirus programs, and it has an advantage of no infection risk and fast unpacking since the packed file does not need to be directly executed. However, it has a disadvantage that we cannot use it if we do not know the packing algorithm used or if the packing technique is partially modified.

The third is a *dynamic unpacking* method that does not depend on the packing algorithm(s). There have been many studies on dynamic unpacking since we can do unpacking even without knowing the exact packing algorithm used. Jeong et al. [21] propose an entropy-based dynamic unpacking method that finds the OEP (original entry point) based on the characteristics of increasing the entropy if the file is packed. Bat-Erdene et al. [27] classify unpacking algorithms into multiple clusters by measuring the entropy changes during the unpacking process. Cesare and Yang [28] propose an algorithm for constructing a control flow graph signature using an inverse transformation technique. They perform the entropy analysis first to determine whether or not the PE file is packed, and if it is packed, they find the hidden code by investigating the end of packing through dynamic analysis. Moreover, recent dynamic methods include OmniUnpack [29], Renovo [30], and PinDemonium [31]. OmniUnpack unpacks the PE file by detecting the executing offset of the original code at the *page* level instead of at the *instruction* level. Renovo exploits *shadow memory* to monitor program execution and memory writes, and through the shadow memory, it extracts the hidden code from the executable. Finally, PinDemonium uses the *Dynamic Binary Instrumentation* (DBI) technique to perform *Import Address Table* (IAT) analysis, JUMP command detection, and entropy calculation, and through these processes, it unpacks the PE file.

The fourth is an integration method of both static and dynamic unpacking techniques. Representative examples of such integration methods are PolyUnpack [26], CoDisasm [32], and BinUnpack [3]. PolyUnpack and CoDisasm first extract the *static model* through static analysis before executing the file. Then, they unpack the PE file by comparing the extracted static model to the *dynamic model* obtained by running the executable. PolyUnpack unpacks the file by exploiting iterative comparisons of two models, and CoDisasm does it by comparing memory snapshots repeatedly. Finally, BinUnpack first performs static analysis by developing a *hook-evasion-resistant API monitor*, and it then performs dynamic analysis by monitoring API calls with the analyzed information.

3. Overall Architecture of All-in-One Unpacking System

As we mentioned in Section 1, we note four important observations: no phase integration, no detection combination, no real-restoration, and no unpacking verification. In this section, we describe the overall architecture of the all-in-one unpacking system that considers these four observations. Figure 1 shows a flow diagram of the proposed all-in-one system. As shown in the figure, we improve *no phase integration* by supporting all three phases of detection, unpacking, and verification in an integrated framework. In the diagram, for a given input PE file (1①), the system first determines whether or not it is packed (2②). If the file is not packed, it proceeds to check the next PE file. If it is packed, the system checks whether or not there is an unpacking library used in packing (3③). If the unpacking library exists, the system performs static unpacking using the library (4④); otherwise, the system performs dynamic unpacking that depends on packing algorithm (5⑤). After completing the unpacking, the system calculates the restoration accuracy to verify the correctness of unpacking (6⑥).

We use the existing and proposed techniques together to implement each phase of the all-in-one unpacking system. Table 3 summarizes the existing or proposed methods used in each phase. First, in the detection phase, we use four techniques to detect the packing of the input PE file. This hybrid approach improves *no detection combination*. Among four techniques, the EP section test is a new one proposed in this paper, which increases the packing detection rate by investigating the explicit existence of the EP section, and we describe it in detail in Section 4.1. Second, in the unpacking phase, we use either static unpacking or dynamic unpacking depending on whether or not an unpacking library exists. This unpacking phase restores the actual file and improves *no real-restoration*. Third, in the verification phase, we verify the correctness of unpacking by computing the restoration accuracy. We calculate the restoration accuracy by comparing the byte codes of two files. This phase improves *no unpacking verification*, and we describe this verification algorithm in detail in Section 4.3.

4. Detection Mechanism of All-in-One Unpacking System

4.1. Detection Phase. As we explained in Section 3, while there have been various packing detection techniques, there has been no attempt to combine the advantages of these techniques. Since they have been studied separately, each technique carries out packing detection focusing on its own characteristics only. In this paper, we propose a hybrid approach to improve the packing detection accuracy by combining four different techniques.

The detection phase consists of four tests: EP section, signature, WRITE attribute, and entropy tests. In this paper, we describe each of these techniques as Detection Phase-No EP Section (DP-NEP), Detection Phase-Signature (DP-SIG), Detection Phase-WRITE (DP-WR), and Detection Phase-Entropy (DP-ENT), respectively. Table 4 compares the detection techniques used in the existing work and the proposed approach. As shown in the table, PEiD supports only DP-SIG, [17] supports only DP-ENT, [16] supports DP-WR and DP-ENT, but the proposed method supports all four techniques.

Algorithm 1 presents the proposed hybrid approach of packing detection. It takes the PE file as an input and confirms if the file is packed or not. Since DP-SIG, DP-WR, and DP-ENT work based on the EP section of a PE file, we first check DP-NEP to confirm if the EP section exists. If the EP section does not exist, we determine that the file is packed (Line 1). On the contrary, if the EP section exists, we perform DP-SIG to check whether or not the PE file has a packer signature. If the signature exists, i.e., if a packer is used, we determine that the file is packed (Line 2). We then perform DP-WR and DP-ENT, where DP-WR checks whether or not the WRITE attribute exists in the EP section, and DP-ENT investigates whether or not the entropy of the EP section is in a specific range, *Packing-Range*. For example, this can be under 6.0 or over 0.68 in this paper. If both conditions hold, we determine it packed and return *Packed* (Line 3-4). Finally, anyone of DP-NEP, DP-SIG, DP-WR, and DP-ENP is not satisfied, we return *Not-Packed* (Line 5).

We now explain four packing detection cases of Algorithm 1 in detail. First, DP-NEP finds the EP section from the input PE file. We here conclude that the file is packed if it has no EP section. This is because every PE file has an EP section, so no EP section means that someone hides the section intentionally. We can know the existence of the EP section by investigating an EP section address in the header of the PE file. We explain the effect of this DP-NEP case in detail. According to actual experimental results, we could not find the EP section from some PE files packed by a packer such as Upack [34] and PESpin [35]. This is because such packers intentionally hide or scramble the EP section. Obviously, we need to detect these PE files as the packed ones in the detection phase, and Algorithm 1 does. Algorithm 1 detects such "No EP Section" PE file as the DP-NEP case, resulting in improving the packing detection rate. In other words, considering DP-NEP in Algorithm 1, we exploit the effect of increasing the packing detection rate,

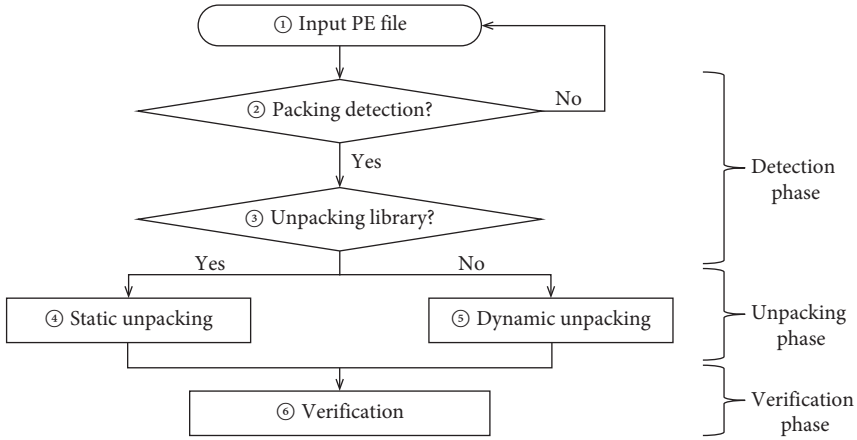

Figure 1: Overall working mechanism of the all-in-one unpacking system.

Table 3: Existing or proposed techniques used in each phase of the all-in-one unpacking system.

Analytical phase	Techniques used or proposed
Detection	(i) EP Section test to be proposed in Section 4.1 (ii) Signature test [23] (iii) WRITE attribute test [16] (iv) Entropy test [17]
Unpacking	(i) Static—library-based unpacking [33] (ii) Dynamic—entropy change-based unpacking [14, 27]
Verification	Verification algorithm to be proposed in Section 4.3

Table 4: Detection techniques used in existing and proposed approaches.

Method	DP-NEP	DP-SIG	DP-WR	DP-ENT
PEiD [23]	×	○	×	×
[17]	×	×	×	○
[16]	×	×	○	○
Proposed approach	○	○	○	○

especially for Unpack and PESpin packers. In the case of DP-NEP, however, we cannot find the EP section itself, and accordingly, we cannot proceed the next phases of unpacking and verification. More precisely, unpacking such DP-NEP packers might be possible if we manually modify the raw hexadecimal codes by using a binary editor. However, this manual raw data modification is beyond scope of the paper, and we do not consider the manual modification of hexadecimal codes. In summary, we use the concept of "No EP Section" in the detection phase of Algorithm 1 to increase the packing detection rate, but we do not consider it in the unpacking and verification phases.

Second, in DP-SIG, we investigate whether or not the signature extracted from the EP section is in the signature database. We construct the signature database with a total of 7,000 signatures, consisting of 4,200 signatures provided by PEiD [23] and 2,800 signatures collected from BobSoft [36]. Figure 2 shows three sample signatures of the packers stored in the database. The first line of each packer represents the type and version of the packer and the maker of the signature; the second line is the hexadecimal values of the packer's signature; and the third line of the ep_only column is true or false that indicates whether the signature can be found in the EP section.

Third, in DP-WR, we check the WRITE attribute of the EP section. In the process of unpacking a packed file, the system needs to modify the memory, so the packed file absolutely requires the WRITE attribute. Thus, we suspect that the file is packed if the WRITE attribute exists. However, since a normal file can also have the WRITE attribute, we cannot determine packing only by the existence of the WRITE attribute. Therefore, as proposed in [16], we detect packing using the WRITE attribute and the entropy value together.

Fourth, in DP-ENT, we compute the entropy of an input PE file using equation (1) and check whether that entropy is in a specific range (*Packing-Range* in Algorithm 1). This is based on the previous observation [16, 17] that the entropy of a packed file differs from that of an ordinary file. We compute the entropy value for the EP section only, not the entire file, using equation (1). To use DP-ENT, we need to set *Packing-Range* used for determining whether or not a file is packed. In this paper, we set this range through the experiment in Section 5.3.

4.2. Unpacking Phase. As mentioned in Section 2.2, unpacking, which is performed automatically without human intervention, can be divided into static unpacking and dynamic unpacking. Static unpacking is based on the characteristics of the packing algorithm, so it is fast and has no risk of infection because of no direct execution of the packed file. On the contrary, dynamic unpacking has a characteristic that it does not depend on a specific packing algorithm even though it takes a long analysis time. In this paper, we use both static and dynamic methods to maximize the unpacking accuracy. More specifically, if the packer type found in the detection phase provides an unpacking library, we use the static unpacking, otherwise we perform the

```
Input: A PE file
Output: Packed or Not-Packed //Packing detection result
begin
(1)   if No EP Section then return Packed; //DP-NEP
(2)   else if Packer Signature Found then return Packed; //DP-SIG
(3)   else if "WRITE" enabled and EP Section Entropy ∈ Packing-Range then
(4)      return Packed; //DP-WR, DP-ENT
(5)   else return Not-Packed;
(6)   end-if
end
```

Algorithm 1: Packing detection.

```
[Code-Lock vx.x]
signature = 47 8B C2 05 1E 00 52 8B D0 B8 02 3D CD 21 8B D8 5A
ep_only = true

[CodeCrypt v0.14b]
signature = E9 C5 02 00 00 EB 02 83 3D 58 EB 02 FF 1D 5B EB 02 0F C7 5F
ep_only = true

[CodeCrypt v0.15b]
signature = E9 31 03 00 00 EB 02 83 3D 58 EB 02 FF 1D 5B EB 02 0F C7 5F
ep_only = true
```

Figure 2: Examples of packer signatures stored in the signature database.

dynamic unpacking. We can easily solve the static unpacking by using the found unpacking library for each packer, so we focus on explaining how the dynamic unpacking works in detail in this section. Our dynamic unpacking method is also known as the "execute-after-write" method. More precisely, the proposed method first restores the original file into the memory by finding the OEP from the packed file, and then analyzes (i.e., executes) the restored original file in the memory. In order to avoid the infection of real machines, we perform the unpacking and validation phases in the debugging mode.

The OEP is the address of the first instruction to execute after the packed file is unpacked, and it represents the entry point of the original program. In other words, since the codes following the OEP represent the starting codes of the original program, finding such OEP is the most important issue in dynamic unpacking. To find the OEP, we use the entropy change-based analysis proposed by Bat-Erdene et al. [27]. When a packed file is executed, the packed data is uncompressed and written to the memory, and at this time, the data might be changed or added to the file. Because of this data change, the entropy value continuously changes during the unpacking process. If unpacking is successful, the entropy value is stabilized, the original file is restored, and the IP (instruction pointer) is moved to another section of the restored original file for its execution. Thus, we can find the OEP by examining whether the IP moves to another section when there is no change in the entropy value of each section. (If the heuristic of investigating change of IP sections is known, someone may create a malicious packing method that can evade the heuristic. However, for this avoidance, the OEP must be found without change of IP sections, and packing a PE file such way might not be easy. According to the actual analysis result, we do not find such cases in the 19 packers we covered. Even though the case hardly exists, it might have occurred in the worst case. We leave the advanced approach that works against such avoidance as a further study). To accomplish this, we need to execute the target process in units of instructions and monitor the entropy change, however, measuring the entropy value after every instruction is very time-consuming and very inefficient. We here note that the OEP generally exists after a branch instruction, so we calculate the entropy value only when the instruction is JMP, Conditional JMP (CJMP), or RETN. We also note that the iterations at program execution are independent of the OEP, so we store and reuse the entropy values measured after JMP, CJMP, and RETN instructions to avoid duplicated calculations by the iterations and reduce the total analysis time.

Algorithm 2 shows the proposed dynamic unpacking algorithm based on the entropy changes, and it returns the *OEP* address for a given PE file. First, it computes the initial entropy of each section (Line 1). Then, it repeats the algorithm until the process ends (Lines 2 to 18). If the current *IP* address is greater than the last address of the current section, it returns *Not-Found* (Line 4). This is because *IP* moves to the next section without a JMP family instruction, so it cannot find *OEP* in this section. On the contrary, if *IP* is one of JMP, CJMP, or RETN, it executes the file up to *IP* and stores the address of the next instruction in *DstAddr* (Lines 5 to 7). But, if *IP* exists in *IP-history* or *DstAddr* exists in *DstAddr-history*, it means an iterative instruction or address, so we move to the next *IP* (Line 8). Otherwise, it stores *IP* and *DstAddr* into each history for the next iteration (Line 9). At this time, if *DstAddr* is out of the range of the file, it returns *Not-Found* because it cannot find *OEP* within the file (Line 10). If *DstAddr* is in the normal range, it calculates the entropy of each section again and compares the current entropy with the previous entropy to check whether it is stable (Lines 11 to 13). If the entropy change is stable and *DstAddr* is in a different section, it means that unpacking a file is successful and *IP* is moved to another section by a JMP family instruction. Thus, it

```
     Input: A PE file
     Output: OEP //the start address of the unpacked original file.
     begin
(1)    Measure the entropy value of each section;
(2)    while the process is not reached to the end do
(3)      IP ⟵ the current instruction pointer;
(4)      if IP's address > current section's last address then return Not-Found;
(5)      else if IP is JMP or CJMP or RETN then
(6)        Execute instructions until IP;
(7)        DstAddr ⟵ the next instruction address;
(8)        if IP ∉ IP-Histroy and DstAddr ∉ DstAddr-History then
(9)          Store IP and DstAddr into each history;
(10)         if DstAddr is out of the file then return Not-Found;
(11)         else
(12)           Re-calculate the entropy value of each section;
(13)           if Entropy change is stable and DstAddr is in the different section then
(14)             return DstAddr;
(15)         end-if
(16)       end-if
(17)     end-if
(18)   end-while
(19)   return Not-Found;
     end
```

Algorithm 2: Dynamic unpacking algorithm.

stops the execution and returns *DstAddr* as *OEP* (Lines 13 and 14).

4.3. Verification Phase. In this section, we present an algorithm for measuring the restoration accuracy by comparing the restored file with its original file. To execute the packed file, the proposed system uses dynamic unpacking and finds the OEP during its execution. If the system finds the OEP, we assume the restoration of an original file is successful and regard that file as the restored file. In the verification phase, we quantitatively measure the restoration accuracy by comparing such restored file and its original file.

Figure 3 shows an example of comparing some bytecodes of the original file with those of the restored file. For the comparison, we first pack the pspasswd.exe file provided by Microsoft [37] and then unpack it again by dynamic unpacking. The figure shows that the bytecodes of the four sections of the original file, .text, .rdata, .data, and .rsrc, are identically appeared in UPX0, UPX1, and .rsrc sections of the restored file. That is, even if the name and location of each section are different, the same bytecodes exist in the restored file. However, the restored file contains padding and garbage values, and some bytecodes may be in other locations in the original file. Thus, in order to compare the original and the restored files, we need to search where the original bytecodes are located in the restored file. Since manually comparing the bytecodes of the original and restored files takes a very long time, in this paper we propose an efficient search algorithm using a hash function. We use the well-known MD5 [38] as a hash function of the search algorithm. As a message reduction algorithm, MD5 is widely used in checking the integrity of long data.

Algorithm 3 shows the restoration accuracy calculation algorithm. Its basic principle is to check whether each section of the original file exists in the restored file. We here use the MD5 hash function to speed up the scan. The inputs to the algorithm are an original file A and a restored file B, and the output is *restorationAccuracy*, a list of storing restoration accuracy values of all sections. First, we remove garbage and padding values from the restored file (Line 1). For each section A_i of A, we obtain the length curLen of the section and calculate a hash value $H1$ by applying the MD5 hash function (Lines 4 to 7). We then calculate another hash value $H2$ from the recovered file B by considering only the length of curLen (Line 9). If $H1$ equals to $H2$ (Line 10), it means that B contains A_i, so we stop the comparison and calculate the restoration accuracy of the section (Line 14). However, if $H1$ and $H2$ are not equal, we perform the hash value comparison again by decrementing the length curLen of A_i (Lines 6 to 13). We repeat this process to calculate the restoration accuracy for the section A_i (Line 14). After completing the accuracy calculation for all sections, the algorithm returns the list restorationAccuracy (Line 16).

5. Performance Evaluation

5.1. Experimental Environment. In this section, we present the results of experimental evaluation to show the superiority of the all-in-one unpacking system. We conduct the experiments in the virtual server as well as the physical server. The reasons for experimenting in two server environments are as follows. First, unpacking results may be different between the virtual server and the physical server if a packed file has anti-unpacking features such as Anti-VM or Anti-Debug. Second, the dynamic unpacking of malware

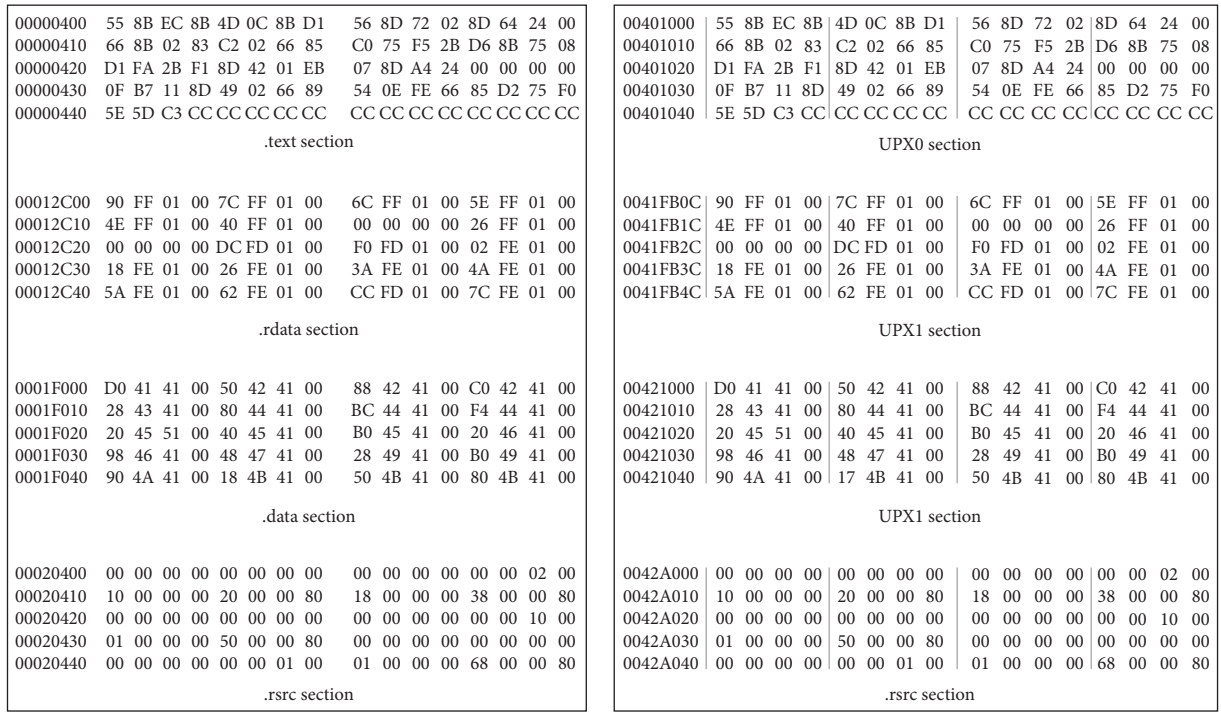

Figure 3: Comparison of sample bytecodes of the original and restored files. (a) Sections of the original file. (b) Sections of the restored file.

Input: A, an original file
B, a restored file
Output: restorationAccuracy[1..n] //a list of restoration accuracy values for n sections
begin
(1) Remove garbage and padding values from B;
(2) $\mathbf{A} \longleftarrow \{A_1, A_2, \ldots, A_n\}$, where A_i is the i-th section of A;
(3) **for each** section $A_i \in A$ **do**
(4) $totalLen \longleftarrow A_i$.length;
(5) $curLen \longleftarrow totalLen$;
(6) **while** $curLen > 0$ **do**
(7) $H1 \longleftarrow$ hash ($A_x[1 : curLen]$); //hash a part of section A_i
(8) **for** $j \longleftarrow 1$ to (B.length $- curLen + 1$) **do**
(9) $H2 \longleftarrow$ hash ($B[j : j + curLen - 1]$); //hash a part of B
(10) **if** $H1 == H2$ **then** break;
(11) **else** $curLen \longleftarrow curLen - 1$;
(12) **end-for**
(13) **end-while**
(14) $restorationAccuracy[i] \longleftarrow (curLen/totalLen) \times 100$;
(15) **end-for**
(16) **return** restorationAccuracy;
end

Algorithm 3: Restoration rate calculation algorithm.

files may infect the physical server. Thus, we perform unpacking first in the virtual server and then perform the experiment in the physical server only when there is no abnormality. The hardware specifications of the experimental platform are as follows. The virtual server is Intel Core i7-6700 3.40 GHz with 8 GB RAM, and the physical server is Intel Core i7-6700 3.40 GHz with 32 GB RAM. The target operating system is Windows 7 OS 32Bit, and we use PEVIEW to view the structure and content of 32 bit PE files. As the debugger, we use Immunity Debugger 1.85 [25] to implement the dynamic unpacking phase of Algorithm 2. To implement the all-in-one unpacking system including the

dynamic unpacking algorithm, we run Python scripts on top of Immunity Debugger.

In the experiment, we use a total 130 PE files, where 90 files are provided by Microsoft [37], and 40 files are randomly collected from putty [39] and MD5 [40] sites. We perform packing of 130 executable files using 19 packers, respectively, and construct a dataset consisting of a total of 2,600 packed files including original 130 files. The 19 packers used are UPX, ASPack, NSPack, MPRESS, Yoda's Protector, MEW, Packman, RLPack, BeRoEXEPacker, PECompact, Petite, JDpack, Molebox, eXpressor, Yoda's Crypter, FSG, exe32pack, WinUpack, and Neolite. The reason for configuring the dataset by packing the normal PE files directly is that we can quantitatively measure the packing detection accuracy, the false-positive rate, and the false-negative rate only if the experimental files are packed in advance.

In this paper, we conduct three experiments. The first experiment is for an integration feature of the all-in-one system, which verifies that the system supports all three phases of packing detection, unpacking, and verification in an integrated framework. The second experiment is for the detection phase, where we first set *Packing-Range*, an entropy range for judging the packing, and then evaluate the packing detection accuracy using the range. The third experiment is for the unpacking and verification phases, where we verify whether the system unpacks the PE file correctly by calculating the restoration accuracy.

5.2. Experimental Results on Overall Working Mechanism. In this section, we confirm through experiments that the integration mechanism of all-in-one unpacking system works correctly. Figure 4 shows the operation screenshot of each phase of the all-in-one system. First, Figure 4(a) shows an example screenshot of the detection phase. To confirm all four cases of packing detection work correctly, even if we detect the packing by the previous case(s), we continue executing the remaining cases. In Figure 4(a), we first examine DP-NEP of Algorithm 2. We detect the EP section in DP-NEP, so we proceed to DP-SIG (the first two lines). By the DP-SIG test, we detect the UPX packer, so we conclude that the file is packed (the second two lines). Even though detection is completed, we proceed to the next DP-ENT and DP-WR tests. The entropy value is 7.93, which is contained in *Packing-Range*, and the attribute value of the EP section is "0xe0000040," which includes the WIRTE attribute. Thus, we conclude that the file is also packed by DP-ENT and DP-WR tests (the third three lines). According to Figure 4(a), we can confirm that DP-NEP, DP-SIG, DP-ENT, and DP-WR tests of the detection phase all work correctly.

Figures 4(b) and 4(c) show example screenshots of the unpacking phase. In the detection phase, if an unpacking library of the packer detected exists, we perform static unpacking as shown in Figure 4(b); if not, we do dynamic unpacking as shown in Figure 4(c). Figure 4(b) shows the result of static unpacking by the UPX commercial tool [33] for the UPX packed file. In the figure, the file size increases from 285, 760 bytes to 731, 200 bytes by unpacking, and it means that the compression ratio is 39.08%. Figure 4(c) shows the result of the entropy-based dynamic unpacking. In the figure, the system calculates the entropy of each section if *IP* encounters JMP and CJMP instructions; it returns the current *DstAddr* as OEP if the entropy change is stable and *DstAddr* is in the different section.

Figure 4(d) shows an example screenshot of the verification phase. In this example, we measure the restoration accuracy of the restored file the .text and .data sections of the original file. The length of the original .text section is 18, 439, and the length of the unpacked file is 872, 555. In the hash value comparison, since the portion corresponding to the original .text section exists in the unpacked file, the restoration accuracy becomes 100%. Similarly, we also calculate the restoration accuracy of the .data section as 100%.

5.3. Experimental Results on Detection Phase. In this section, we describe two experimental results of the detection phase. The first experiment is to set *Packing-Range*, and the second is to measure the detection accuracy of the proposed detection algorithm. In the first experiment, we experimentally determine *Packing-Range* used in DP-ENT of Algorithm 1. The previous study by Lyda and Hamrock [17] uses the entropy value of the whole file, and sets the entropy range of the packed file to $[6.677, 6.926]$. On the contrary, Han and Lee [16] focus on the EP section only and compute the entropy value for the EP section instead of the whole file. Through the experiment, they set the entropy value to 6.85 and judge a file packed if its entropy is over 6.85.

We also determine the entropy range, *Packing-Range*, through the experiments on various packers. Figure 5 shows the entropy distribution of the EP section for different packers. Figure 5(a) shows the average distribution of the original and 19 packed files; Figures 5(b)–5(m) show the more detailed entropy distribution for each packer. According to Figures 5(b)–5(m), the entropy distribution of the packed file differs depending on the packer type. In particular, as shown in Figures 5(d), 5(e), and 5(k), some packed files have smaller entropy values than the original file, and thus, we cannot simply set an upper threshold value as a packed criterion, as in the previous study [16]. However, we note that, while the entropy of packer files distributes over a wide range as shown in Figures 5(c)–5(m), the entropy of original files distributes only in a specific range as shown in Figure 5(b). More specifically, the entropy value of the normal file is between 6.00 and 6.85, as shown in Figure 5(b). Based on these experimental results, we thus set *Packing-Range* to $[0, 6.00)$ or $[6.85, \infty]$ of the EP section. According to [9, 29], the entropy method has the disadvantage of being low accuracy and easy evading. To overcome this point, we use three other detection techniques of DP-NEP, DP-SIG, and DP-WR together with DP-ENT in a hybrid manner. Using this hybrid approach, we can further improve the detection accuracy.

In the second experiment, we evaluate the detection accuracy and false-negative/false-positive rates. Table 5 compares the three detection techniques of the detection phase with the proposed hybrid approach. Since EP sections

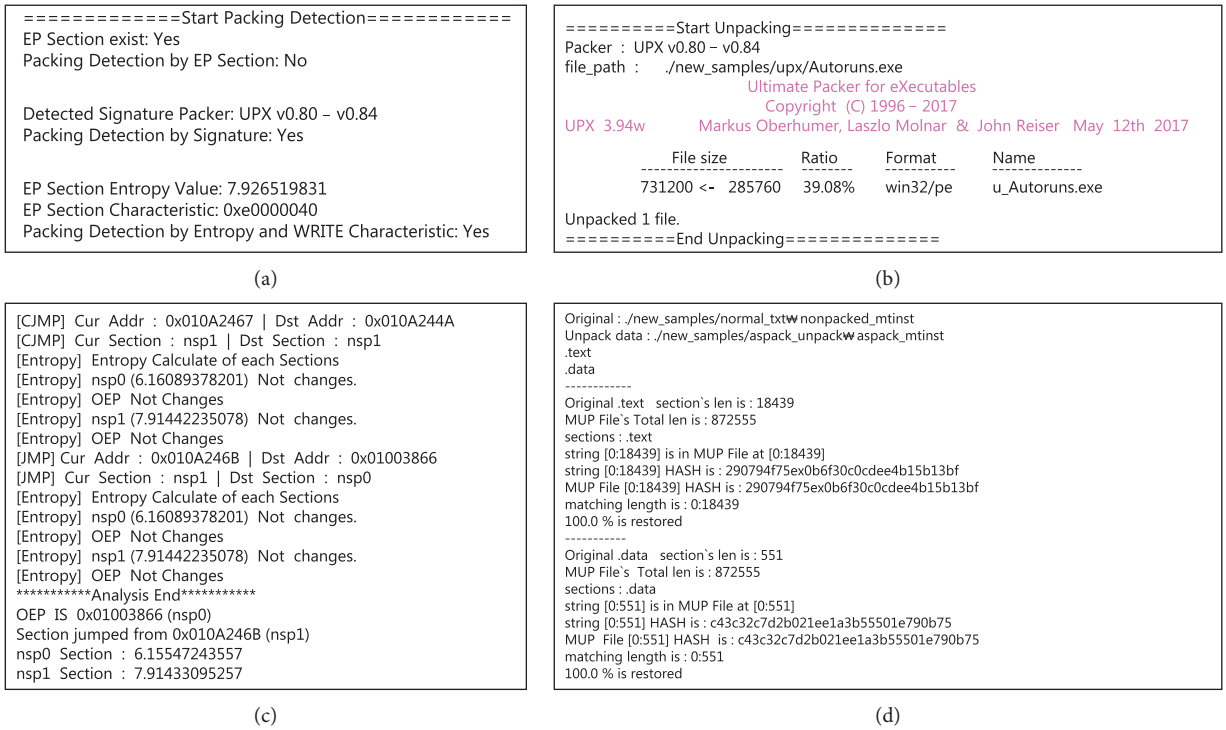

Figure 4: Operation screenshots of detection, unpacking, and verification phases. (a) Detection phase. (b) Unpacking phase (static unpacking). (c) Unpacking phase (dynamic unpacking). (d) Verification phase.

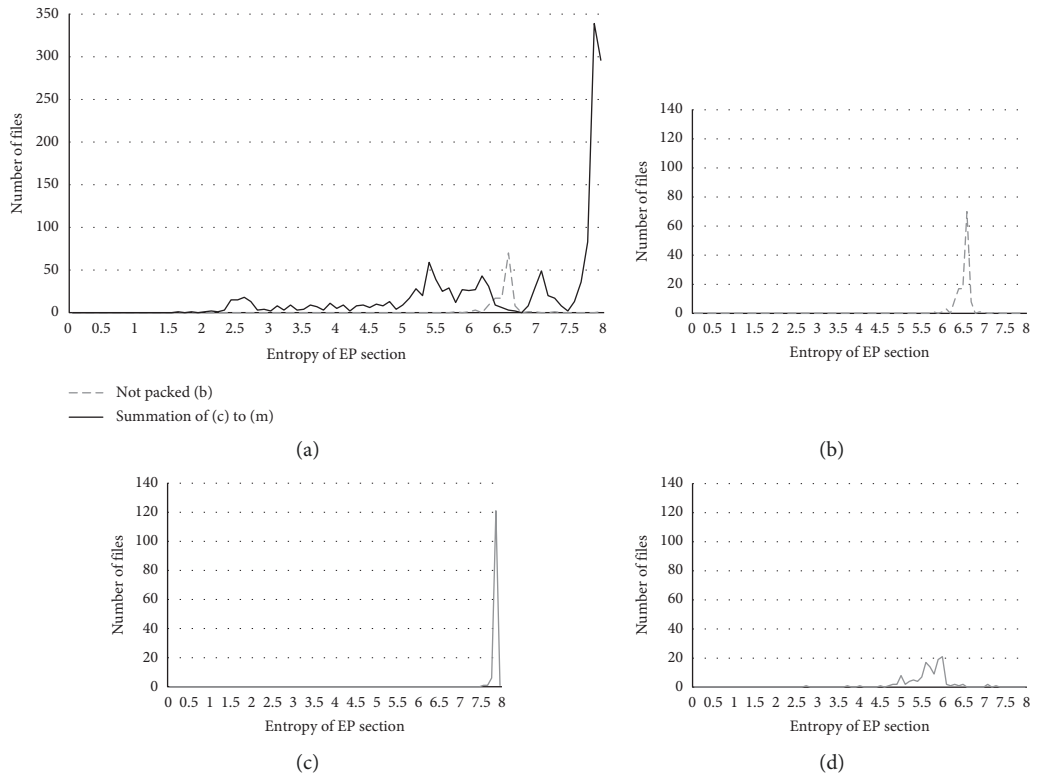

Figure 5: Continued.

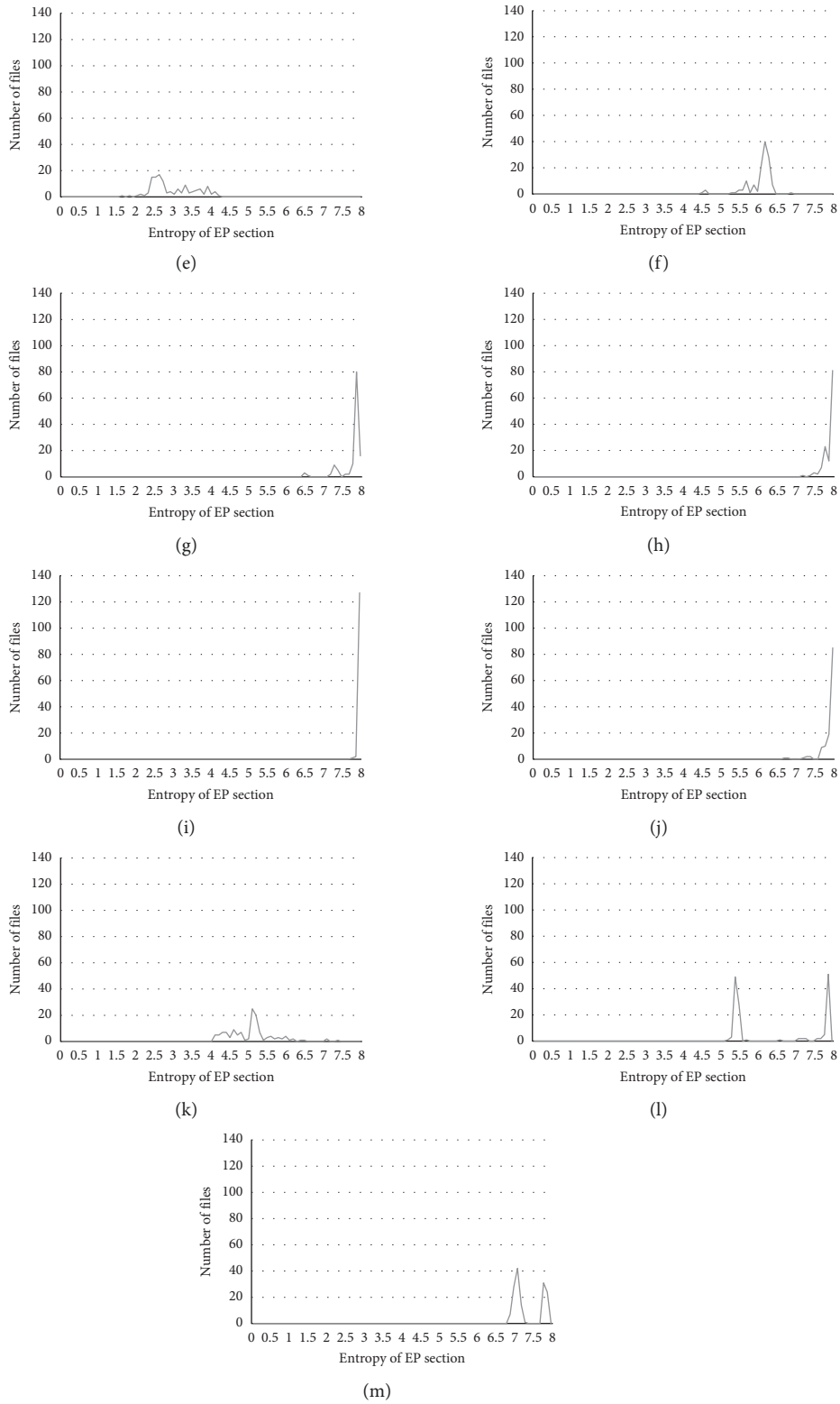

Figure 5: Entropy distributions of the EP section for original and representative packed files. (a) Average of original and 19 packers. (b) Original (not packed). (c) UPX. (d) ASPack. (e) NSPack. (f) MPRESS. (g) Yoda's Protector. (h) RLPack. (i) BeroEXE. (j) MEW. (k) PACKMAN. (l) WinUpack. (m) exe32pack.

Table 5: Comparison of packing detection rates of three techniques and the proposed hybrid approach.

Packer	Technique			
	DP-SIG (%)	DP-WR (%)	DP-ENT (%)	Hybrid approach (%)
ASPack	100	70.0	91.5	100
NSPack	5.38	70.0	100	70.0
MPRESS	100	70.0	24.6	100
UPX	100	70.0	100	100
Yoda's Protector	100	70.0	96.9	100
MEW	100	70.0	100	100
BeRoEXEPacker	100	70.0	100	100
Packman	100	70.0	93.8	100
RLPack	100	70.0	99.3	100
PECompact	100	70.0	100	100
Petite	100	70.0	0.00	100
JDpack	100	70.0	5.62	100
Molebox	100	70.0	100	100
eXpressor	100	70.0	0.00	100
Yoda's Crypter	100	70.0	91.5	100
FSG	100	70.0	100	100
exe32pack	100	0.00	100	100
WinUpack	100	70.0	100	100
Neolite	100	70.0	1.61	100
Average	95.0	66.3	73.9	98.4
Not-Packed	0.00	0.00	3.84	0.00

are mandatory in all PE files, we can detect EP sections unless intentionally concealed. All files of the dataset used in the experiment also have EP sections, and the detection rate by DP-NEP is trivially 0%. In other words, there is no packing detection case due to absence of the EP section, so we exclude the experimental result of DP-NEP from Table 5.

We now explain the results of Table 5 by each column. First, we explain the column of DP-SIG. In the case of DP-SIG, Not-Packed shows a detection rate of 0% since no signature exists in normal files. This is because packing detection does not occur in normal files, which means that DP-SIG performs an accurate detection for Not-Packed files. On the other hand, for the packed files, DP-SIG detects 100% of packers except NSPack. The detection rate of NSPack is merely 5.38% because of the nature of NSPack that the signatures often exist in different sections rather than the EP section. In summary, DP-SIG shows 95.0% of detection accuracy on average.

Second, for the experiment on the detection rate of DP-WR, we intentionally remove the WRITE attribute from 39 files (30% of packed files) per packer. We perform this removal process in 18 packers except exe32pack. The reason for changing the dataset is that normal packed files usually have WRITE attributes, so we arbitrarily remove the WRITE attribute from 30% of files for the DP-WR experiment. Looking at the DP-WR column in Table 5, DP-WR detects only 70% files per packer for 18 packers since we intentionally remove the WRITE attribute from 30% files. In case of exe32pack, however, we need not perform the removal process since it has no WRITE attribute in the PE header. And accordingly, as shown in the column DP-WR of Table 5, the detection rate of exe32pack by DP-WR is trivially 0%. In the case of Not-Packed, the detection rate is 0% since normal files have no WRITE attribute. In summary, checking the presence of the WRITE attribute is very simple, and DP-WR works correctly if the packed file has the WRITE attribute; but, it does not work well if the WRITE attribute does not exist or maliciously modified.

Third, the column of DP-ENT shows the result of entropy-based detection rates. As shown in the table, it shows the 100% detection rate for 9 packers including NSPack, UPX, MEW, and BeRoEXEPacker, but it incurs some false negatives for 10 packers including ASPack, Yoda's Protector, Packman, and MPRESS by showing the detection rate lower than 100%. In particular, for MPRESS, Petite, JDpack, eXpressor, and Neolite, it shows a low detection rate below 24.6% because their entropy value in the packer characteristic is in entropy distribution of the normal file. DP-ENT also has a false-positive rate of 3.84% for Not-Packed. In summary, DP-ENT works very accurately for most packers, but it may incur false negatives for a few specific packers or false positives for some normal files. To overcome this limitation, we use DP-ENT together with other detection techniques. We can see the improvement through the experimental results of Tables 5 and 6. In the tables, when using only DP-ENT, the detection accuracy is low in some packers (especially, MPRESS), but when using it together with the other three techniques, the accuracy is close to 100%.

Fourth, the proposed hybrid approach of integrating four techniques detects 100% of all packers except NSPack and shows no false positives. The reason for this high detection rate is that the proposed method has all the advantages of DP-SIG, DP-WR, and DP-ENT as well as DP-NEP. In case of NSPack, however, the detection rate is merely 70.0%; this is because, as mentioned above, we forcedly remove the WRITE attribute from the 30% file. According to the results of Table 5, detection rate differs according to each detection technique, and some techniques

Table 6: Comparison of packing detection accuracy of two or three combinations and of the hybrid approach.

Metrics	DP-{SIG, ENT} (%)	DP-{ENT, WR} (%)	DP-{SIG, ENT, WR} (%)	Hybrid approach (%)
Detection accuracy	95.0	92.0	98.4	98.4
False-negative rate	5.00	8.00	1.60	1.60
False-positive rate	0.73	0.00	0.00	0.00

incur false positives or false negatives; on the other hand, the proposed hybrid shows the highest detection rate of 98.4% and no false positives.

Table 6 shows the detection accuracy of two or three combinations of detection techniques and of the proposed hybrid approach. First, DP-{SIG, ENT}, a combination of signature and entropy tests, detects 100% of the packed file but detects 18 files incorrectly, showing a false-positive rate of 0.73%. Second, DP-{ENT, WR}, a combination of entropy and WRITE tests, shows a low detection rate of 92.0% because the entropy value of MPRESS is in the distribution of normal file. Third, DP-{SIG, ENT, WR}, a combination of signature, entropy, and WRITE tests, and our hybrid approach show 98.4% detection accuracy and 1.60% false-negative rate. This is because DP-{SIG, ENT, WR} and the hybrid approach go through the same detection procedure except DP-NEP. The false-negative rate of 1.60% is due to the files having no WRITE attributes. According to the results of Tables 5 and 6, we believe that the proposed hybrid approach is an excellent method for high detection accuracy by combining the advantages of the existing and new detection techniques.

5.4. Experimental Results on Unpacking and Verification Phases. In this section, we verify the unpacking and verification phases by performing the unpacking of Algorithm 2 and calculating the restoration accuracy of the files through the actual restoration of Algorithm 3. However, unpacking and verification phases take a lot of time, so we experiment with randomly selected 40% files of the dataset. The unpacking phase consists of static unpacking and dynamic unpacking. In the experiment, we exclude static unpacking since it uses the existing library as it is, and focus on only dynamic unpacking of Algorithm 2 since it tries to restore the original file without any information of packers.

Table 7 shows the restoration accuracy of the proposed method for .text and .data sections for each of the 19 packers. We cannot directly unpack Yoda's Protector and PECompact since they use Anti-Debug engineering techniques. To overcome this problem, we unpack these packers by bypassing three Anti-Debugging API calls: GetCurrentProcessID(), BlockInput(), and IsDebuggerPresent() [41]. That is, we run Algorithm 2 with the bypassing techniques for Yoda's Protector and PECompact, and include its experimental result in Table 7). In general, since only .text and .data sections of the original file are packed [42], in the experiment we consider only the restoration accuracy of these .text and .data sections of the restored file. In Table 7, among 19 packers, 11 packers including UPX, NSPack, MEW, RLPack, and BeRoEXEPacker show at 100% restoration accuracy for both text and data sections. On the other hand, 8 packers including ASPack, Packman, and MPRESS show less than 100% restoration accuracy because of the manner in which the packers handle the .reloc section. More specifically, if there is a .reloc section in the original file, the unpacking relocates the .text and .data sections, changing all the data values in each section. Therefore, the restoration accuracy depends on whether there is a .reloc section in the original file.

Table 8 shows the restoration results in detail according to the presence of the .reloc section. As shown in the table, the restoration accuracy is 100% in the absence of the .reloc section, while it is very low in the range of 0.00% to 9.98% in the presence of the .reloc section. However, the low restoration accuracy is due to the nature of the packer, and it does not mean that unpacking does not work correctly. In other words, even if the unpacking is successful, the data value may change due to the packer characteristic, and the restoration accuracy is calculated to be low. In summary, we can use the proposed verification algorithm to calculate the restoration accuracy of unpacked files, which can quantitatively measure the accuracy of the actual restoration.

In addition to .text, .data, and .reloc sections, we further consider the PE header, which includes the IAT information. In general, the IAT is located in the header of the PE file, and if we actually need to run the unpacked file, we must also restore this header information, including the IAT. In fact, the study of [3, 43] restores the header including the IAT and produces the actual whole executable file. On the other hand, most studies, including this paper, do not restore the actual whole executable file itself, but restore .text, .data, and .reloc sections, which are major parts of the executable file. This is because the purpose of unpacking is not to execute the restored file, but to identify and extract malicious codes included in the restored file. Therefore, in this paper, we do not need to restore the header required for the actual execution, and accordingly, comparison of IAT information is also not necessary. Instead, we restore and compare .text, .data, and .reloc sections that are more likely to hide malicious code. Restoring the PE header is beyond the scope of this study, we leave it as a separate research topic.

6. Conclusions

In this paper, we analyzed recent studies of packing detection and unpacking techniques, and derived four important observations: *no phase integration*, *no detection combination*, *no real-restoration*, and *no unpacking verification*. We then proposed an *all-in-one unpacking system* to improve these four observations. First, no phase integration was a difficulty in which analysts had to perform packing detection, unpacking, and verification phases separately, and we solved this problem by presenting an all-in-one

Table 7: Restoration accuracy of .text and .data sections of each packer.

Section	UPX (%)	NSPack (%)	MEW (%)	RLPack (%)	BeRoEXE (%)	Neolite (%)	FSG (%)	eXpressor (%)	Molebox (%)	Petite (%)	JDpack (%)	ASPack (%)	Yoda's protector (%)	exe32pack (%)	MPRESS (%)	Yoda's crypter (%)	PECompact (%)	WinUpack (%)	Packman (%)	Average (%)
.text	100	100	100	100	100	100	100	100	100	100	100	84.3	84.0	80.8	78.8	74.5	74.0	68.0	55.0	89.2
.data	100	100	100	100	100	100	100	100	100	100	100	84.3	84.0	100	75.3	74.5	74.0	68.0	48.5	90.2

Table 8: Restoration accuracy according to the presence of the .reloc section.

Section	ASPack (%)		Packman (%)		Mpres (%)		Yoda's crypter (%)		PECompact (%)		Yoda's Protector (%)		exe32pack (%)		WinUpack (%)	
	○	×	○	×	○	×	○	×	○	×	○	×	○	×	○	×
.text	0.00	100	9.98	100	0.00	100	0.00	100	0.00	100	0.00	100	0.00	100	0.00	100
.data	0.00	100	8.81	100	0.00	100	0.04	100	0.00	100	0.05	100	100	100	0.00	100

unpacking system of integrating these three phases in an integrated framework. Second, no detection combination was due to that there was no attempt to combine various packing detection methods, and we solved this problem by presenting a hybrid approach of combining existing and new packing detection methods. Third, no real-restoration came from that there was little discussion about restoring the actual executable file, and we solved this problem by actually restoring an original image of the packed file. Fourth, no unpacking verification was due to that there was no explicit work to measure the restoration accuracy of unpacked files, and we solved this problem by proposing a verification algorithm of quantitatively evaluating the restoration accuracy.

We constructed a dataset of 2,600 PE files and experimentally evaluated the proposed system using the dataset. First, we performed all the phases of the all-in-one unpacking system sequentially and confirmed that its overall working mechanism worked well. Next, through the comparison of the hybrid packing detection approach and existing detection techniques, we showed that the detection accuracy of the proposed method was the highest at 98.4% on average with no false positives. For this, we also determined an entropy range of packing, Packing-Range, through extensive experiments on 19 representative packers. Finally, in the unpacking and verification phases, we confirmed that unpacking was successful for all files except the files packed with Yoda's Protector. We also showed that the restoration accuracy of unpacked files was nearly 100% except for a few packers.

As the future research, we will focus on unpacking packed files with Anti-VM, Anti-Debug, and Anti-Reverse engineering techniques. We will also study how to compute the restoration accuracy of the file having .reloc sections.

Data Availability

All the data files used in the experiments are available at https://github.com/chesvectain/PackingData.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was partly supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korea government (MSIT) (no. 2017-0-00158, Development of Cyber Threat Intelligence (CTI) analysis and information sharing technology for national cyber incident response), and the Korea Electric Power Corporation (no. R18XA05).

References

[1] M. Morgenstern, "AV-TEST Security report 2017/18," 2019, https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2017-2018.pdf.

[2] T. Ban, R. Isawa, S. Guo, D. Inoue, and K. Nakao, "Efficient malware packer identification using support vector machines with spectrum kernel," in *Proceedings of the Eighth Asia Joint Conference on Information Security*, pp. 69–76, Seoul, South Korea, July 2013.

[3] B. Cheng, M. Jiang, J. Fu et al., "Towards paving the way for large-scale Windows malware analysis: generic binary unpacking with orders-of-magnitude performance boost," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 395–411, Toronto, Canada, October 2018.

[4] A. Gupta and M. S. Arya, "Hashing based encryption and anti-debugger support for packing multiple files into single executable," *International Journal of Advanced Research in Computer Science*, vol. 9, no. 1, pp. 914–920, 2018.

[5] Y. Kawakoya, M. Iwamura, and M. Itoh, "Memory behavior-based automatic malware unpacking in stealth debugging environment," in *Proceedings of the 5th International Conference on Malicious and Unwanted Software*, pp. 39–46, Lorraine, France, October 2010.

[6] G. Liang, J. Pang, Z. Shan, R. Yang, and Y. Chen, "Automatic benchmark generation framework for malware detection," *Security and Communication Networks*, vol. 2018, Article ID 4947695, 8 pages, 2018.

[7] A. Pektaş and T. Acarman, "A dynamic malware analyzer against virtual machine aware malicious software," *Security and Communication Networks*, vol. 7, no. 12, pp. 2245–2257, 2014.

[8] I. Santos, J. Nieves, and P. G. Bringas, "Semi-supervised learning for unknown malware detection," in *International Symposium on Distributed Computing and Artificial Intelligence*, A. Abraham, J. M. Corchado, S. R. González, and J. F. De Paz Santana, Eds., pp. 415–422, Springer, Berlin, Germany, 2011.

[9] X. Ugarte-Pedrero, I. Santos, B. Sanz, C. Laorden, and P. G. Bringas, "Countering entropy measure attacks on packed software detection," in *Proceedings of the 9th International Conference on Consumer Communications and Networking*, pp. 164–168, Las Vegas, NV, USA, January 2012.

[10] H. Won, S.-P. Kim, S. Lee, M.-J. Choi, and Y.-S. Moon, "Secure principal component analysis in multiple distributed nodes," *Security and Communication Networks*, vol. 9, no. 14, pp. 2348–2358, 2016.

[11] S. Cesare, Y. Xiang, and W. Zhou, "Malwise–an effective and efficient classification system for packed and polymorphic

malware," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1193–1206, 2013.

[12] C. Malin, E. Casey, and J. Aquilina, *Malware Forensics: Investigating and Analyzing Malicious Code*, Syngress, Burlington, MA, USA, 2008.

[13] W. Yan, Z. Zheng, and A. Nirwan, "Revealing packed malware," *IEEE Security and Privacy*, vol. 6, no. 5, pp. 65–69, 2008.

[14] M. Bat-Erdene, T. Kim, H. Park, and H. Lee, "Packer detection for multi-layer executables using entropy analysis," *Entropy*, vol. 19, no. 3, p. 125, 2017.

[15] Y.-S. Choi, I.-K. Kim, J.-T. Oh, and J.-C. Ryou, "PE file header analysis-based packed PE file detection technique (PHAD)," in *Proceedings of International Symposium on Computer Science and Its Applications*, pp. 28–31, Hobart, Australia, October 2008.

[16] S. Han and S. Lee, "Packed PE file detection for malware forensics," *The KIPS Transactions: PartC*, vol. 16C, no. 5, pp. 555–562, 2009, in Korean.

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

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

[19] A. Rohit, A. Singh, H. Pareek, and U. Edara, "A heuristics-based static analysis approach for detecting packed PE binaries," *International Journal of Security and Its Applications*, vol. 7, no. 5, pp. 257–268, 2013.

[20] G. Amato, "PEframe," 2019, https://github.com/guelfoweb/peframe.

[21] G. Jeong, E. Choo, J. Lee, M. Bat-Erdene, and H. Lee, "Generic unpacking using entropy analysis," *Journal of Advanced Information Technology and Convergence*, vol. 7, no. 1, pp. 232–238, 2009.

[22] V. DeMarines, "Obfuscation–how to do it and how to crack it," *Network Security*, vol. 2008, no. 7, pp. 4–7, 2008.

[23] H. Neil, "PEiD," 2019, https://github.com/wolfram77web/app-peid.

[24] O. Yuschuk, "OllyDbg," 2019, http://www.ollydbg.de/download.html.

[25] N. Waisman, "Immunity Debugger," 2019, https://www.immunityinc.com/products/debugger/index.html.

[26] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the hidden-code exetraction of unpack-executing malware," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, pp. 289–300, Miami Beach, FL, USA, December 2006.

[27] M. Bat-Erdene, Y. Park, H. Li, H. Lee, and M.-S. Choi, "Entropy analysis to classify unknown packing algorithms for malware detection," *International Journal of Information Security*, vol. 16, no. 3, pp. 227–248, 2017.

[28] S. Cesare and X. Yang, "Classification of malware using structured control flow," in *Proceedings of the 8th Australasian Symposium on Parallel and Distributed Computing*, pp. 61–70, Brisbane, Australia, January 2010.

[29] L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: fast, generic, and safe unpacking of malware," in *Proceedings of the 23rd Annual Computer Security Applications Conference*, pp. 431–441, Miami Beach, FL, USA, December 2007.

[30] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: a hidden code extractor for packed executables," in *Proceedings of the 5th ACM Workshop on Recurring Malcode*, pp. 46–53, Alexandria, VA, USA, November 2007.

[31] S. Mariani, L. Fontana, F. Gritti, and S. D'Alessio, "PinDemonium: a DBI-based generic unpacker for windows executables," in *Proceedings of the Black Hat 2016*, Las Vegas, NV, USA, July 2016.

[32] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "CoDisasm: medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 745–756, Denver, CO, USA, October 2016.

[33] M. Oberhumer, L. Molnar, and J. Reiser, "UPX," 2019, https://upx.github.io/.

[34] A. Swinnen and A. Mesbahi, "One packer to rule them all: empirical identification, comparison and circumvention of current antivirus detection techniques," in *Proceedings of the Black Hat 2014*, Las Vegas, NV, USA, August 2014.

[35] P. Bania, "Generic unpacking of self-modifying, aggressive, packed binary programs," 2009, https://arxiv.org/abs/0905.4581.

[36] H. Neil, "BobSoft Signature," 2019, http://woodmann.com/BobSoft/Files/Other/UserDB.zip.

[37] M. Russinovich, "Microsoft Sysinternals Suite," 2019, https://download.sysinternals.com/files/SysinternalsSuite.zip.

[38] R. Rivest, "The MD5 message-digest algorithm," RFC 1321, IETF, Fremont, CA, USA, 1992.

[39] S. Tatham, "Putty," 2019, https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html.

[40] L. Pascoe, "MD5," 2019, http://www.md5summer.org/about.html.

[41] P. Ferrie, "Anti-unpacker tricks - part one," 2019, https://www.virusbulletin.com/virusbulletin/2008/12/anti-unpacker-tricks-part-one.

[42] T.-Y. Wang and C.-H. Wu, "Detection of packed executables using support vector machines," in *Proceedings of the 10th International Conference on Machine Learning and Cybernetics*, pp. 717–722, Guilin, China, July 2011.

[43] D. Korczynski, "RePEconstruct: reconstructing binaries with self-modifying code and Import address table destruction," in *Proceedings of the 11th International Conference on Malicious and Unwanted Software*, pp. 31–38, Fajardo, Puerto Rico, October 2016.