# REFORM: A Framework for Malware Packer Analysis Using Information Theory and Statistical Methods

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in Mathematics

**Li Sun**

B. Engineering, M. Internet and Web Computing

School of Mathematical and Geospatial Sciences,

College of Science, Engineering and Health,

RMIT University

April, 2010

# DECLARATION

The candidate hereby declares that the work in this thesis, presented for the award of Doctor of Philosophy in Mathematics and submitted in the School of Mathematical and Geospatioal Sciences, RMIT University:

- **has been done by the candidate alone and has not been submitted previously, in whole or in part, in respect of any other academic award and has not been published in any form by any other person except where due reference is given, and**
- **has been carried out under the supervision of Associate Professor Serdar Boztaş**

........................................

**Li Sun**

# Certification

This is to certify that the above statements made by the candidate are correct to the best of our knowledge.

.......................................

**Associate Professor Serdar Boztaş**

**Supervisor**

# Abstract

Malware (malicious software) is a term used to describe computer viruses, Trojan horses, and other pieces of software that are used to attack computer systems. The increasing outbreak of malware in recent years poses a serious security threat to computer networks.

Malware writers often obfuscate malware to hinder malware scanners from malicious code detection, i.e., to hide the fact that the software is actually malicious. Packing is the most common obfuscation method used by malware writers. Recently, there has been a dramatic increase in the number of new packers and variants of existing ones. Moreover, packers are employing increasingly sophisticated anti-unpacker tricks and obfuscation methods.

Identifying a packer and obtaining a sample of unpacked malware are important to AV (Anti-virus) researchers who work on updating antivirus software to defend against malware, so that they can perform in-depth analysis. However, packer analysis is a technically intense research task, requiring the AV experts' deep knowledge of hardware, operating systems, compilers and programming languages. The significant growth of packers, in both number and complexity, prevents AV researchers from carrying out their daily AV research work efficiently and effectively.

This PhD project has investigated the common features of packers and presented a novel, fast yet effective packer analysis framework called REFORM (Reverse Engineering For Obfuscation ReMoval). The system applies various technologies including reverse engineering, compression algorithms and statistical methods to de-obfuscate packers.

REFORM is comprised of three major components that solve the problem of automatic packer analysis at three important stages of the packer analysis life cycle, namely packer detection, packer identification and unpacking, respectively: (1) It incorporates a novel randomness test that preserves local detail in the packer. This makes it easy for an AV researcher to distinguish areas of compressed/encrypted data from other code and data.

(2) Using the above randomness test, each packer is seen to exhibit a unique pattern in its randomness distribution. The REFORM framework therefore provides an extremely effective packer classification model based on a set of randomness measurements generated from a packed file. Various statistical classifiers have also been integrated in REFORM to achieve even better classification performance. (3) REFORM enables an efficient generic unpacking strategy which uses an ordered address execution histogram to capture the memory after the unpacking loop has executed.

We demonstrate REFORM's capability on speeding up packer detection, identification and unpacking procedures. Such an automatic system is shown in the thesis to be essential to keeping up with the accelerating growth in packed malware.

# Acknowledgements

I would like to take this opportunity to express my deepest gratitude to Associate Professor Serdar Boztaş (Senior supervisor), Professor Kathy Horadam (Co-supervisor) and Dr Tim Ebringer (consultant) for their helpful suggestions, guidance and support in planning and research of this thesis.

A deep and heart-felt thank to my husband John and kids Jeffrey and Michelle. Your love, support and sacrifice over the long period of my study has been invaluable to me. Also, I wish to address my appreciation to my parents and parents in law for their unwavering support and love.

This work would not have been possible without the support of the industrial partners, CA and HCL, in providing scholarship support, the necessary research equipment and data as well as other useful sources of information. I would particularly like to extend my appreciation to Dr Steven Versteeg as he has been a constant source of encouragement and support throughout my last stage research.

Through CA/HCL, I had the opportunity to talk with and listen to various AV experts. I thank Jakub Kaminski, Trevor Yann, Scott Molenkamp, Hamish O'Dea, Amir Fouda, Chun Feng and David Wo for their important contributions during discussion of this work. And I greatly appreciate the assistance from Mr Boris Lao, in kindly providing some data to my experiments.

I am grateful to staff members of the School of Mathematics and Geospatial Sciences at RMIT University, who helped me broaden my knowledge and provided an enjoyable working and research environment.

Finally, I would like to thank all the people who make it possible to complete this thesis.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

This thesis presents REFORM, an approach to malware packer analysis that aims to speed up packer analysis, detection and unpacking procedures. This opening chapter introduces the research problem, its significance, general approach, and scope.

## 1.1 Problem Background

The Internet has become an essential part of human life in the modern information society: we read on-line newspapers to get the latest news from around the world; we research and look for information using search engines; we shop online, arrange our finances via on-line banking; we work with colleagues through email and on-line conferences; we communicate with friends on on-line social networks; we listen to on-line music, watch on-line movies, play on-line games, etc.

While the Internet brings enormous convenience to users, malware, which includes computer viruses, worms, Trojan horses, other malicious or unwanted software, produces a tremendous impact on users' security, reliability and privacy. This is a serious threat to the security of computer networks. In recent years, the number of malware infections has risen exponentially. According to a recent report by Gostev [38, 39], Kaspersky Security Network (KSN) registered 73,619,767 attacks on their users in 2009 compared to 23,680,646 and 220,172 in 2008, 2007 respectively. To make things worse, most new malware is now designed for illicit financial gain, obtained through the theft of personal information and confidential data from the victims. This confidential data typically includes bank accounts and/or online game related information. According to the 2008 CSI annual computer crime and security surveys, malware is now the most common form of attack in computer crime, overtaking those due to insider abuse, unauthorized access and denial of service [95].

Many efforts have been made to combat malware. For every malware binary, antivirus (AV) researchers need to reverse engineer the malware and update the AV scanner to detect and counter it. Unfortunately, malware authors are aware of this. Most new malware implements various obfuscation techniques in order to disguise themselves, and therefore preventing successful analysis and thwarting detection by AV scanners. The obfuscation techniques which are informally discussed in this section will be formally defined later on in this thesis.

Packing is a favorite obfuscation technology used by malware. It has been reported that the majority of malware in the WildList, (the list of current in-the-wild viruses [50]), is runtime-packed [10, 13, 75]. Packing malware has several benefits for the attacker. Firstly, it usually reduces its file size, thus allowing fast transfer over the network. Secondly, it increases the AV scanner's scanning time as runtime packers require additional work from the scanner, such as checking the file format and the code, performing the unpacking, etc. Thirdly, it is more resistant to AV scanners since the appearance of the malicious code has been changed due to the compression and/or encryption the packer employs. Lastly, the packer's complexity can be enhanced almost without limit by applying various new armoring techniques as described in Section 2.3.1.2. The quick analysis and identification of a complex packer is a very challenging task for the AV researchers.

In recent years, successful packer identification and analysis by AV analysts have been confounded by counter-counter attacks by packer and malware developers. An increasing number of anti-unpacking tricks and obfuscation methods are now being applied to packers. The target packers include both existing common packers and the newly emerging ones, with the aim of providing a hard shell for malware. As a result, the volume and the sophistication of new packer strains and variants are dramatically increasing. This makes it difficult, costly and time-consuming for AV researchers to carry out the traditional static packer identification and classification methods which are mainly based on the packer's signature.

Furthermore, simple packer detection isn't enough for identifying a sample as harmful. This is because that packers are not only used by malware. A lot of commercial software applies packer techniques for legitimate purposes. As a result, all detected packed executables need to be unpacked before being passed into further malware analysis. Obtaining the original unpacked file is important to AV researchers since it enables them to undertake in-depth analysis such as

1. malware detection;
2. malware classification;
3. static malware code analysis;
4. common malware features identification; and
5. determination of malware payload trigger conditions, e.g., under what conditions and on what particular platform will the malware pabe executed?

However, unpacking is a technically intense research task, requiring the AV experts' deep knowledge based on hardware, operating systems, networking, programming and security. As packers are increasing in both number and complexity, it is crucial to develop an automatic unpacking strategy.

In this doctoral thesis, the focus of our research is the development of an automatic packer analysis strategy without the need to manually reverse engineer every instance of a packed file. Our aim is to find the common features of packers, use techniques from statistics and information theory to abstract these features and determine techniques for identifying these features, and therefore address the problem of automatic packer analysis at three important stages of the packer analysis life cycle, namely packer detection, packer identification and unpacking.

## 1.2 The REFORM Framework – An Outline

This thesis will investigate the common features of packers and present REFORM, a new fast and effective packer analysis framework. This is designed to speed up packer analysis, detection and unpacking procedures. To date, there has been substantial work in automated packer analysis [9, 40, 45, 53, 70, 92, 97, 106, 109, 110]. REFORM presents a unique strategy that uses information theory and statistical methods. It also introduces a novel automatic packer classification system.

As presented in Figure 1.1, REFORM is comprised of three major techniques which are used for packer detection, packer classification and unpacking respectively:

- REFORM presents a novel fast randomness test algorithm that preserves local detail of the packer. A plot of the test result on an entire file provides an effective aid to find packed data. As shown in Figure 1.2, the graph of a packer's randomness distribution shows areas of relatively high and low randomness across the file. High, flat sections of the plot indicate highly random data, often indicating compressed or encrypted data, while low, varying sections of the plot tend to be produced by code. This test is also useful for animating unpacking which allows AV research to easily gain a feel about how a packer works.

- REFORM proposes an automatic packer classification system that identifies packers effectively and efficiently. To the best of our knowledge, **this is the first published packer classification system that achieves high accuracy on real malware**.

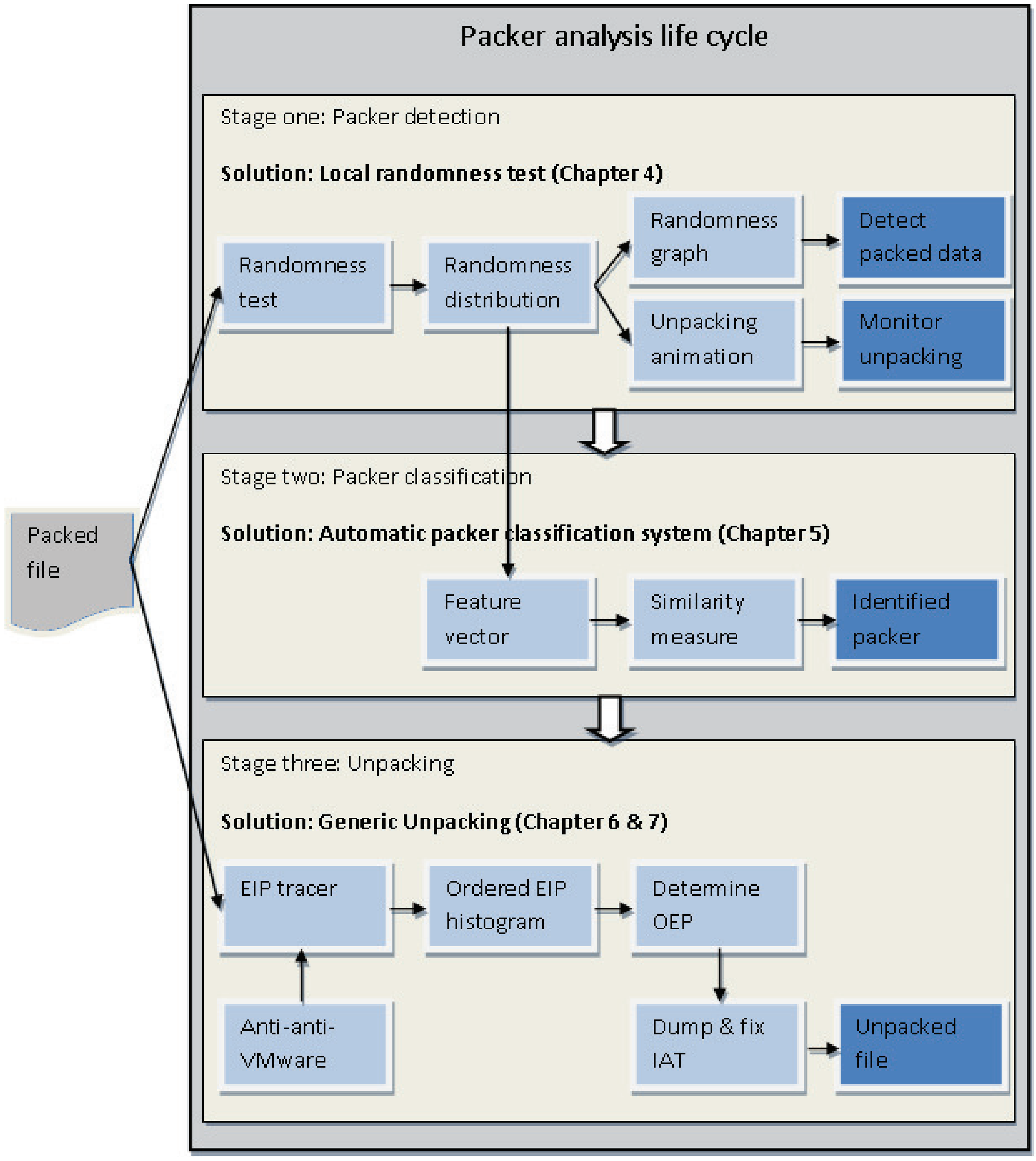


**Figure 1.1:** The REFORM malware packer analysis framework. Here, EIP is the instruction pointer, OEP is the original entry point of the file and IAT refers to the import address table. The detailed description of these terminologies will be discussed in Chapter 2.

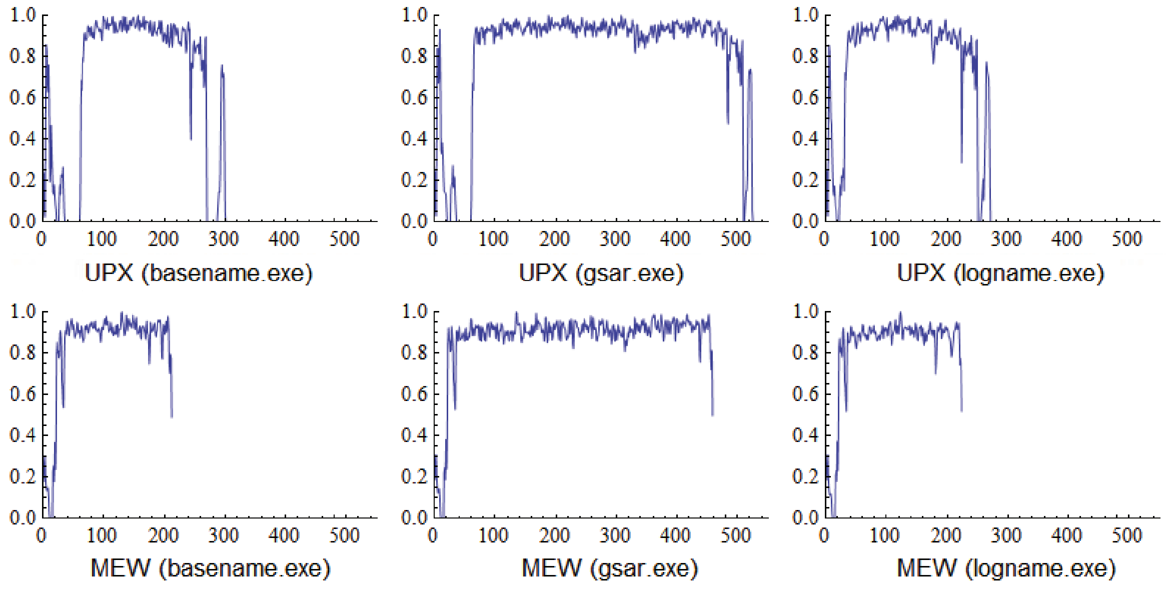


**Figure 1.2:** Example randomness test plots of packer UPX and MEW packed executable files

It is observed that each packer produces a distinctive pattern in its randomness distribution plot (refer to Figure 1.2). This unique feature of the packer is generated very quickly and therefore is selected and extracted in REFORM for packer classification. We refine the randomness test algorithms and conduct experiments to give guidance for optimal parameter settings. Various statistical classifiers have also been integrated into REFORM to achieve high classification performance.

- REFORM presents a new and efficient generic unpacking algorithm which effectively locates the boundary area between the unpacking activity and the execution of the original executable file. With REFORM, this unpacking solution can be applied directly on a packed file without identifying its family and finding corresponding unpacking routine. The algorithm firstly creates a histogram of the addresses of executed instructions and then orders it by the last time the address is executed. The histograms for some packers, on both linear and log scales (see Figure 1.3), clearly illustrate that the unpacking finishes after a massive recursive execution (i.e., a "hump") in the graph. This gives a good opportunity to dump the memory and retrieve the unpacked binary. The technique is extremely efficient to implement. To improve its performance, a new automatic anti-anti-VMware (The VMware Workstation virtualization software) technique is incorporated into REFORM.

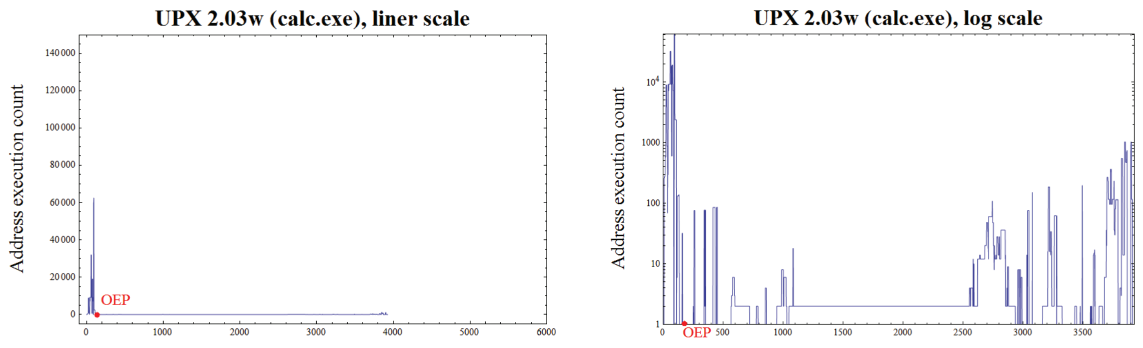


**Figure 1.3:** Example ordered address execution histogram, packer UPX packed calc.exe file

## 1.3 Structure of the Thesis

The structure of the thesis is as follows:

Chapter 2 and 3 constitute a literature review of the relevant background fields necessary to understanding the topic as a whole. The fundamental background to packers is introduced initially in Chapter 2. Following this is a summary of the current state of the research on packer analysis and unpacking. The strengths and weaknesses of each approach are analysed. Chapter 3 gives the mathematical background of this research. It consists of a brief introduction to the randomness test and the statistical classification algorithm and their application.

Chapter 4 investigates the features of packers and presents a novel randomness test. The test is fast and preserves local detail by showing areas of high randomness and low randomness data across the entire file. The results of the test in Chapter 4 suggest that a kind of "signature" of the packer can be developed based on a randomness signal. This is utilised in later chapters in the packer classification system that is developed in this thesis.

Chapter 5 introduces an automatic packer classification system using packer's randomness profile. Similarity measures and statistical classification algorithms are both tested as means of achieving highly efficient and effective performance.

Chapter 6 addresses the problem of unpacking and introduces a novel generic unpacking strategy. In this strategy, the OEP can be precisely located using a statistical approach.

Chapter 7 is devoted to improving unpacking performance. Increasing numbers of malware use anti-VMware techniques that attempt to detect if the malware is being executed in VMware. This chapter firstly discusses two main types of anti-VMware tricks and presents a dynamic scanning and patching technique that thwarts VMware detection. This strategy will be an essential component of packer analysis systems.

Finally, Chapter 8 summaries the REFORM framework, indicates the contributions of the thesis and puts the study in perspective, within the larger context of malware and computer security. Recommendations for future research are also outlined.

## 1.4 Publications

The work performed in this thesis has resulted in the following publications:

1. The randomness test described in Chapter 4 and the randomness signature based packer classification system introduced in Chapter 5 have been published and presented:

   Tim Ebringer, Li Sun, and Serdar Boztaş, "A fast randomness test that preserves local detail", *Proceedings of the 18th Virus Bulletin International Conference*, pages 34–42, Oct 2008.

2. The packer classification system using pattern recognition techniques introduced in Chapter 5 has been accepted for publication in Springer's Lecture Notes in Computer Science series:

Li Sun, Steven Versteeg, Serdar Boztaş and Trevor Yann, "Pattern Recognition Techniques for the Classification of Malware Packers", *15th Australasian Conference on Information Security and Privacy (ACISP)*, Jul 2010.

3. The generic unpacking strategy described in the Chapter 6 has been presented:
   Li Sun, Tim Ebringer, and Serdar Boztaş, "Hump and dump: Efficient generic unpacking using an ordered address execution histogram", *2nd CARO Workshop*, http://www.datasecurity-event.com/uploads/hump_dump.pdf, May 2008.

4. The Anti-anti-VMware technique introduced in Chapter 7 has been published and presented:
   Li Sun, Tim Ebringer, and Serdar Boztaş, "An automatic anti-anti-vmware technique applicable for multi-stage packed malware", *Proceedings of the 3rd International Conference on Malicious and Unwanted Software (Malware)*, pages 19–25, Oct 2008.

# Chapter 2

# General AV Concepts

This chapter provides fundamental background knowledge in the area of malware packer analysis. Section 2.1 briefly introduces malware and its obfuscation techniques. Section 2.2 describes Portable Executable (PE) format, the main file format of malware and packers discussed in this thesis. Section 2.3 discusses how the packer works and its evolution. Section 2.4 and Section 2.5 give a general overview of packer analysis, including detection, identification and unpacking. At the end, Section 2.6 and Section 2.7 introduce the packer analysis environment VMware and the main packer analysis tool IDA pro 5.1 used in this thesis.

## 2.1 Malware and Its Obfuscation Techniques

Malware is a catch-all term for any kind of *Mal*icious soft*ware* including computer viruses, internet worms, Trojan horses, exploits, droppers, rootkits, etc. The software is malicious if it exhibits unanticipated or undesired effects, caused by an agent intent on damage to a single computer, server, or computer network [84].

Malware employs a wide variety of code obfuscation techniques in order to hinder reverse-engineering. As well as slowing initial analysis, obfuscation serves to make simple variants of existing malware time-consuming and difficult to recognize. Examples of some effective obfuscation techniques employed are:

- *Entry-point obscuring (EPO)*: A technique for infecting programs through which a virus tries to hide its entry point in order to avoid detection. Instead of taking control and carrying out its actions as soon as the program is used or run, the virus allows it to work correctly for a while before the virus goes into action.
- *Polymorphism and Metamorphism*: A polymorphic virus encrypts its code differently with each infection, or generation of infections. Therefore, every instance has a different decryptor. A string-based virus scanner would not be able to reliably identify all variants of this sort of virus. However, once inside the decrypted virus body, polymorphic viruses are static. Thus the string-based scanning can be applied at a later stage, e.g., after decryption. A more sophisticated obfuscation is metamorphism. In this technique, the code body of the virus payload is changed for every new generation. Therefore, scan string and range scanning are no use as detection techniques.
- *Encryption /Packing*: Files, or groups of files, are encrypted/compressed into another file so that they take up less space and hinder signature based malware detection.

The focus of this thesis is packing since it is the most common obfuscation technique. Moreover, the sophisticated packing tool (packer) also employs various obfuscation techniques to increase the chance for a successful hiding.

## 2.2 PE File Format

The malware world moves very fast. Several years ago, most malware existed as Office macro viruses. Today most malware exists as Windows Portable Executable (PE) binaries. PE file format is a file format for executables, object code and dynamic link library (DLLs) used in 32 bit and 64 bit versions of Windows operating systems. In this thesis, all clean data and malware samples used for experiments are in Win32 PE format.

To fully understand the malware structure and its behavior, it is better to first study the PE format. Here, some important concepts of the PE files that are closely related to this thesis are discussed. An executable PE file calc.exe (the Windows Calculator) will be used to clarify the context. For further information, a detailed file format can be seen in Figure 2.1 and a full description of the format can be found in the Microsoft technical paper [85].

### 2.2.1 Load and Execute Process

When a computer user runs a PE program in Windows, the Windows operating system firstly examines the PE file and determines its preferred base address and required size. If the memory address space range from the preferred base address to the base address plus the image size is available, the file is loaded at the preferred base address and the image of the file is mapped to the CPU's physical memory address. This mapping is consistent in that higher offsets in the file correspond to high memory addresses when mapped into the memory. The data structures on disk are the same data structures used in memory, though the offset of an item in the disk file may differ from its offset once loaded into memory. If the PE file's preferred base address has already take by something else, the operating system will rebase the PE file. The system loads the file somewhere else in memory and applies fixups to its address.

Once the program is loaded into the memory, Windows executes a branching instruction that causes the CPU to begin execution of the program's first machine instruction (called *entry point*). As soon as the program begins running, it is called a *process*. Windows assigns the process an identification number (*process ID*), which is used to keep track of it while running. The process runs by itself. It is the operating system's job to track the execution of the process and to respond to requests for system resources, such

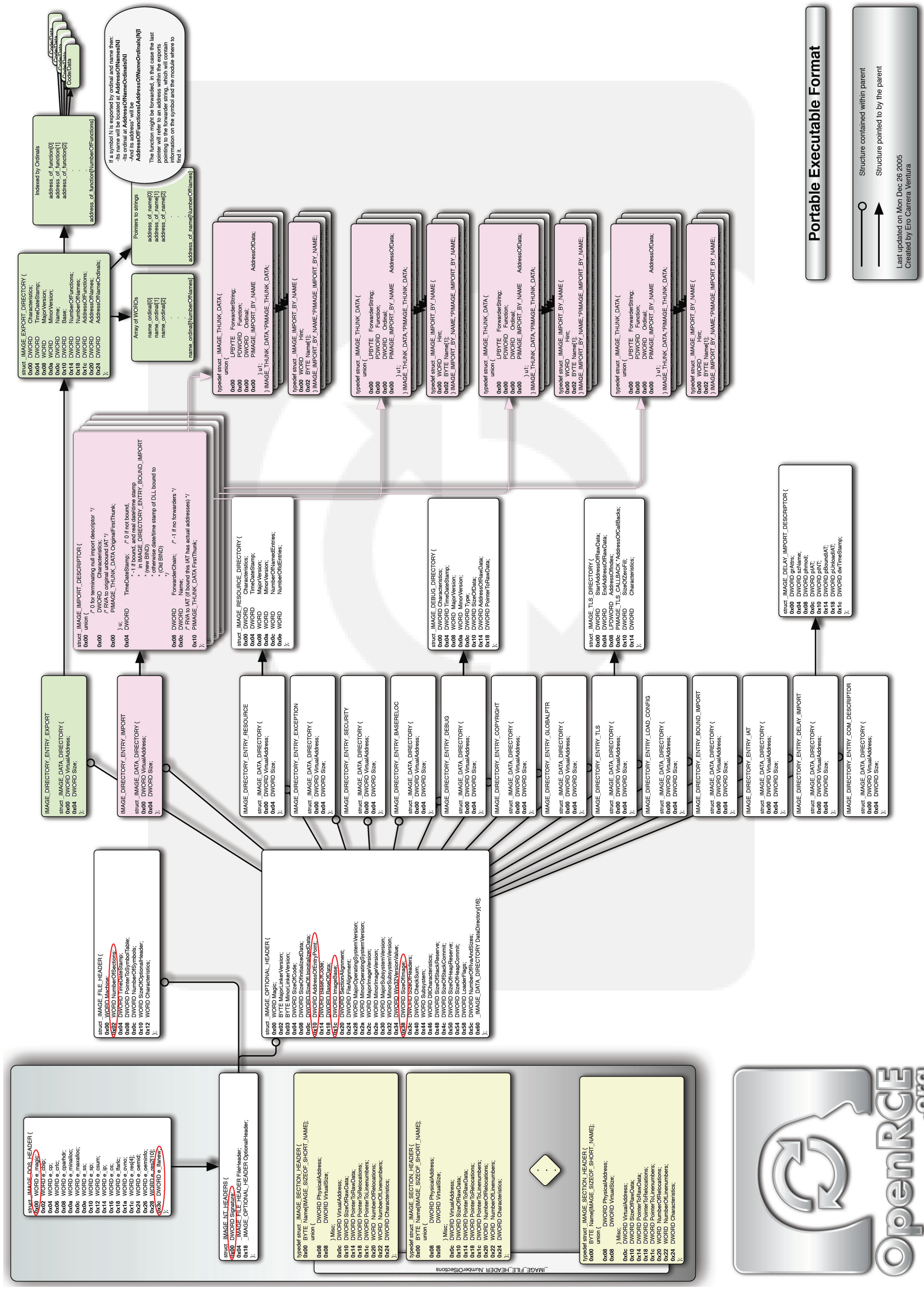

Portable Executable Format
Structure contained within parent
Structure pointed to by the parent
Last updated on Mon Dec 26 2005
Created by Ero Carrera Ventura
OpenRCE.org
IMAGE_DIRECTORY_ENTRY_EXPORT
IMAGE_DIRECTORY_ENTRY_IMPORT
IMAGE_DIRECTORY_ENTRY_RESOURCE
IMAGE_DIRECTORY_ENTRY_EXCEPTION
IMAGE_DIRECTORY_ENTRY_SECURITY
IMAGE_DIRECTORY_ENTRY_BASERELOC
IMAGE_DIRECTORY_ENTRY_DEBUG
IMAGE_DIRECTORY_ENTRY_COPYRIGHT
IMAGE_DIRECTORY_ENTRY_GLOBALPTR
IMAGE_DIRECTORY_ENTRY_TLS
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT
IMAGE_DIRECTORY_ENTRY_IAT
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR
IMAGE_FILE_HEADER.NumberOfSections

**Figure 2.1:** A detailed PE format, extracted from openrce website [80].

as memory, disk files and input-output devices.

When the process ends, its handle is removed and the memory it used is released so it can be used by other programs.

### 2.2.2 PE File Structure

As shown in Figure 2.2, a PE file consists of a number of headers and sections, which together tell the dynamic linker how to map the file into memory.

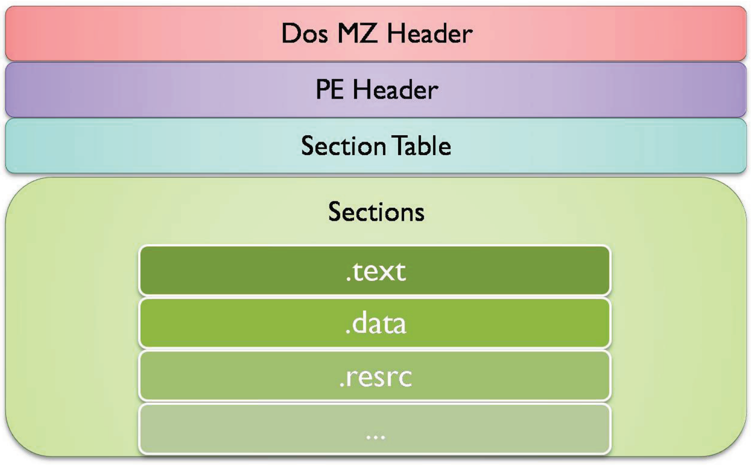


**Figure 2.2:** PE file structure

#### 2.2.2.1 MS-DOS Header

The very beginning of the PE file presents the MS-DOS header which includes a MS-DOS MZ header followed by a MS-DOS stub. The DOS stub is an MS-DOS executable program. When executed in a non-windows operating system, it displays an error message such as "This program cannot be run in MS-DOS mode".

The traditional MS-DOS MZ header, defined as an IMAGE_DOS_HEADER structure, contains two important fields, namely *e_magic* and *e_lfanew*. The *e_magic* field locates at the start of the mapping address (i.e., the *image base*). For a valid PE file, the *e_magic* must be set to the value of '5A4Dh' (the assembly code of 'MZ'). This is the #define value of IMAGE_DOS_SIGNATURE. That is, the DOS signature of a PE file is 'MZ'.

The *e_lfanew* field locates at offset 3Ch and provides the file offset of the PE header (see section 2.2.2.2) in the memory, so that the loader can pick up the value and locates the PE header to this address.

Figure 2.3 displays the MS-DOS header of calc.exe executable when opens it in a Hex viewer. As shown, the first word 'MZ' is the value of *e_magic*. The memory at offset 3Ch (*e_lfanew*) is "F0000000". Intel processors store and retrieve data from memory using *little endian* order, that is, the least significant byte is stored at the first memory address allocated for the data and the remaining bytes are stored in the next consecutive memory positions. Therefore, the double word stored at *e_lfanew* equals 000000F0h. This means the PE header locates at the offset 000000F0h from the image base in the memory.

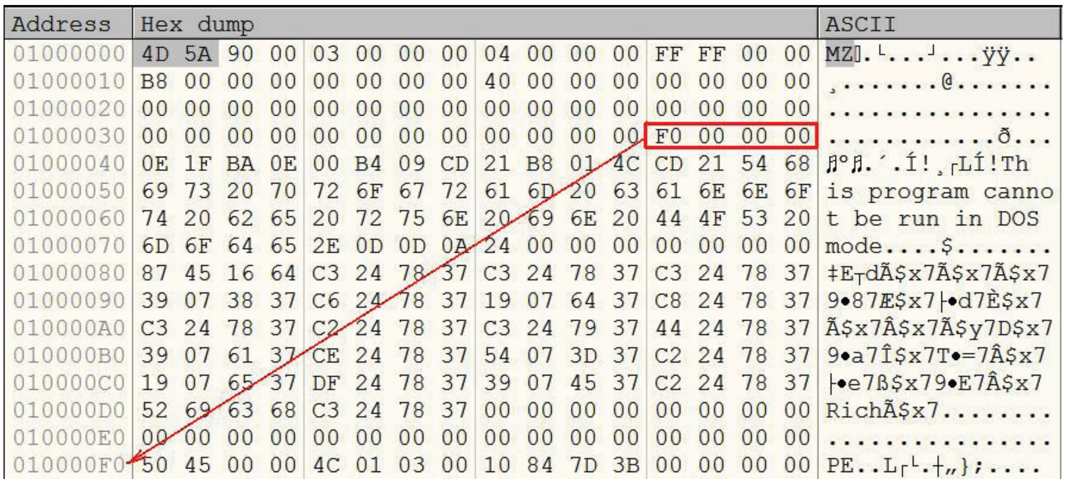

```
Address   Hex dump                                         ASCII
01000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ.˪...˩...ÿÿ..
01000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
01000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01000030  00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00  ............ð...
01000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ♬°♬.´.Í!¸┌LÍ!Th
01000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
01000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
01000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
01000080  87 45 16 64 C3 24 78 37 C3 24 78 37 C3 24 78 37  ‡E┬dÃ$x7Ã$x7Ã$x7
01000090  39 07 38 37 C6 24 78 37 19 07 64 37 C8 24 78 37  9•87Æ$x7├•d7È$x7
010000A0  C3 24 78 37 C2 24 78 37 C3 24 79 37 44 24 78 37  Ã$x7Â$x7Ã$y7D$x7
010000B0  39 07 61 37 CE 24 78 37 54 07 3D 37 C2 24 78 37  9•a7Î$x7T•=7Â$x7
010000C0  19 07 65 37 DF 24 78 37 39 07 45 37 C2 24 78 37  ├•e7ß$x79•E7Â$x7
010000D0  52 69 63 68 C3 24 78 37 00 00 00 00 00 00 00 00  RichÃ$x7........
010000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
010000F0  50 45 00 00 4C 01 03 00 10 84 7D 3B 00 00 00 00  PE..L┌˪.┼„};....
```

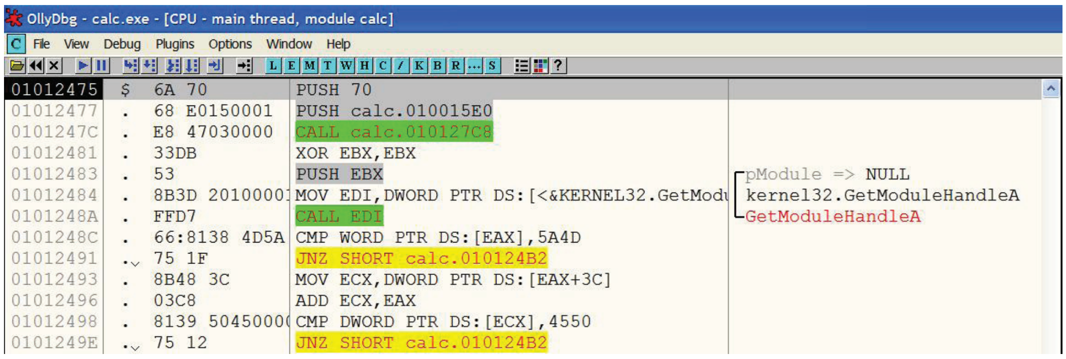
**Figure 2.3:** Example MS-DOS header: MS-DOS header of calc.exe in a Hex viewer.

### 2.2.2.2 PE Header

Immediately following the DOS stub in the PE file is the PE header. Every PE header starts with a signature field (see Figure 2.1), namely IMAGE_NT_SIGNATURE. The specific PE signature value is "PE\0\0" . This signature together with the DOS signature ('MZ') mentioned above are always used to check whether an executable file is a PE file.

The PE signature is followed by two structures, IMAGE_FILE_HEADER and IMAGE_OPTIONAL_HEADER. These two structures encapsulate the information necessary for the Windows operating system loader to manage the wrapped executable code. Some important values within structures are listed below and their addresses (circled in red) can be found in Figure 2.1:

- *NumberOfSections*: The number of sections in the executable file.
- *SizeOfOptionsHeader*: The size of the IMAGE_OPTIONAL_HEADER structure.
- *AddressOfEntryPoint*: The address of the entry point (EP). This is the address where the program starts execution. As shown in Figure 2.4, when calc.exe is loaded by the debugger OllyDBG, the debugger stops at the line 01012475 which is the EP of the calc.exe.

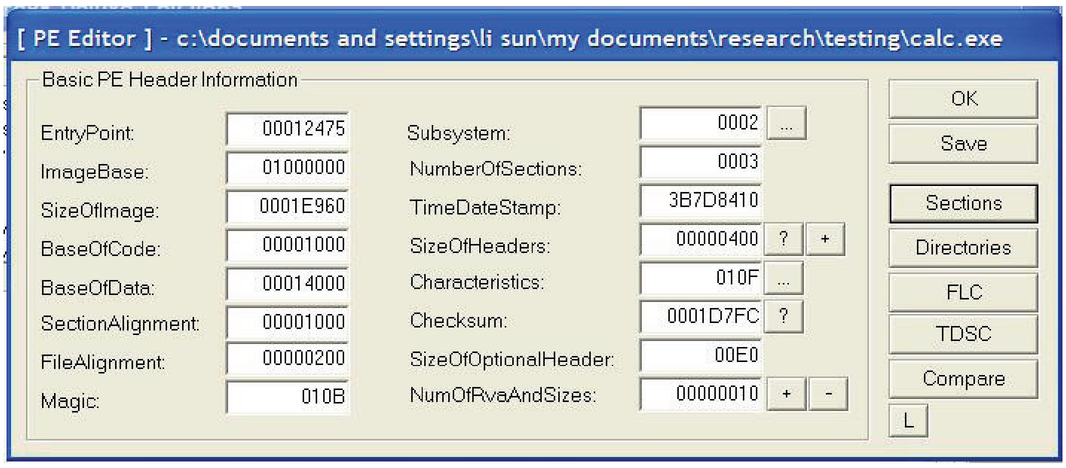

OllyDbg - calc.exe - [CPU - main thread, module calc]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S ?

```
01012475  $   6A 70          PUSH 70
01012477  .   68 E0150001    PUSH calc.010015E0
0101247C  .   E8 47030000    CALL calc.010127C8
01012481  .   33DB           XOR EBX,EBX
01012483  .   53             PUSH EBX                                   ┌pModule => NULL
01012484  .   8B3D 20100001  MOV EDI,DWORD PTR DS:[<&KERNEL32.GetModu   │kernel32.GetModuleHandleA
0101248A  .   FFD7           CALL EDI                                   └GetModuleHandleA
0101248C  .   66:8138 4D5A   CMP WORD PTR DS:[EAX],5A4D
01012491  .v  75 1F          JNZ SHORT calc.010124B2
01012493  .   8B48 3C        MOV ECX,DWORD PTR DS:[EAX+3C]
01012496  .   03C8           ADD ECX,EAX
01012498  .   8139 5045000(  CMP DWORD PTR DS:[ECX],4550
0101249E  .v  75 12          JNZ SHORT calc.010124B2
```

**Figure 2.4:** Example PE entry point: OllyDBG loads calc.exe and stops at its entry point.

- *ImageBase*: The preferred address when mapping into the memory. For EXE, the default ImageBase is 400000h.
- *SizeOfImage*: The amount of memory that the system needs to reserve when loading the file into the memory. This covers from the start of the ImageBase to the end of the last section and is rounded up to a multiple of the section alignment.
- *DataDirectory*: An array of data structures (IMAGE_DATA_DIRECTORY ) containing export table (colored in green in Figure 2.1), import table (colored in pink in Figure 2.1), etc.

Some PE editors are available to edit/view different parts of PE, such as LordPE, Peditor and Stud-PE. Figure 2.5 presents some important PE header fields of calc.exe in LordPE.


[ PE Editor ] - c:\documents and settings\li sun\my documents\research\testing\calc.exe

Basic PE Header Information

| | | | |
|---|---|---|---|
| EntryPoint: | 00012475 | Subsystem: | 0002 |
| ImageBase: | 01000000 | NumberOfSections: | 0003 |
| SizeOfImage: | 0001E960 | TimeDateStamp: | 3B7D8410 |
| BaseOfCode: | 00001000 | SizeOfHeaders: | 00000400 |
| BaseOfData: | 00014000 | Characteristics: | 010F |
| SectionAlignment: | 00001000 | Checksum: | 0001D7FC |
| FileAlignment: | 00000200 | SizeOfOptionalHeader: | 00E0 |
| Magic: | 010B | NumOfRvaAndSizes: | 00000010 |

**Figure 2.5:** Example PE header: view some basic PE header information of calc.exe in LordPE.

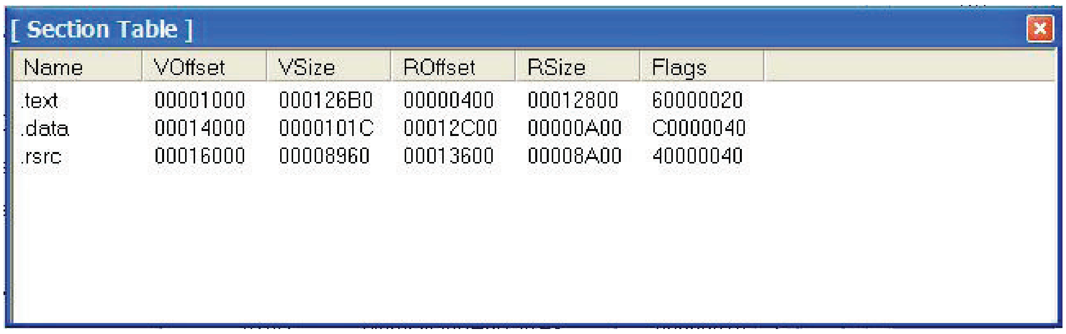
### 2.2.2.3 Section Tables

The section table lists all sections that exist in the application. For each section, an IMAGE_SECTION_HEADER structure is used to state the section name, its physical and virtual locations, actual and raw data size, characteristics, etc.

Normally each section has a distinctive name to represent different functionality. The section name can be user defined, e.g. UPX1 and UPX2 are used in a UPX packer. However, the most common and widely recognized names used by Windows are:

- *.text* section places all executable code;
- *.data* section stores all initialized data;
- *.rsrc* section contains the resources for the application;
- *.idata* section has the import table of the PE file.

The total number of section and its corresponding IMAGE_SECTION_HEADER structure is given by the value of NumberOfSections field in the PE header. Figure 2.6 listed information of calc.exe's three sections, namely *.text*, *.data* and *.rsrc* section respectively.



| Name | VOffset | VSize | ROffset | RSize | Flags |
|---|---|---|---|---|---|
| .text | 00001000 | 000126B0 | 00000400 | 00012800 | 60000020 |
| .data | 00014000 | 0000101C | 00012C00 | 00000A00 | C0000040 |
| .rsrc | 00016000 | 00008960 | 00013600 | 00008A00 | 40000040 |

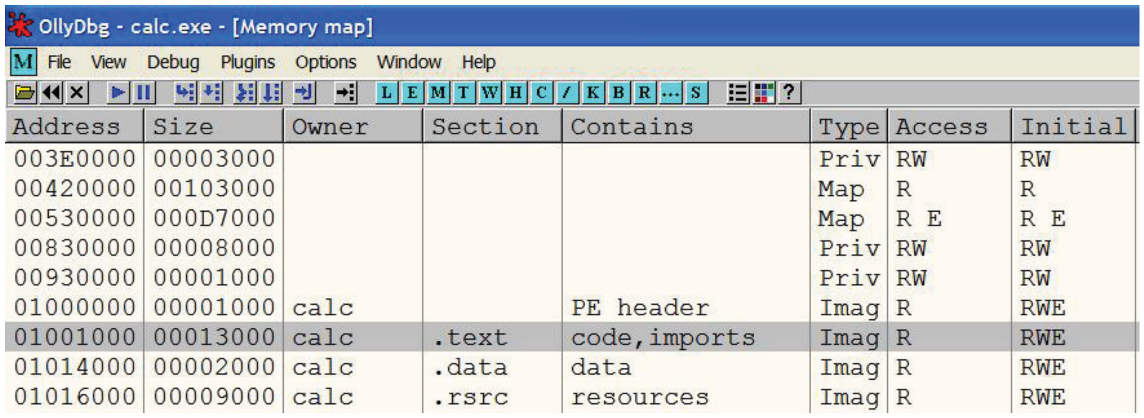
**Figure 2.6:** Example PE section table: view section table of calc.exe in LordPE.

#### 2.2.2.4 Sections

The last part of the PE file is the raw data. These data are separated into sections in the location described in the section table. For instance, in Figure 2.6, the `VOffset` of the *.text* section is 00001000h. So, once mapped into memory, the location of *.text* section of calc.exe will be

$$\begin{aligned}\texttt{Virtual address (VA)} &= \texttt{Base address + VOffset}\\ &= \texttt{01000000h + 00001000h}\\ &= \texttt{01001000h}\end{aligned}$$

Here, the `VSize` of the *.text* section is 000126B0h. However, as shown in Figure 2.7, its section size in the memory is 00013000h. This is because that the start of each section must be aligned to a page boundary. On x86 CPUs, pages are 4kb (1000h) aligned, so the section size is automatically adjusted to a multiple of 1000h.


OllyDbg - calc.exe - [Memory map]
M File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S ?

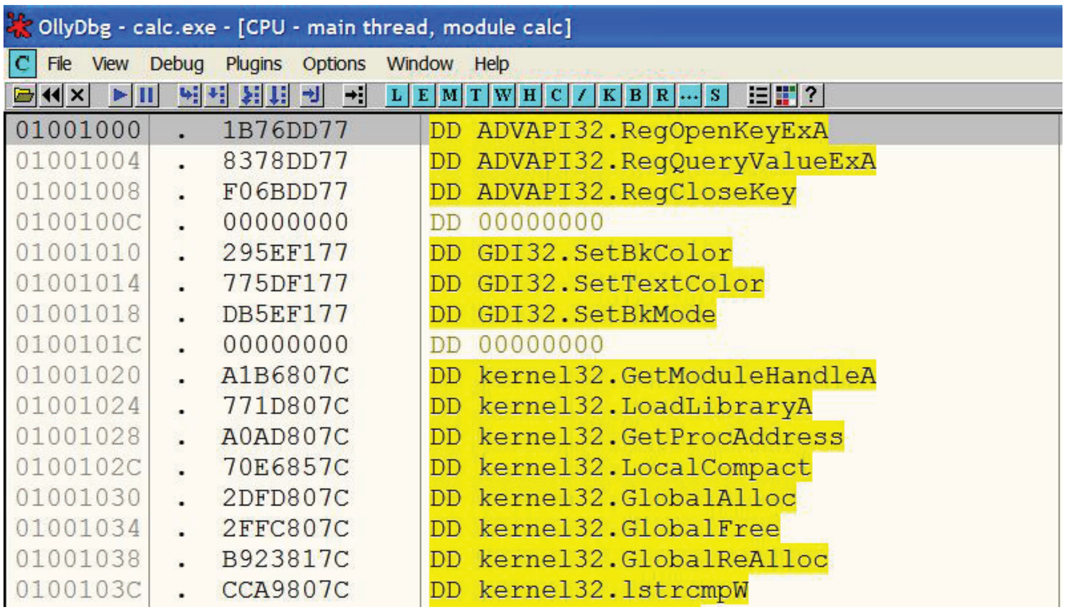
| Address | Size | Owner | Section | Contains | Type | Access | Initial |
|---|---|---|---|---|---|---|---|
| 003E0000 | 00003000 | | | | Priv | RW | RW |
| 00420000 | 00103000 | | | | Map | R | R |
| 00530000 | 000D7000 | | | | Map | R E | R E |
| 00830000 | 00008000 | | | | Priv | RW | RW |
| 00930000 | 00001000 | | | | Priv | RW | RW |
| 01000000 | 00001000 | calc | | PE header | Imag | R | RWE |
| 01001000 | 00013000 | calc | .text | code,imports | Imag | R | RWE |
| 01014000 | 00002000 | calc | .data | data | Imag | R | RWE |
| 01016000 | 00009000 | calc | .rsrc | resources | Imag | R | RWE |

**Figure 2.7:** Example PE memory map: view memory map of calc.exe in OllyDBG.

### 2.2.3 Import Address Table (IAT)

One very important part of the PE file is the IAT. This is a table of the external functions that an application wants to use. For each import function, its location in memory is listed in the table. This location is dynamic. When the executable is compiled, and the import table is built, the compiler and the linker do not know where in memory the particular DLL will be. So at that time, an IAT contains NULL memory pointers to each function. At the beginning of an application's execution, Windows finds its IAT location (from the PE header) and overwrites all NULL entries with the correct memory location for each function.

Then IAT serves like a lookup table. Whenever an external function is called during the execution, a pointer to the value in the Import Address table is called. For example, when calc.exe is running in OllyDBG (Figure 2.8), the corresponding value in the IAT for the first function, RegOpenKeyExA, is filled with "1B76DD77" . Then once the function RegOpenKeyExA is called in calc.exe, the program will locate address "77DD761Bh" in the memory (see *little endian* order described in Section 2.2.2.1) and execute from there.


OllyDbg - calc.exe - [CPU - main thread, module calc]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S

```
01001000 .  1B76DD77   DD ADVAPI32.RegOpenKeyExA
01001004 .  8378DD77   DD ADVAPI32.RegQueryValueExA
01001008 .  F06BDD77   DD ADVAPI32.RegCloseKey
0100100C .  00000000   DD 00000000
01001010 .  295EF177   DD GDI32.SetBkColor
01001014 .  775DF177   DD GDI32.SetTextColor
01001018 .  DB5EF177   DD GDI32.SetBkMode
0100101C .  00000000   DD 00000000
01001020 .  A1B6807C   DD kernel32.GetModuleHandleA
01001024 .  771D807C   DD kernel32.LoadLibraryA
01001028 .  A0AD807C   DD kernel32.GetProcAddress
0100102C .  70E6857C   DD kernel32.LocalCompact
01001030 .  2DFD807C   DD kernel32.GlobalAlloc
01001034 .  2FFC807C   DD kernel32.GlobalFree
01001038 .  B923817C   DD kernel32.GlobalReAlloc
0100103C .  CCA9807C   DD kernel32.lstrcmpW
```

**Figure 2.8:** Example PE IAT: the partial set of IAT of calc.exe in OllyDBG.

## 2.3 Packing

Packing is the favorite obfuscation method used by malware authors to hinder both analysis and runtime detection of malware [10, 13, 75]. A packer is an executable program that takes an executable file or dynamic link library (DLL), compresses and/or encrypts its contents and then packs it into a new executable file (Figure 2.9).

Assuming the original executable file contains already known malicious code, the signature based AV scanner should be able to detect it. However, the appearance of the malicious code is changed after packing due to the compression and/or encryption. Therefore, the packed file will thwart malware detection as no signature match will be found. The analysis and detection of malware can only be undertaken after the file is unpacked. In this thesis, the packers discussed are in PE format.

A packed file contains two basic components. The first part is a number of data blocks which form the compressed and/or encrypted original executable file. The second part is an unpacking stub which can dynamically recover the original executable file on the fly. When the packed file is running, the unpacking stub executes firstly to unpack the

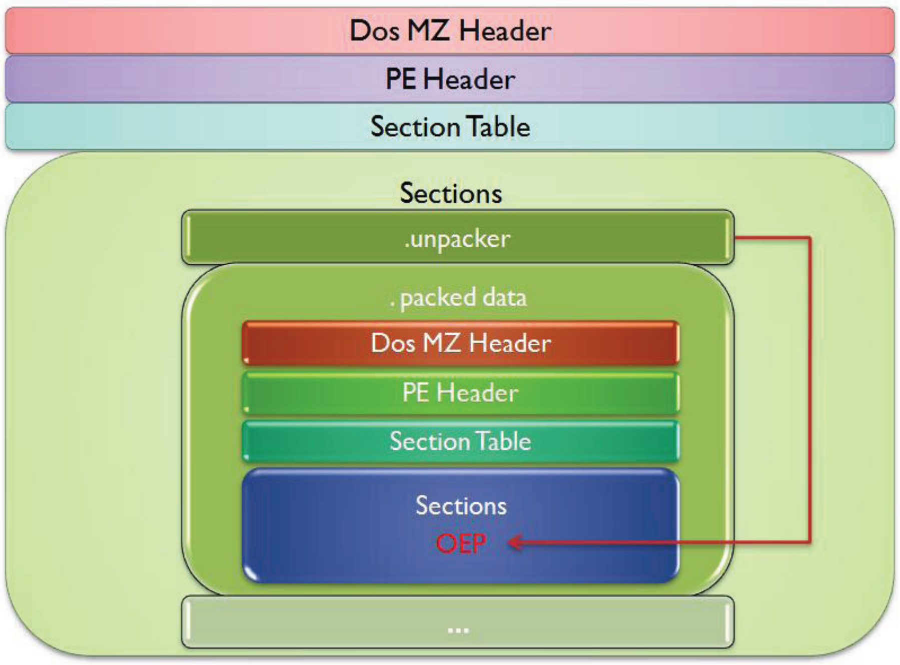


**Figure 2.9:** A PE Packer

original executable code and then transfers the control to the original file. The execution of the original file is mostly unchanged and starts from its original entry point (OEP) with no runtime performance penalties (after the unpacking has been completed.)

### 2.3.1 Packer Evolution

Recently, the packing techniques have undergone a rapid evolution. Packers can be classified into three generations based on their complexity.

#### 2.3.1.1 First Generation: Compressor

The first generation packer, compressor, is primarily designed to reduce the secondary storage requirements of the software through compression, thus reducing the time and bandwidth required to transfer the file on the net. Compressor usually contains little or no encryption. The compression engine used for compressor must perform at a fast decompression speed without deterioration in compression ratio. This is due to the fact that the packer needs to decompress the compressed code quickly during the unpacking so that the original packed file can run without delay. Some good candidates are aPLib, JCALG1 and LZMA.

Popular compressors include the Ultimate Packer for eXecutables (UPX), the Ultimate PE Packer (UPack), ASPack, PECompact, FSG, and MEW.

#### 2.3.1.2 Second Generation: Protector

In order to hide malicious payload from AV scanners, malware authors start to develop and use cryptors and protectors which pack the file through encryption and/or obfuscation instead of compression. Examples of common executable cryptors/protectors are: Yoda's Crypter, ASProtect, Armadillo, Morphine, EXECryptor, Obsidium, Enigma, and Themida.

It is common for compression and encryption to be combined in protectors. Moreover, anti-unpacking tricks continually evolve and are implemented quickly by malware writers in a range of protectors, from long-existing packers to modern new packers, with the goal of protecting malware. The main types of anti-unpacker tricks are listed below and most tricks have been described in Ferrie's papers [28, 29, 30, 31, 32].

- *Anti-dumping* tricks mainly alter the process memory of the running process to hinder further analysis on the dumped memory. The alteration is mainly performed on some useful information, such as PE header, imports, entry point codes, etc.

- *Anti-debugging* tricks prevent AV researchers from using a debugger easily. The debugger is the most common tool used to trace the execution of malware in action. Three popular types of debuggers are: 1. Kernel-mode debugger (SoftICE [21] and Syser [122]); 2. User-mode debugger (OllyDBG [123] and IDA— build-in debugger [47]); 3. User/kernel-mode debugger (WinDBG [22]).

- *Anti-emulating* tricks attack the software-based environment such as an emulator or a virtual machine, e.g. VMware [49]. Such an environment is essential to safely execute/monitor malicious behavior.

- *Anti-intercepting* tricks thwart page-level interception which stops the packer from executing newly written pages.

Furthermore, there are many other malware obfuscation techniques such as *junk code insertion*, *polymorphism*, *interrupt and exception handler control flow obfuscation*, etc. that are also widely applied on packers to improve its complexity. Some packers, such as tElock and Yoda's, apply *multistage unpacking* that decrypts the unpacking stub on the fly before its execution and encrypts it again just after the execution.

#### 2.3.1.3 Third Generation: Virtual Machine (VM) protection system

The new generation of packers, e.g., Themida [115] and VMProtect [108], extensively employ *virtual machine protector* which converts parts of the original code into its own bytecodes. When executed, it translates codes using its own virtual machine. As the meaning of bytecodes used in each VM protector are extremely time consuming and costly to determine, it is very difficult to analyse the protected program and impossible to precisely recover the original program.

## 2.4 Packer Detection and Identification

Packer analysis normally consists of three major tasks:

1. **Detection** Detecting if a PE packer is being used.
2. **Identification** Once a packer is detected, identify which packer is being used.
3. **Unpacking** Removing the PE packer's unpacking stub and restoring the original executable. The unpacking process will be explained in Section 2.5.

Packer detection and identification are important for an accurate and fast malware analysis. For a suspicious executable, the first thing to do is to distinguish whether it is packed or non-packed. Only non-packed executables are directly sent to malware scrutiny, and packed executables need to be unpacked firstly and then sent to malware inspection. The identification process may be distinct from detection, or identification may occur as a side effect of the detection method being used. Correct identification of the packer family and its version is crucial if a static unpacking strategy is used (see Section 2.5.2). Detection/identification techniques can be classified as static, dynamic or statistical. We discuss these categories below.

## 2.4.1 Static Detection

Static approaches are carried out without actually running any code. This section examines two static techniques: signature scanning and heuristic detection.

### 2.4.1.1 Signature Scanning

Signature-based packer scanners, such as PEiD [51] and pefile [15], use a signature database to determine if a binary contains packed code. A packer signature is typically a distinctive set of bytes which occur at the entry point or in sections in a PE file. Below are two examples extracted from PEiD's signature database (straight out of its UserDB.txt). For both signatures, `ep_only` is set to 'true', so the specified unique bytes only occur at the entry point, not in sections.

```
[UPX 0.50 - 0.70]
signature = 60 E8 00 00 00 00 58 83 E8 3D
ep_only = true

[UPX 0.72]
signature = 60 E8 00 00 00 00 83 CD FF 31 DB 5E
ep_only = true
```

In this approach, the incoming packer is checked against the database of the signatures for known packers. If there is an exact match, the program concludes that the corresponding packer is being used and the name of the packer is also identified.

Although these detectors are effective at detecting known packers, they fail to recognize new packers and carefully crafted variants of old packers. Besides, this approach is expensive as the signature detection and updating must be performed accurately by knowledgeable AV experts.

#### 2.4.1.2 Static Heuristic Detection

Some properties of PE packer such as file size, number/names of sections, and unusual imports may serve as good aids in detecting packers heuristically [25].

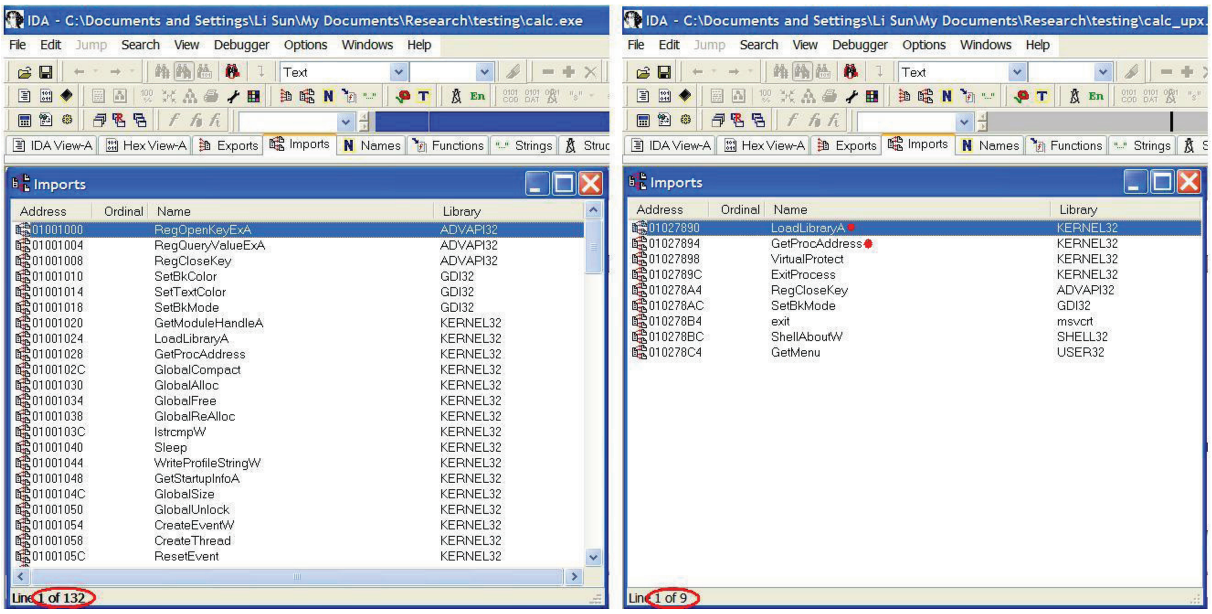
##### 2.4.1.2.1 Small File Size

Due to the compression and encryption applied to the packer, most packed files have small file size especially the code body size. This is due to the fact that the original 'code' body has been shrunk or replaced by packed 'data' inside the executable.

##### 2.4.1.2.2 Small Number of Imports

When a packed file is executed, Windows only loads the packed file's IAT but not the original file's IAT. The original file's IAT needs to be dynamically loaded during the unpacking. Therefore, if one checks the imports of the packed file, one will normally only find a small numbers of imports. As shown in Figure 2.10, a UPX packed file calc_upx.exe only contains 9 import functions while the original calc.exe file calls 132 import functions.

To dynamically rebuild the original IAT, a packed file normally contains two main import functions called "loadLibrary" and "getProAddr". These two functions will be repeatedly called by the unpack routine to get the memory location of the imports of the original executable.


IDA - C:\Documents and Settings\Li Sun\My Documents\Research\testing\calc.exe
File Edit Jump Search View Debugger Options Windows Help
IDA View-A Hex View-A Exports Imports Names Functions Strings Struc
Imports

| Address | Ordinal | Name | Library |
|---|---|---|---|
| 01001000 | | RegOpenKeyExA | ADVAPI32 |
| 01001004 | | RegQueryValueExA | ADVAPI32 |
| 01001008 | | RegCloseKey | ADVAPI32 |
| 01001010 | | SetBkColor | GDI32 |
| 01001014 | | SetTextColor | GDI32 |
| 01001018 | | SetBkMode | GDI32 |
| 01001020 | | GetModuleHandleA | KERNEL32 |
| 01001024 | | LoadLibraryA | KERNEL32 |
| 01001028 | | GetProcAddress | KERNEL32 |
| 0100102C | | GlobalCompact | KERNEL32 |
| 01001030 | | GlobalAlloc | KERNEL32 |
| 01001034 | | GlobalFree | KERNEL32 |
| 01001038 | | GlobalReAlloc | KERNEL32 |
| 0100103C | | lstrcmpW | KERNEL32 |
| 01001040 | | Sleep | KERNEL32 |
| 01001044 | | WriteProfileStringW | KERNEL32 |
| 01001048 | | GetStartupInfoA | KERNEL32 |
| 0100104C | | GlobalSize | KERNEL32 |
| 01001050 | | GlobalUnlock | KERNEL32 |
| 01001054 | | CreateEventW | KERNEL32 |
| 01001058 | | CreateThread | KERNEL32 |
| 0100105C | | ResetEvent | KERNEL32 |

Line 1 of 132

IDA - C:\Documents and Settings\Li Sun\My Documents\Research\testing\calc_upx.
File Edit Jump Search View Debugger Options Windows Help
IDA View-A Hex View-A Exports Imports Names Functions Strings
Imports

| Address | Ordinal | Name | Library |
|---|---|---|---|
| 01027890 | | LoadLibraryA ● | KERNEL32 |
| 01027894 | | GetProcAddress ● | KERNEL32 |
| 01027898 | | VirtualProtect | KERNEL32 |
| 0102789C | | ExitProcess | KERNEL32 |
| 010278A4 | | RegCloseKey | ADVAPI32 |
| 010278AC | | SetBkMode | GDI32 |
| 010278B4 | | exit | msvcrt |
| 010278BC | | ShellAboutW | SHELL32 |
| 010278C4 | | GetMenu | USER32 |

Line 1 of 9

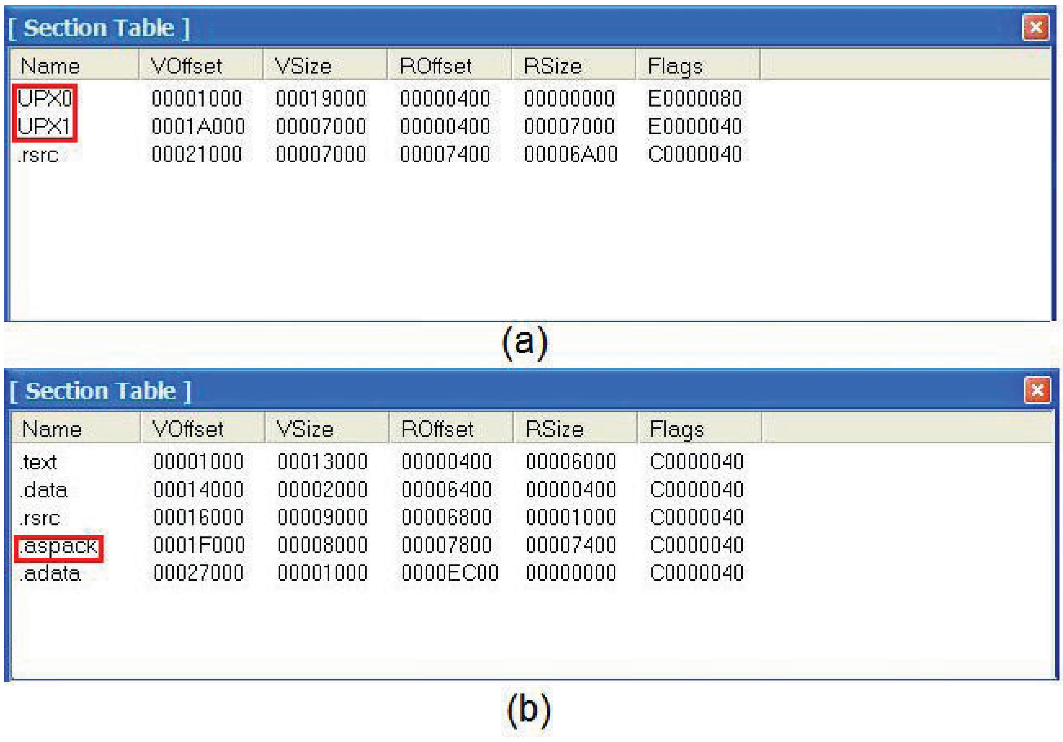
**Figure 2.10:** Comparison of import entries of packed file and its original file.

#### 2.4.1.2.3 Missing or Corrupted String Table

The string table is a table of commonly used strings in the application. A missing, corrupted or encrypted string table is usually an indicator that a PE packer has been used.

#### 2.4.1.2.4 Unique Section Names

As described in section 2.2.2.3, compilers/linkers normally have a standard naming convention for each code/data section. For example, Microsoft compiler uses .text while Borland compiler uses CODE for code area. Similarly, most packers have their own unique section names as packer authors like to stamp their identity on sections. Some good examples can be found in Figure 2.11.



| Name | VOffset | VSize | ROffset | RSize | Flags |
|---|---|---|---|---|---|
| UPX0 | 00001000 | 00019000 | 00000400 | 00000000 | E0000080 |
| UPX1 | 0001A000 | 00007000 | 00000400 | 00007000 | E0000040 |
| .rsrc | 00021000 | 00007000 | 00007400 | 00006A00 | C0000040 |

| Name | VOffset | VSize | ROffset | RSize | Flags |
|---|---|---|---|---|---|
| .text | 00001000 | 00013000 | 00000400 | 00006000 | C0000040 |
| .data | 00014000 | 00002000 | 00006400 | 00000400 | C0000040 |
| .rsrc | 00016000 | 00009000 | 00006800 | 00001000 | C0000040 |
| .aspack | 0001F000 | 00008000 | 00007800 | 00007400 | C0000040 |
| .adata | 00027000 | 00001000 | 0000EC00 | 00000000 | C0000040 |

**Figure 2.11:** Unique section names of packers: (a) Section table of UPX packed calc.exe, (b) Section table of Aspack packed calc.exe.

### 2.4.2 Dynamic Detection

Dynamic detection of the packer is achieved as follows. The code is run and its behavior is monitored to decide whether the file is packed or not. The execution is carried out in a safe environment, such as a virtual machine, to avoid possible malicious attack on the host machine.

The main heuristic used in dynamic detection is the packer's written-then-executed behavior. In a packed file, the original codes are compressed/encrypted. To be run, they must be decompressed/decrypted (unpacked) firstly. A unpacking stub usually unpacks the codes and stores them in the memory. When unpacking finishes, the code control is transferred to unpacked code so the newly created/modified memory will be executed.

### 2.4.3 Statistical Detection

There have been attempts to use statistical tests to distinguish between packed and unpacked files. These tests can be automatically carried out without any code analysis and therefore significantly reduce the cost. Two techniques, random byte frequency distribution and entropy analysis are discussed here.

#### 2.4.3.1 Random Byte Frequency Distribution

A commonly used metric to quickly identify packed and encrypted samples is to examine the byte frequency distribution of a sample file. The sorted byte frequency distribution in executable files is highly skewed. However, heavy use of compression or encryption intuitively should bring the bytes closer to a uniform distribution. Figure 2.12 shows a sorted byte distribution for the venerable Windows "calc.exe", the same program after being packed with UPX 2.03w, and random bytes. The intuition is that the flatter the slope, the more likely the executable has been packed.

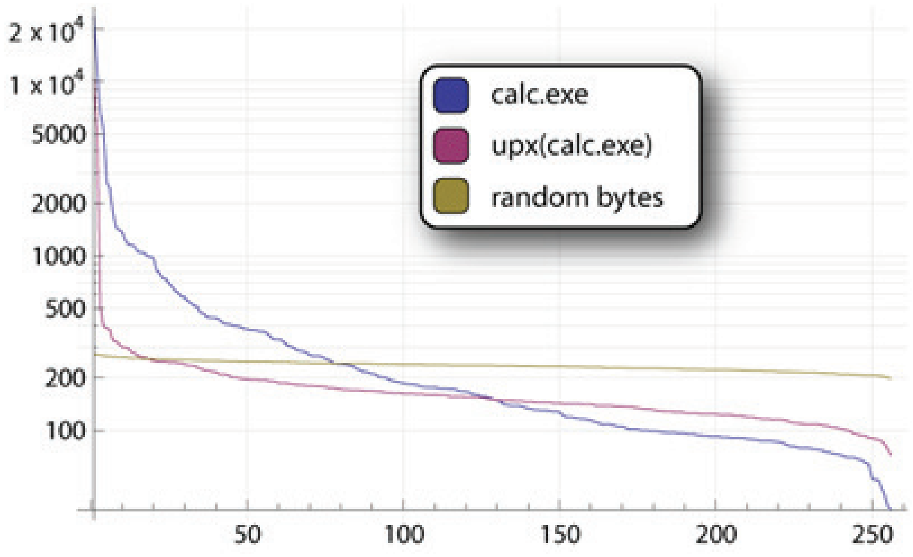
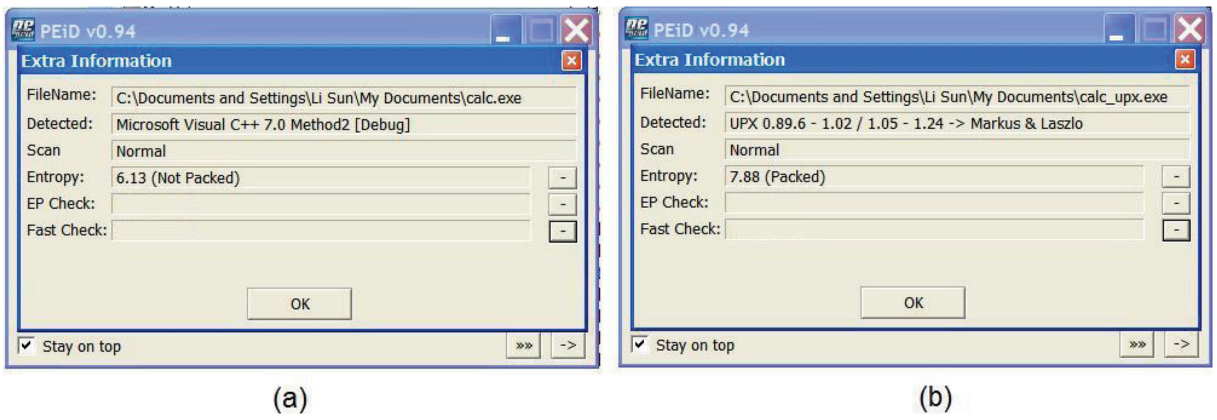


**Figure 2.12:** Byte frequency distribution of calc.exe, UPX 2.03w packed calc.exe and random bytes

#### 2.4.3.2 Entropy

The entropy approach uses an entropy-based metric to detect packed executables. Entropy, or information density, is a term from information theory describing the amount of information in each symbol within a series of symbols. Normally, the highly random data such as a packed file and encrypted/compressed data will exhibit high entropy values while non-random data such as a unpacked file and code exhibits low entropy values. Chapter 3 describes entropy and other information theory concepts in detail.

PEiD provides an entropy check function (Figure 2.13), but no implementation is available. Bintropy [68] separates important sections into blocks of bytes and then calculates the standard *Shannon entropy* value [103] of each block and measures two values for each section, namely average block entropy and maximum block entropy. If there is any section, for which both of these two values are within a pre-defined range that belongs to the class of packed files, the file is then identified as packed. Conti et al. [20] applied byte plot to locate the compressed image in a Word document. The authors mentioned in the paper that the employment of entropy could add meaning to the visualisation.


(a)

PEiD v0.94
Extra Information
FileName: C:\Documents and Settings\Li Sun\My Documents\calc.exe
Detected: Microsoft Visual C++ 7.0 Method2 [Debug]
Scan Normal
Entropy: 6.13 (Not Packed)
EP Check:
Fast Check:
✓ Stay on top

(b)

PEiD v0.94
Extra Information
FileName: C:\Documents and Settings\Li Sun\My Documents\calc_upx.exe
Detected: UPX 0.89.6 - 1.02 / 1.05 - 1.24 -> Markus & Laszlo
Scan Normal
Entropy: 7.88 (Packed)
EP Check:
Fast Check:
✓ Stay on top

**Figure 2.13:** PEiD's entropy check: (a) calc.exe (b) UPX packed calc.exe

## 2.5 Unpacking

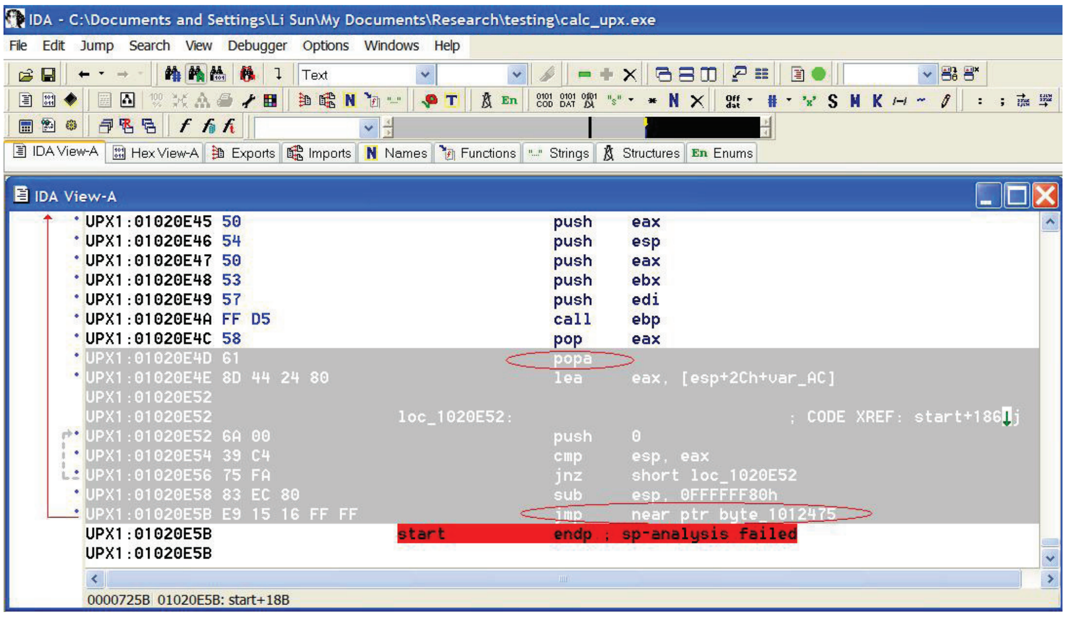
While modern malware uses layers of packers to evade malware detection, legitimate software applies packers too. In fact, many commercial and customized packers are designed for protecting legitimate software. Therefore, in a malware detection process, blacklisting [113] packers, i.e., simply treating the file wrapped in some particular packers as malware, may cause high false positives. To reduce the false positive rate, all detected packed executables need to be unpacked before being passed into the malware detection process. It is a vital step to successful malware analysis and detection.

The objective of PE unpacking is to remove the PE packer's unpacking stub and restore the original executable. A perfect recovery brings the file back to its original state before packing. The unpacker decompresses and decrypts the code and data sections, sets up the import address table, unwinds the stack and transfers the control to the OEP of the unpacked file. However, in the AV industry, sometimes a dumped memory of unpacked file which exhibits malicious code is sufficient. Therefore, the main step involved in unpacking is determining the time when unpacking has finished. At that point, the memory can be dumped for further analysis.

Three approaches have been developed to unpack packed executables, namely manual unpacking, static unpacking and generic unpacking.

### 2.5.1 Manual Unpacking

AV researchers often use a controlled environment to execute the packed binary in a Win32 debugger, such as OllyDBG, for manual unpacking. For a known packer, AV researchers extract its OEP finding module from the database, set a hardware breakpoint on specific instruction, and then let the program run in the debugger until the breakpoint is hit. For example, UPX's unpacking stub usually contains a pair of PUSHA/PUSHAD and POPA/POPAD instructions before it transfers the control (jumps) to OEP. So the OEP is located with the first 'JMP' instruction after the POPA/POPAD instruction. As shown in Figure 2.14, the OEP of the sample (a UPX packed calc.exe) is 01012475. Then if the hardware breakpoint, which has been set on the instruction at address 01012475, is triggered, the unpacking must be finished.


IDA - C:\Documents and Settings\Li Sun\My Documents\Research\testing\calc_upx.exe
File Edit Jump Search View Debugger Options Windows Help
Text
IDA View-A
Hex View-A
Exports
Imports
Names
Functions
Strings
Structures
Enums
IDA View-A

```
UPX1:01020E45 50                       push    eax
UPX1:01020E46 54                       push    esp
UPX1:01020E47 50                       push    eax
UPX1:01020E48 53                       push    ebx
UPX1:01020E49 57                       push    edi
UPX1:01020E4A FF D5                    call    ebp
UPX1:01020E4C 58                       pop     eax
UPX1:01020E4D 61                       popa
UPX1:01020E4E 8D 44 24 80              lea     eax, [esp+2Ch+var_AC]
UPX1:01020E52
UPX1:01020E52           loc_1020E52:                      ; CODE XREF: start+186↓j
UPX1:01020E52 6A 00                    push    0
UPX1:01020E54 39 C4                    cmp     esp, eax
UPX1:01020E56 75 FA                    jnz     short loc_1020E52
UPX1:01020E58 83 EC 80                 sub     esp, 0FFFFFF80h
UPX1:01020E5B E9 15 16 FF FF           jmp     near ptr byte_1012475
UPX1:01020E5B           start          endp ; sp-analysis failed
UPX1:01020E5B
```

```
0000725B 01020E5B: start+18B
```

**Figure 2.14:** Find OEP in UPX packed file

However, many packers use obfuscation methods to alter their signature, such as inserting junk code. Some packers also intentionally use fake strings from other packers or standard compiler code, in order to fool the packer recognizer. For this kind of 'new version' packers, AV researchers need to reverse engineer the packed binary, understand the compression/encryption loop and combine following heuristics to determine whether the OEP has reached:

- **ESP rule**. Before transferring the control to the original file, the program needs to restore all of the registers to their original states before the unpacking routine starts [71]. This normally is achieved by using a push instruction (e.g., 'PUSHAD') to create a backup of the registers on the stack before the unpacking routine is run and executing a pop instruction, 'POPAD', to retrieve all the registers from the stack at the end of the unpacking.

- **Jumps to a different section**. During the packing, the packer usually creates a new section and stores the unpacking stub there. So when the unpacking finishes, the unpacking code should take an intersection jump to the original file which is unpacked in the section other than the unpacker.

- **Executes on the newly written memory**. The original file is compressed/encrypted in a packed file. At some point, the unpacker needs to decompress/decrypt it and store it in the memory, so that the code can be executed.

- **Reaches standard compiler's entry point**. Compilers such as Microsoft Visual C++, GNU C++, and Delphi generate standard entry point signatures for all compiled programs.

Manual unpacking requires the AV researcher's broad knowledge, including Windows internal and assembly programming. Once a certain level of skill has been achieved such actions can be quite quick and precise. However, it can't keep up with daily research work when incredibly huge amounts of packed malware comes in every day.

### 2.5.2 Static Unpacking

Static unpacking is a traditional approach adopted by malware detectors to automate unpacking. It uses specific unpacking routines to decompress/decrypt a packed file without executing the suspicious programs. Some packers, such as UPX and Upack, come with their own stand-alone native unpackers. For other packing algorithms, AV researchers carry a detailed analysis of each algorithm, extract required parameters (e.g., encryption key and the size of the packed area), and develop a corresponding routine to unpack it. One possible simple solution is to disassemble a sample packed file, and just copy the assembler code of the unpacking routine in a self running program (e.g., an AV engine). However, this method is not portable as the code generated is processor-specific.

Obviously static unpacking works well for packers with a known compression engine. However, this general approach has several problems. Firstly, malware authors can easily bypass them using various obfuscation methods on existing packers or using unforeseen or custom packers. Secondly, specific code is needed for each packer (or perhaps each version of the packer). As the number of new packers and variants of existing ones keeps on increasing, the AV engines need to be updated frequently. As a result, the size of the AV engine grows without bound at an increasing rate. Thirdly, it takes a non-trivial amount of effort to develop an accurate unpacker. While the complexity of the packer grows, more effort needs to be invested. Finally, this approach is not safe. If the unpacker contains design or implementation bugs, the malicious payload might be executed. Therefore, it is essential to encapsulate the code within a safe and contained environment.

### 2.5.3 Generic Unpacking - Dynamic Approach

Attempts have been carried out with the aim of providing a generic unpacking that obtains fully decrypted memory for every type of packer without knowing which PE packer is used and what encryption/compression algorithm it utilizes [40, 45, 53, 70, 92, 97, 106, 109, 110]. All these methods involve the execution of the program and of course need to be done in a controlled environment.

#### 2.5.3.1 Run and Dump

The simplest way is directly running the code in a safe environment, such as a virtual machine, and then attaching it to a utility such as Procdump [98] and dumping the memory. This technique is fast and somewhat effective. However, it does not indicate the right point to stop the running program. Besides, it might miss processes that terminate quickly.

#### 2.5.3.2 Emulation

AV researchers often emulate the execution in a safe virtual environment for a sufficiently long period of time until the file is fully decompressed. Computing emulation is a safe, portable and very powerful unpacking tool which has attracted attention from a lot of researchers [6, 9, 40, 106, 109]. However, this method has some drawbacks.

First of all, it is tricky to determine when the unpacking is finished and stop the emulator. Heuristic rules, such as those listed below are good indicators, but they are not always the right choice. In addition, some form of resource limitation is needed to stop a "forever" unpacking. This can be achieved by setting the maximum time-out [9] or the maximum opcode count.

- program jumps into a different section;
- program reaches the entry point that matches the standard compiler's entry point signature;
- program calls API functions which are not usually called from PE compression engines, e.g., CreateWindow();
- program emulates a lot of opcodes but only a few simulated API functions count as this most likely emulating the PE compression engine.

Secondly, the emulator can be easily fooled by the anti-emulation tricks as described in [26, 29].

Thirdly, emulation requires extra computational overhead, and therefore it is slow.

Finally, it is very difficult and time-consuming to develop a high-fidelity and high-performance emulator.

#### 2.5.3.3 Debugger Scripts

Lots of debugger scripts have been developed to control a debugger running to the OEP. These scripts mainly apply certain unpacking heuristics.

Based on the ESP rule [71], a hardware breakpoint can be set on ESP after the program runs to the instruction "pushad" in a debugger. When the breakpoint is hit, it is reasonable to assume that the unpacking has been finished since all original register values are retrieved.

IDA's Universal PE unpacker plugin [43] sets a breakpoint on API GetProcAddress() assuming that the program has been unpacked in the memory and now it starts to set up its import table when a call to this API occurs. Once the breakpoint is hit, the unpacker traces the program in the single step mode until the program jumps to a specified code area that assumed containing the original entry point. Then the unpacker suspends the execution and informs the user.

OllyBonE [110] uses the page protection mechanism to implement break-on-execution. An OllyDBG plugin is provided to allow AV researchers to identify which section in the memory map will be executed when the unpacking is finished, then set the break-on-execute flag for that section. This loads the kernel driver into memory and protects the desired physical memory pages from being executed. When the program is run, OllyDBG will break on the first instruction, which would be the OEP.

While all scripts only work with simple packers, they fail in most other packers. Moreover, these tools require human intervention in specifying the OEP area to the unpacker which is not practical.

#### 2.5.3.4 Normalization Algorithm

The malware normalizer [18] undoes the effects of three common obfuscations which are code reordering, packing and junk insertion. For packed programs, it firstly uses a modified version of the emulator, QEMU, to collect all the memory writes and monitor execution flow. Then it constructs a normalized program instance that contains the generated code and removes all the writes. The algorithm is complex and requires that the execution flow reaches the code generator before it reaches the trigger instruction and therefore it does not suit multistage packer.

Metasm [6, 44] applies semantic approach to automatically remove packer's virtual machine based protections (see Section 2.3.1.3). It analyses the semantics of instruction and optimises the code using techniques such as constant propagation, constant folding and operation folding [44]. After it builds the description of the semantics of the interpreter, a compiler is generated and virtual bytecodes are compiled into native x86 assembly. This approach relies on the hypothesis that they are able to disassemble most of the code they study. It doesn't suit to packed file with control flow obfuscation and requires heavy computation.

#### 2.5.3.5 Automated Tools

There have been several recent attempts at building tools for automated malware unpacking. Most of them are based on packer's intrinsic nature that the unpacked code are created or modified by the unpacking stub on the run time due to the decompression/decryption.

PolyUnpack [97] assumes that the newly generated code (unpacked code) is not presented in the code section of the original packed file. It firstly disassembles the packed file to build a static code model of the program. In the static model, the program is separated into two sections, a set of sequences of code and a set of sequences of data. Then the program is single-step executed and the current instruction is compared with code section of the model. The unpacked code is identified if it does not exist in the static model and the malware's execution is halted. Due to the use of disassembling and single stepping, this approach significantly increases computational overhead and incurs several orders of magnitude slowdown.

Renovo [53] considers the execution of newly written code is an indicator of unpack completion. The system monitors each instruction and tracks instructions that perform memory writes and control transfers. If it is a memory write instruction, the corresponding destination memory is marked as ‘dirty’ which means it is newly generated. If any dirty memory is executed, one layer of unpacking is finished. The address pointed by the instruction pointer is identified as OEP. This approach supports multiple layers of unpacking. However, similar to PolyUnpack, Renovo is an instruction-level approach and also suffers from performance overhead.

Saffron [92] combines dynamic instrumentation techniques with a page fault assisted debugger. It uses a modified version of OllyBonE to dispatch information on the page-faults and run Intel PIN, an instrumentation tool, to monitor the flow of executables. If there is any execution control transfers to dynamically created or modified pages, a memory is dumped for further analysis. It is noted by the author of Saffron that this approach is slow and does not cope with standard anti-debugging tricks [29] due to the use of PIN.

OmniUnpack [70] tracks execution at the page level. It relies on OllyBonE’s page-level protection mechanisms to identify when code is executed from a page that was newly modified. Meanwhile, dangerous system calls are monitored. The termination of unpacking is determined only when the overwritten pages is executed and the program is about to invoke a dangerous system call. At this time, malware detection tool is used to check whether the unpacked file is a malware. If it isn’t, execution resumed.

Justin [45] is another page-level unpacking approach. It detects the written-then-executed memory page and combines this with several behavior heuristics to detect the end of unpacking, including stack point check, and command line argument check. This approach is effective and its additional performance delay is small. However, it is not portable. To date, it only works on Windows.

In general, generic unpacking methods impose a run-time overhead that is not incurred by static unpacking. Moreover, to date, it is infeasible to have a fully generic unpacking strategy. Some sophisticated packers employ virtual machine techniques and other advantage techniques. For this kind of packer, it is impossible to precisely restore an executable into its original form since the original code has been translated into corresponding op-codes of the packer's virtual machine. However, some unpacking strategies exist which can give useful output. Although the dumped unpacked binary may no longer execute on its own, many of its features will be exposed, thereby greatly simplifying analysis.

## 2.6 Analysis Environment

To prevent any damage to the computer system that is in use, malware content researchers need a secure, contained environment in which they can safely let loose and monitor submitted samples. The popular candidate is the virtual machine.

A virtual machine allows the user to run one or more virtual '*guest*' operating systems on top of the '*host*' system, the native operating system. The advantages of using a virtual machine are significant. First of all, the environment provided by the virtual machine is self contained and isolated. Malware can be allowed to execute safely, without fear of its malicious consequences on the wider environment. Secondly, it requires a smaller number of physical machines for the analysis of malware and therefore reduces the hardware cost. This is due to the fact that some malware depend not only on a particular operating system, but also on an actual system version. To such malware, the particular execution environment is needed and a virtual machine provides means of achieving this. Thirdly, complicated network configurations can be simulated in the virtual environment. For example, network shares, Internet servers and services, etc. The other advantage is that virtualised machines can be quickly restored to "known good" states, or to particular steps of infection.

There are several solutions [5, 7, 23, 49, 72] available for running virtual machines. This thesis uses VMware Workstation virtualisation software (VMware) to create a test environment for packed malware.

VMware executes non-sensitive instructions directly on the host CPU at native speed. It is a '*reduced privilege guest*' virtual machine [27] which means that the 'guest' system must virtualize the important data structures and registers themselves, and is run at a lower privilege level than usual. The main reasons to chose VMware over other virtual machines for this thesis are:

- VMware allows us to create and run the simulation of multiple computers on a single physical system simultaneously. Each virtual machine represents a complete PC which has its own processor, memory, network connections and peripheral ports configuration. It is easy to switch from one operating system to another.

- VMware supports a broadest set of operating systems. It runs on both Windows and Linux host operating systems and supports over 80 guest operating systems[1], including Windows, Linux, Solaris x86, Netware, FreeBSD, etc. Moreover, VMware supports both 32-bit and 64-bit host and guest operating systems.

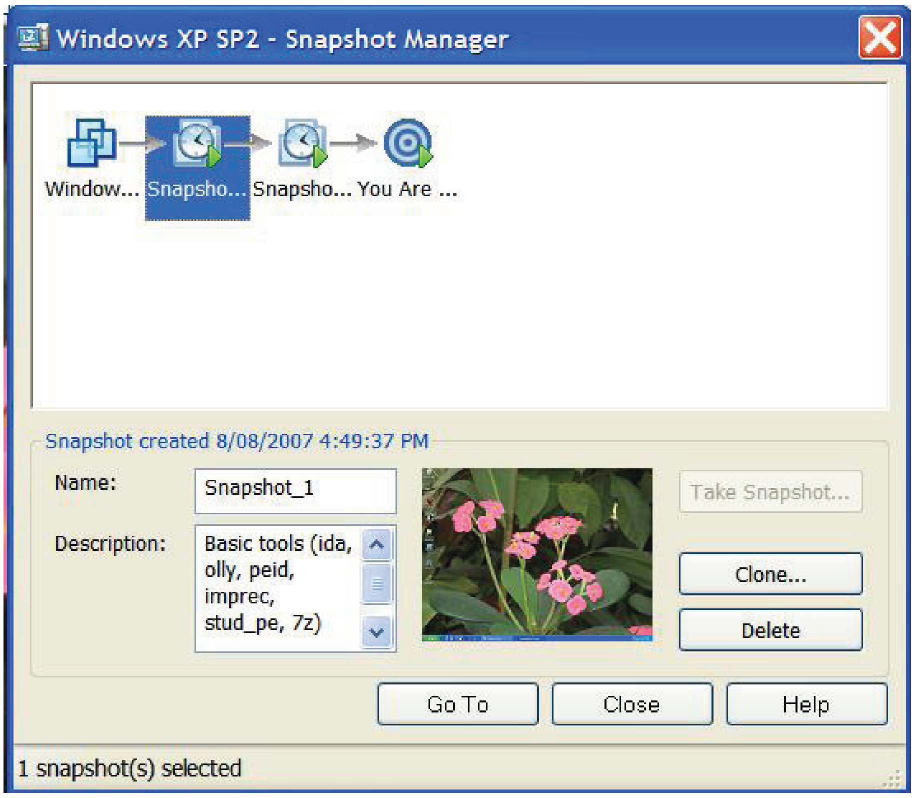
- VMware supports guest-to-host and host-to-guest communications which enable file sharing between virtual machines. While it is certainly possible for a program within a VMware guest to interact with the host system, at this point there is no known malware that can 'escape' the VMware sandbox to execute on the host.

- VMware supports a variety of virtual network configurations including Bridged, NAT, host-only and custom virtual network settings.

- VMware's snapshot feature makes it easy to take a 'snapshot' that preserves the virtual machine disk file at a specific time. With the snapshot manager (Figure 2.15), the virtual machine can be instantly restored to a know state (see the description section in Figure 2.15).

While VMware provides many advantages and therefore is widely used for analysis of malware, attackers are aware of this and new generation of malware have a significant

[1]The full list is available on VMware website, `http://pubs.vmware.com/ws65_ace25/ws_user/wwhelp/wwhimpl/common/html/wwhelp.htm?context=ws_user&file=intro_sysreqs_ws.html`



Snapshot created 8/08/2007 4:49:37 PM

Name: Snapshot_1

Description: Basic tools (ida, olly, peid, imprec, stud_pe, 7z)

1 snapshot(s) selected

**Figure 2.15:** VMware snapshot manager

stake in detecting the presence of a VMware. The detail of these anti-VMware techniques will be discussed in Chapter 7.

## 2.7 Analysis Tool Used in This Thesis

This thesis uses IDA Pro 5.1 as the main packer analysis tool. Most add-on functions are developed as C++ IDA plugins. IDA has many features that make it well suited for our needs. It is a disassembler, a debugger, and is interactive and programmable.

Nowadays, IDA is the most popular disassembler used by AV researchers on static malware analysis since it provides many features that other disassemblers can't achieve. It supports a wide range of processors such as Intel Pentium family, Motorola 68K family and Intel IA-64 Architecture. The whole list of supported processors can be found on

Hex-Rays' website[2]. It also supports various input file formats and can produce various output files. The implementation of advanced techniques such as FLIRT [41] and PIT [42] makes code more readable and accurate. Furthermore, its ability to disassemble code on the fly makes malware analysis work easier, especially on self-modified code analysis.

In addition, the fully functional debugger in IDA Pro complements the static analysis capabilities of the disassembler. IDA supports debugging of 80x86 Windows PE/Linux ELF files and AMD64 Windows PE files, either locally or remotely. Remote debuggers are very useful when one wants to safely dissect potentially malicious application.

The powerful functionality of IDA can be extended and enhanced with a compiled plugin. IDA offers an open plugin architecture which gives user access to the most intimate details of the IDA database and all other aspects of the disassembly via an extensive API. The plugin can be developed in C++ and can be compiled with Borland, Microsoft or GNU C++ compilers. Since the code is compiled to native binary, the execution speed is similar to any other native application.

Furthermore, while other tools suffer from a lack of short or long term support, IDA provides quick and professional technical support to their clients . Finally, a bulletin board is set for various discussions including unsupported IDA SDK discussion. Therefore our tool has more potential on improvement with the new emerging techniques.

[2]`http://www.hex-rays.com/idapro/idaproc.htm`

# Chapter 3

# General Mathematical Concepts

This chapter reviews related mathematical methods and their applications in malware and packer analysis. Section 3.1 presents entropy measures and randomness tests. Section 3.2 discusses pattern recognition systems.

## 3.1 Randomness Measurement

One important characteristic of a packed file is that the packed file usually contains a series of random-looking data bytes, which are transformed from the bytes of the original file via compression and encryption. This feature has been explored and widely used to detect whether an executable is packed or not (see Section 2.4.3). Such statistical approaches are mainly based on two approaches, namely entropy measure and randomness test. We prefer to discuss these approaches separately, even though the goal of a randomness test is to estimate the entropy, or equivalently the departure from the uniform distribution, in some sense. We discuss these two measures below.

### 3.1.1 Entropy measures

The term “entropy” is commonly used in the anti-malware community to broadly refer to how closely a sample set of bytes deviates in frequency from a purely random distribution.

Different scientific disciplines have different definitions of entropy. In thermodynamics, for example, entropy is a measure of the unavailability of a system's energy to do work. However, the formal definition of entropy more commonly used in cryptography, communications theory and coding comes from Claude E. Shannon, and is often called Shannon Entropy to distinguish it from other definitions.

Shannon Entropy is an information-theoretic measure of the amount of uncertainty in a variable [8]. It was introduced in Shannon's seminal works on communication [103] [104], and has found widespread use. The formal definition is as follows:

**Definition 1** *Let the random variable $X$ take values from some set $x_1, \ldots, x_n$. The value $x_i$ occurs with probability $p(X = x_i)$, where*

$$\sum_{i=1}^{n} p(X = x_i) = 1$$

.

**(i)** *The uncertainty of the $x_i$, which is also called self-information $I(X = x_i)$ associated with outcome $x_i$, is defined as*

$$I(X = x_i) = -\log_b \, p(X = x_i) \tag{3.1}$$

**(ii)** *The entropy $H$, or the average uncertainty of $X$, is then calculated as the weighted sum, across all symbols $x_i$ with non-zero probability $p(X = x_i)$, of the information content of each symbol:*

$$\begin{aligned} H(X) &= \sum_{i=1}^{n} p(X = x_i)\, I(X = x_i) \\ &= -\sum_{i=1}^{n} p(X = x_i)\, \log_b \, p(X = x_i) \end{aligned} \tag{3.2}$$

To measure the entropy in bits, the base of the logarithm $b$ is defined to be 2. Therefore, Equation 3.2 becomes

$$H(X) = -\sum_{i=1}^{n} p(X = x_i)\ \log_2\ p(X = x_i) \tag{3.3}$$

It is easy to show that $0 \leq H(X) \leq \log_b n$ with the maximum attained if $p(X = x_i) = 1/n$ for $i = 1, 2, \ldots, n$.

Whilst this remains the formal definition of entropy, the heuristic commonly desired by the anti-malware community is a convenient answer to "how random is this data?", with the expectation that compressed or encrypted data will exhibit a higher degree of randomness than code or text.

### 3.1.2 Statistical Randomness Tests

Randomness tests measure the degree of randomness for a set of data by analyzing its distribution pattern.

**Definition 2** *A numeric sequence is said to be statistically random when each number of the sequence is uniformly distributed and statistically independent of the others, i.e., all numbers of the sequence are generated independently of each other, and the value of the next number in the sequence cannot be predicted, regardless of how many numbers have already been produced.*

A large set of statistical tests have served as randomness tests in the context of random number generators (RNG), in order to assess how "good" they are. A statistical test is formulated to find statistical evidence against the ***null hypothesis*** ($H_0$): "the sequence is a sample from independent and identically distributed (i.i.d.) U(0,1) random variables.". The given sequence is random if it is able to pass the test within a given degree of significance, i.e., the null hypothesis is accepted. In which case, the RNG which generates the testing sequence is proved producing random values.

M. G. Kendall and B. Babington Smith developed the first set of randomness tests in 1938 [55]. They used four separate hypothesis tests to determine whether a given numeric sequence was "random". The first was a *frequency test*, which checked to make sure that the count of each number in a sequence was closely equal. The second test was a *serial test*, which tested whether pairs of two digits (01, 11, 12, etc.) are equally distributed. This is to avoid some non-random sequences passing the first test. For example, a sequence of "123123123" repeats sub-string "123" three times but has equal frequency of 1, 2 and 3. The third test was known as *poker test.* It was another type of frequency test which looked for the expected frequency of the set of five-digit sequences. The fourth test was called *gap test* as it looked for gaps between pairs of the same digit (usually between zeros, e.g. 1 gap for 010, 2 gaps for 0120, and so on) in long sequences, and compared the frequency of gaps with their statistical probabilities. A set of numbers was classified as "random" if it passed all four tests.

As the need for random numbers arises in many applications such as cryptographic applications, more randomness tests, of increasing sophistication were developed. Knuth devotes a significant section of TAOCP (The Art of Computer Programming) to randomness tests [59], including frequency, serial, gap, poker, coupon collector's, permutation, run, maximum-of-t, collision, birthday spacings, and serial correlation. DIEHARD [69] and DIEHARDER [11] battery of randomness tests have enjoyed considerable popularity over the years as an effective suite of tools for measuring how truly "random" given binary data is. Other well-known battery tests include pLab [46], supported by the Department of Mathematics at the University of Salzburg; the Crypt-X suite of statistical tests [78], developed at the Information Security Research Centre at Queensland University of Technology; and the NIST Statistical Test Suite [76].

Two basic randomness tests, the *frequency* test and *runs* test, are introduced below with examples given. For detailed information about the other tests, please refer to the related papers. In each test, a *P-value* is calculated to summarize the statistical evidence against the null hypothesis. This value is the probability that the tested sequence would be higher random than a perfect RNG produced sequence. Therefore, a *P-value* of 1 means that the sequence appears to be perfect random and a *P-value* of 0 means that the sequence appears to be completely non-random. The pass/fail criteria of the testing revolves around the significance level, $\alpha$. If *P-value* $> \alpha$, then the null hypothesis is accepted, i.e., the sequence appears to be random. Otherwise, the sequence appears to be non-random. It is common practice to set the value of $\alpha$ to 0.01, i.e., to 1%. In other words, the confidence level of the test is 99%.

Tests are based on the assumption that the size of the input sequence length, $n$, is large enough (of the order $10^6$ [76]), so that the asymptotic reference distribution can be derived and applied to carry out the tests. If the small size sequence is given, the applied distribution would be inappropriate and would need to be replaced by exact distributions that would commonly be difficult to compute. Here, the small number ($n = 10$) used in examples are just for illustrative purposes.

#### 3.1.2.1 Frequency (Monobit) Test

The Frequency (Monobit) test examines the proportion of zeroes and ones for the entire sequence to determine whether the number of ones and zeros in a sequence are approximately the same as would be expected for a random sequence. The test is used to test the *null hypothesis* $H_0$ that

> *The sequence to be tested is made up of n independent and identically distributed (i.i.d.) uniform Bernoulli random variables, where the probability of ones is 1/2.*

Let $X_1, X_2, \ldots, X_n$ be a sequence of $n$ i.i.d. random variables. For a sufficiently large number of $n$, the distribution of the binomial sum, $S_n = X_1 + \ldots + X_n$, normalized by $\sqrt{n}$, is closely approximated by a standard normal distribution. According to the Central Limit Theorem [52], if $\Phi(z)$ is the cumulative distribution function for the standard Gaussian distribution, then for every real number $z$, we have

$$\lim_{n\to\infty} P\left(\frac{S_n}{\sqrt{n}} \le z\right) = \Phi(z) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{z} e^{-u^2/2}\,du \tag{3.4}$$

So, for positive $z$,

$$\begin{aligned} P\left(\frac{|S_n|}{\sqrt{n}} \le z\right) &= \Phi(z) - \Phi(-z) \\ &= \Phi(z) - (1 - \Phi(z)) \\ &= 2\Phi(z) - 1 \end{aligned} \tag{3.5}$$

If we define the observed value as $s_{obs} = |S_n|/\sqrt{n}$, then the corresponding *P-value* is calculated using Equation 3.5, which is

$$P\text{-}value = 2\Phi(s_{obs}) - 1 = \boldsymbol{erfc}\left(\frac{s_{obs}}{\sqrt{2}}\right) \tag{3.6}$$

Here ***erfc*** is the complementary error function,

$$\boldsymbol{erfc}(z) = \frac{2}{\sqrt{\pi}}\int_{z}^{\infty} e^{-u^2}\,du.$$

The Frequency test is a basic randomness test. If a RNG fails this test then the likelihood of other tests failing is high. Basically, the procedure of the test involves five steps as described in Algorithm 3.1.

We illustrate with an example where $n$ is chosen to be 10 for convenience and should not be used in practice. Consider a sequence of $n$ values such as

$$\varepsilon = 0110101110, \text{ where } n = 10,$$

then $S_n, S_{obs}$, and *P-value* are calculated as below:

```
Data  : A sequence with $n$ bits, $\varepsilon = \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\}$.
Result: The decision that the input sequence is random or non-random
begin
1.  foreach $i \in n$ do
        Convert 0 or 1's into $-1$ or 1, that is $X_i = 2\varepsilon_i - 1 = \pm 1$.
    end
2.  Sum all values $S_n = X_1 + X_2 + \ldots + X_n = 2(\varepsilon_1 + \varepsilon_2 + \ldots + \varepsilon_n) - n$.
3.  Compute the observed value $S_{obs}$ and use it as a test statistic. $S_{obs}$ is
    calculated as the binomial sum, normalized by $\sqrt{n}$.
    $S_{obs} = \frac{|S_n|}{\sqrt{n}}$.
4.  Calculate $P\text{-}value = \boldsymbol{erfc}\left(\frac{S_{obs}}{\sqrt{2}}\right)$, where $\boldsymbol{erfc}$ is the complementary error
    function.
5.  Make the decision based on the $P\text{-}value$:
    if $P\text{-}value < 0.01$ then
        the sequence $\varepsilon$ is non-random;
    else
        the sequence $\varepsilon$ is random.
    endif
end
```

**Algorithm 3.1:** The Frequency test

$$S_n = 2(\varepsilon_1 + \varepsilon_2 + \ldots + \varepsilon_n) - n = 2(0+1+1+0+1+0+1+1+1+0) - 10 = 2$$

$$S_{obs} = \frac{|S_n|}{\sqrt{n}} = \frac{2}{\sqrt{10}} = 0.632455532$$

and

$$P\text{-}value = \boldsymbol{erfc}\left(\frac{S_{obs}}{\sqrt{2}}\right) = \boldsymbol{erfc}\left(\frac{0.632455532}{\sqrt{2}}\right) = \boldsymbol{erfc}(0.447213595)$$

The value of *P-value* can be obtained by checking the ***erfc*** table available in standard probability texts, e.g., in Papoulis [81]. For this example, *P-value* $= 0.5377$. Since *P-value* $> 0.01$, the conclusion is that the input sequence $\varepsilon$ is random with a confidence of 99%.

#### 3.1.2.2 Runs Test

The Runs test exams the total number of runs in the sequence to determine whether the number of runs of ones and zeros of various lengths are approximately the same as would be expected for a random sequence. A run is an uninterrupted sequence of like bits (i.e., either all zeroes or all ones of various length). Each run is bounded before and after with a bit of the opposite value. For example, string '001110' contains three runs, which are '00', '111' and '0'. The Runs test is used to test the *null hypothesis* $H_0$ that

> *We have a sequence of $n$ independent and identically distributed (i.i.d.) Bernoulli random variables, where the probability of a run of ones is 1/2.*

As described in the last section, the Frequency (Monobit) test is a basic randomness test. Therefore, passing the Frequency test is a prerequisite of the Runs test. Let $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$ be a sequence of $n$ i.i.d. random variables. According to the Frequency test, the fixed proportion of 1s in the sequence, $\pi = \sum_j \varepsilon_j/n$, must be close to 1/2, that is,

$$\left|\pi - \frac{1}{2}\right| \leq \frac{2}{\sqrt{n}}$$

The Runs test is based on the distribution of the total number of runs $V_n$. If $\Phi(z)$ is the cumulative distribution function for the standard Gaussian distribution, then for every real number $z$, we have [36]

$$\lim_{n\to\infty} P\left(\frac{V_n - 2n\pi(1-\pi)}{2\sqrt{n}\pi(1-\pi)} \leq z\right) = \Phi(z) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{z} e^{-u^2/2}\, du \tag{3.7}$$

So, for positive $z$,

$$\begin{aligned} P\left(\frac{|V_n - 2n\pi(1-\pi)|}{2\sqrt{n}\pi(1-\pi)} \leq z\right) &= \Phi(z) - \Phi(-z) \\ &= \Phi(z) - (1 - \Phi(z)) \\ &= 2\Phi(z) - 1 \end{aligned} \tag{3.8}$$

For the observed number of runs in a sequence of length $n$, $V_n(obs)$, the corresponding *P-value* is calculated as

$$\begin{aligned} P\text{-}value &= 2\Phi\left(\frac{|V_n(obs) - 2n\pi(1-\pi)|}{2\sqrt{n}\pi(1-\pi)}\right) - 1 \\ &= \boldsymbol{erfc}\left(\frac{|V_n(obs) - 2n\pi(1-\pi)|}{2\sqrt{n}\pi(1-\pi)} \times \frac{1}{\sqrt{2}}\right) \\ &= \boldsymbol{erfc}\left(\frac{|V_n(obs) - 2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right) \end{aligned} \tag{3.9}$$

Here ***erfc*** is the complementary error function defined earlier.

The procedure of the test basically involves six steps. They are illustrated in Algorithm 3.2.

We illustrate with an example. Consider the same sequence used in the Frequency test, that is,

$$\varepsilon = 0110101110, \text{ where } n = 10.$$

Let us try the pre-requested Frequency test firstly. There are total six 1s in the sequence, therefore $\pi = 6/10 = 0.6$, and $|\pi - 1/2| = 0.1$.

The Frequency test bound, $\dfrac{2}{\sqrt{n}} = \dfrac{2}{\sqrt{10}} \approx 0.63246 > |\pi - 1/2|$. So the sequence passed the Frequency test and the Runs test continues.

For the sequence $\varepsilon$, new runs are: $r_1, r_2, \dots, r_{n-1} = 1, 0, 1, 1, 1, 1, 0, 0, 1$. Then $V_n(obs)$ and *P-value* are:

$$V_n(obs) = \sum_{i=1}^{n-1} r_i + 1 = (1 + 0 + 1 + 1 + 1 + 1 + 0 + 0 + 1) + 1 = 7$$

```
Data  : A sequence with $n$ bits, $\varepsilon = \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\}$.
Result: The decision that the input sequence is random or non-random
begin
1.   Compute the proportion of 1s in the $\varepsilon$:
     $\pi = \frac{\sum_{j=1}^{n} \varepsilon_j}{n}$ where $\varepsilon_j = 1$.
2.   Determine if the pre-requisite Frequency test is passed:
     if $|\pi - 1/2| \geq \frac{2}{\sqrt{n}}$ then
         Set $P\text{-}value = 0$;
         Jump to step 6 (i.e., the sequence $\varepsilon$ is non-random).
     endif
3.   Set new runs.
     foreach $i \in (n-1)$ do
         if $\varepsilon_i = \varepsilon_{i+1}$ then
             $r_i = 0$;
         else
             $r_i = 1$;
         endif
     end
4.   Compute the observed number of runs $V_n(obs)$ as a test statistic.
     $V_n(obs) = \sum_{i=1}^{n-1} r_i + 1$.
5.   Calculate $P\text{-}value = \boldsymbol{erfc}\left(\frac{|V_n(obs) - 2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right)$, where erfc is the
     complementary error function.
6.   Make the decision:
     if $P\text{-}value < 0.01$ then
         the sequence $\varepsilon$ is non-random;
     else
         the sequence $\varepsilon$ is random.
     endif
end
```

**Algorithm 3.2:** The Runs test

and

$$P\text{-}value = \boldsymbol{erfc}\left(\frac{|7 - 2 \times 0.6 \times (1 - 0.6)|}{2\sqrt{2 \times 10} \times 0.6 \times (1 - 0.6)}\right) = 0.147232$$

Since $P\text{-}value > 0.01$, the conclusion is that the input $\varepsilon$ is random with a confidence of 99%.

## 3.2 Pattern Recognition

Pattern recognition techniques have recently been used in anti-malware community, mainly for the purpose of identifying new or unknown malware [3, 4, 19, 56, 61, 100, 102, 107, 116]. Pattern recognition [117] aims to recognize a particular class from a measurement vector. Different pattern classes with different measurement vectors correspond to different points in measurement space and patterns with similar appearance tend to cluster together. Therefore, a mapping relationship can be established from the measurement space into the decision space. There are two essential technologies involved in a pattern recognition system, namely feature extraction and classification.

Feature extraction retrieves the common features (patterns) among a set of objects. A feature is the measurement of a property of an object. A feature set describes the essential properties of an object using a greatly reduced number of parameters. However, only the features that properly represent the original object can lead to a satisfying pattern recognition result. In other words, the extracted features of objects in each class should be represented in a distinctive way. This permits a set of objects to be classified into different classes.

Classification is the process that first analyses the training set, develops a classification model for each class and then applies the model to classify the test set based on their features.

**Definition 3** *A training set is a collection of records of which the class label have been* *provided by a trusted source.*

**Definition 4** *A testing set is used to verify the accuracy of the model and consists of records with class labels that you want to predict.*

**Definition 5** *A classification model (also called a classifier) can be built as follows: given a set of $N$ training data in which each record consists of a pair: a feature vector of $n$ features $\boldsymbol{x} = \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$ and the associated "truth" class $y_j$, produce a relationship $f : \boldsymbol{x} \rightarrow y_j$ that maps any feature vector $\boldsymbol{x} \in X$ to its true class $y_j \in y$.*

The rest of this section are organized as follows: Section 3.2.1 describes various statistical classifiers. Section 3.2.2 gives a literature review on the related anti-malware works that apply pattern recognition technologies. Section 3.2.3 presents general metrics used for the evaluation of classification effectiveness. The last section introduces the main pattern recognition tool, Weka, used in this thesis.

## 3.2.1 Classifiers

### 3.2.1.1 One-Rule

One simple classification technique is the one-rule classifier which classifies records using only one "if ... then..." rule. The objective of this technique is to extract a classification rule that covers many of the positive examples and as few of the negative samples in the training set. However, finding an optimal rule is computationally expensive given the exponential size of the search space. The one-rule function addresses the exponential search problem by growing the rules in a greedy fashion. It generates an initial rule $r$ and keeps refining the rule until a certain stopping criterion is met. The rule is then pruned to improve its generalization error.

#### 3.2.1.2 The Naive Bayes (NB) Algorithm

The NB classifier [114] uses a statistical approach to the problem of pattern recognition. The Bayes rule is the fundamental idea behind a NB classifier. For a feature vector $\mathtt{x}$ with $n$ attributes $\mathtt{x} = \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ and a class variable $y_j$, let $P(\mathtt{x}|y_j)$ be the class-conditional probability for the feature vector $\mathtt{x}$ whose distribution depends on the class $y_j$. Then $P(y_j|\mathtt{x})$, the *posteriori* probability that feature vector $\mathtt{x}$ belongs to class $y_j$ can be computed from $P(\mathtt{x}|y_j)$ by Bayes rule:

$$P(y_j|\mathtt{x}) = \frac{P(\mathtt{x}|y_j) \times P(y_j)}{P(\mathtt{x})} \tag{3.10}$$

The NB algorithm applies "naive" conditional independence assumptions which states that all $n$ features $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ of the feature vector $\mathtt{x}$ are all conditionally independent of one another, given $y_j$. The value of this assumption is that it dramatically simplifies the representation of $P(\mathtt{x}|y_j)$, and the problem of estimating it from the training data. In this case,

$$\begin{aligned} P(\mathtt{x}|y_j) &= P(\mathtt{x}_1 \ldots \mathtt{x}_n|y_j) \\ &= \prod_{i=1}^{n} P(\mathtt{x}_i|y_j) \end{aligned} \tag{3.11}$$

and equation (3.10) becomes

$$P(y_j|\mathtt{x}) = \frac{P(y_j) \prod_{i=1}^{n} P(\mathtt{x}_i|y_j)}{P(\mathtt{x})} \tag{3.12}$$

In a classification system, the feature vector $\mathtt{x}$ belongs to the class $y_j$ with the highest probability $P(y_j|\mathtt{x})$. Since $P(\mathtt{x})$ is always constant for every class $y_j$, it is sufficient to choose the class that maximizes the numerator in (3.12), $P(y_j) \prod_{i=1}^{n} P(\mathtt{x}_i|y_j)$. In other words,

$$y_{max} = \arg\max_{y_m} P(y_j) \prod_{i=1}^{n} P(\mathtt{x}_i|y_j) \tag{3.13}$$

The NB classifier is easy to implement and can be trained very efficiently in a supervised learning setting, computation time varies approximately linearly with the number of training samples. Despite the apparent over-simplified assumptions of independence, the NB classifier often competes well with more sophisticated classifiers [124].

#### 3.2.1.3 The $k-$Nearest Neighbor (kNN) Algorithm

k-Nearest Neighbor [1] is amongst the simplest of all machine learning algorithms. The idea behind it is quite straightforward. To classify the test packed file, the system firstly finds the $k$ training files which are the most similar to the attributes of the test file. These training files are called *Nearest Neighbors* as they have the shortest Euclidean distance to the test file. If we have a feature vector $\mathtt{x}$ in $R^n$ and a vector $\mathtt{y}$ in $R^n$, the Euclidean distance $d(\mathtt{x}, \mathtt{y}) = \sqrt{\sum_i (\mathtt{x}_i - \mathtt{y}_i)^2}$. The test file is categorized based on the category of its nearest neighbors. In the case where neighbors belong to more than one class, the test is assigned to the majority class of its nearest neighbors.

As shown in Figure 3.1, the test file (green cross) can be classified as the first class of the red plus sign or the second class of the blue minus sign. If set $k$ to 1, the test file will be classified as the class of red plus sign since it is inside the inner circle. Similarly, if set $k$ to 3, the test file will be classified as the class of the blue minus sign as there are two blue minus signs but only one red plus sign inside the outer circle.

#### 3.2.1.4 The Sequential Minimal Optimization (SMO) algorithm

SMO is a fast implementation of Support Vector Machines (SVM) [12]. Given data of two classes as two sets of feature vectors in an $n-$dimensional space, a SVM constructs an optimal hyperplane that separates a set of one class instances from a set of other class instances and maximizes the margin between the two data sets. That is to say if

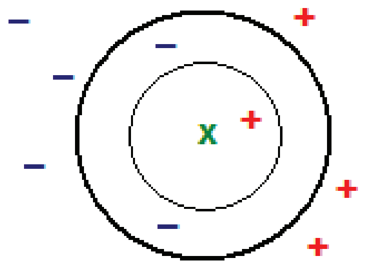


**Figure 3.1:** Examples of kNN algorithm

two parallel hyperplanes are constructed, one on each side of the hyperplane and passing through the nearest data point in each data class, the distance between the parallel hyperplanes needs to be as far apart as possible while still separating the data into two classes.

Training a SVM is slow due to solving a very large quadratic programming (QP) optimization problem. SMO [87] decomposes the large QP problem of SVM into QP sub-problems, which can be solved analytically and thus avoids using an entire numerical QP as an inner loop. In addition, SMO requires no extra matrix storage at all therefore the amount of memory required is linear in training set size. It allows SMO to handle very large training sets.

#### 3.2.1.5 The Best-First Decision Tree (BFTree) Algorithm

The BFTree algorithm is a decision tree [89] that maps from attributes of an item to conclusions about its target class. In a tree that describes a set of packed files, each internal node represents a test on a file feature, each branch from a node corresponds to a possible outcome of the test, and each terminal node contains a packer class prediction. In each step of tree expansion, the best-first top-down strategy is applied [105]. Thus, the “best” node, which maximally reduces the impurity among all nodes available for splitting, is added to the tree firstly. This partitioning of the feature space is recursively executed until all nodes are non-overlapping or a specific number of expansions is reached. For the latter, pruning methods are used to decrease the noise and variability in the data and therefore to achieve better performance.

### 3.2.2 Related Work

There have been a few recent attempts to use pattern recognition techniques for automated malware detection [56, 61, 102, 107, 116]. Most of them only classified files between malicious and benign, but not by family of malware. In early attempts, Tesauro et al. [56, 116] developed a neural network for virus detection. The system is specially designed for the detection of boot sector viruses using feature trigrams. The *trigrams* are three byte strings. They are selected in the feature set if they appear frequently in viral boot sectors but infrequently in uninfected software and definitely do not appear in legitimate ones. In the experiments, 200 viral boot sectors and 100 legitimate boot sectors were used as the data set in which half of them were the training set and the other half were the test set. Using a classification threshold of 0.5, performance on the test set was typically 80-85% for the viral boot sectors and 100% for the legitimate boot sectors. The classifier has also been incorporated into the IBM anti-virus product and has caught approximately 75% of new boot sector viruses.

Schultz et al. [102] used data mining methods to detect malware. They used three types of features, binary profiles of DLLs, strings and sequences of $n$ adjacent bytes (also called $n$-grams), and paired each feature with a single learning algorithm. That is, a rule-base classifier was applied to the binary profiling; string data was used to fit a NB; and an ensemble of multi-NB classifiers is used on the $n$-grams data. In the latter, the $n$-grams data were partitioned into six parts firstly, then each classifier was trained on each partition of the data. The experiments were carried on a data set which contained 3301 malicious programs and 1001 clean programs. Among them, 38 of the malicious programs and 206 of the benign programs were in the Windows Portable Executable (PE) format.

The results showed that naive Bayes with strings achieved the best accuracy than others. However, their experiments did not provide a fair comparison among the classifiers as different features were used for different classifiers. Moreover, different training sets were used to training different classifiers.

Using the same ideas, MECS [61] extracted byte sequences from the executables, converted these into n-grams, and constructed several classifiers: kNN, NB, SVM, decision trees J48, boosted NB, boosted J48 and boosted SVM. Muazzam tried other features [107]. Instead of using fixed length instructions or *n-gram* features, the author used Vector Space Model [34, 101, 120] to extract variable length instruction sequence as the primary classification feature and applied an array of classification models, including logistic regression, neural network, decision tree, SVM, Bagging and random forest.

Several researchers addressed the issue of email classification with the aid of machine learning techniques [3, 4, 19, 100]. Cohen [19] used a rule-based algorithm to classify email into folders based on the text of messages. Sahami et al. [100] employed a NB technique to the problem of junk E-mail filtering. Androutsopoulos et al. [3, 4] also used this approach to classify spam emails and legitimate ones. They compared two classifiers, NB and $k$NN. Both algorithms achieved very high classification accuracy but $k$NN with $k = 2$ slightly outperformed others.

The closest work to this thesis was conducted by Perdisci et al. [83]. They applied various pattern recognition techniques to classify executables into two categories, packed and non-packed. Nine features are combined together for classification, namely number of standard and non-standard sections, number of executable sections, number of readable/writable/executable sections, number of entries in the PE file's Import Address Table (IAT), PE header entropy, code section entropy, data section entropy and file entropy. The system achieved very high accuracy (above 95%) using NB, J48 decision tree, bagged, kNN or Multi Layer Perceptron (MLP) classifiers. However, this approach is not able to detect what family of packers a packed file belongs to.

### 3.2.3 Evaluating Classification Performance

When comparing the performance of different classification techniques, it is important to assess how well a classification model is able to correctly predict records to the actual classes. Several metrics are conventionally in use to numerically qualify classification effectiveness performance.

To introduce the metrics, let us define that for a class $y_j$, a record is *positive* if it is predicted to belong this specific class and is *negative* if it is predicted to belong other classes.

**Definition 6** *Suppose that for a test set with* $n$ *records, the set of positive records and negative records for the class are known (for example, as the result of human judgment), and* $P$ *and* $N$ *are the number of positive records and negative records respectively,* $n = P + N$*. We use four important counts defined below:*

- *TP represents the true positives which is the number of positive records correctly identified as specific class.*
- *FP represents the false positives, the number of negative records which don't belong to the class but were incorrectly identified as it.*
- *TN represents the true negatives which refers the number of negative records correctly identified as other classes.*
- *FN represents the false negatives, that is the number of positive records which belong to the class but were incorrectly identified as other classes.*

*so,* $P = TP + FN$ *and* $N = FP + TN$*.*

#### 3.2.3.1 Confusion Matrix

The *confusion matrix* provides information needed to determine how well a classification model performs. In Table 3.1, classification results of two classes, A and B, are listed in a 2x2 confusion matrix. For the testing set of total $n = a + b + c + d$ records, class A includes $P = a + b$ positive records and $N = c + d$ negative records while

$$TP = a \; ; \; FP = c \; ; \; TN = d \; ; \text{ and } FN = b.$$

Similarly, class B contains $P = c + d$ positiver records and $N = a + b$ negative records while

$$TP = d \; ; \; FP = b \; ; \; TN = a \; ; \text{ and } FN = c.$$

| | | Predicted Class | |
|---|---|---|---|
| | Class | A | B |
| Actual | A | $a$ | $b$ |
| Class | B | $c$ | $d$ |

**Table 3.1:** Confusion matrix for for a binary classification problem

#### 3.2.3.2 One-dimensional Measures

In addition to the confusion matrix, some single-figure measures of effectiveness have been developed. In many cases, these single measures are more attractive because of their compactness. The main single measures are introduced below.

**Definition 7** *The accuracy is the percentage of test set records that are correctly identified by the classifier. That is,*

$$Accuracy = \frac{TP + TN}{n} = \frac{TP + TN}{P + N} \tag{3.14}$$

The accuracy provides an overall performance of the effectiveness. However, this measure has one problem. Suppose that a test set contains a large amount of negative records and very small amount of positive records, and the classifier only can label negative class. In this case, though TN is very high and TP is very low, the accuracy is still high.

**Definition 8** *The true positive rate (TPrate) and false positive rate (FPrate) are introduced to measure the proportion of the positive records that are correctly identified and the proportion of the negative records that are incorrectly identified, respectively. For each class, they are calculated as*

$$TPrate = \frac{TP}{P} = \frac{TP}{TP + FN} \tag{3.15}$$

*and*

$$FPrate = \frac{FP}{N} = \frac{FP}{FP + TN} \tag{3.16}$$

#### 3.2.3.3 Precision and Recall

Two other fundamental ways to measure classification effectiveness are *precision* and *recall*.

**Definition 9** *Precision is the proportion of records classified as positive which are classified correctly. For each class, the precision is defined as*

$$\begin{aligned} Precision &= \frac{\textit{number of records that are correctly classified as positive}}{\textit{total number of classified positive records}} \\ &= \frac{TP}{TP + FP} \end{aligned} \tag{3.17}$$

**Definition 10** *Recall is the proportion of positive records that have been correctly identified. For each class, the recall is*

$$\begin{aligned} Recall &= \frac{\textit{number of records that are correctly classified as positive}}{\textit{total number of positive records}} \\ &= \frac{TP}{TP + FN} \end{aligned} \tag{3.18}$$

A combined measure for precision and recall is the *F-measure*.

**Definition 11** *The F-measure is computed as*

$$F - Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3.19}$$

#### 3.2.3.4 $K$-Fold Cross Validation

$K$-fold cross validation [60] is a common technique to estimate the accuracy of a classifier. Using this technique, the whole data set is randomly partitioned into $K$ equal-size subsets $D_1, D_2, \ldots, D_k$. There are a total $K$ runs of validation. During each run, one subset is used as a *testing set* to assess the performance of a fully-trained classifier, and the other $K - 1$ subsets are used as a *training set* to train the classifier. That is, in the first run, subsets $D_2, D_3, \ldots, D_k$ serve as the training set to test subset $D_1$. In the second run, subsets $D_1, D_3, \ldots, D_k$ serve as the training set to test subset $D_2$, and so on. Therefore, all the examples in the dataset are eventually used for both training and testing and each example is used as a test sample exactly once.

### 3.2.4 Weka

There are a number of tools [2, 73, 93, 121] which have been developed for statistical classification and pattern recognition. The thesis makes use of Weka 3.6.0 (Waikato Environment for Knowledge Analysis) developed at the University of Waikato [121]. It is a public-domain machine learning software that is written in Java and runs on various platform, including Windows, Mac OS X and Linux.

Weka contains a collection of visualization tools and algorithms for data analysis and predictive modelling. The choice of Weka was based on the research direction of the thesis. Weka supports several standard data mining tasks, specifically, data preprocessing, clustering, classification, regression, visualization, and feature selection. In terms of classification, Weka supports a large number of statistical classifiers, from simple techniques such as rule-based and $k$NN classifiers to more advanced techniques such as SVM and random forest. In addition, Weka provides different classifier evaluation options, such as cross validation and percentage split. Weka is easy to use. It provides both graphical user interface (GUI) and command line interface. In this thesis, Windows GUI is used as the start point. The command line interface can be integrated into an AV product later for in-depth usage.

# Chapter 4

# A Local Randomness Test

Packer detection is the first step in not only packer analysis but also malware analysis. For any suspicious executable, it is important to check whether it is packed. If it isn't, it can be directly passed to a malware scanner for malware detection. Otherwise, the file will be firstly sent for packer analysis. The packer used needs to be identified and the file needs to be unpacked. Then the unpacked file can be sent to a malware scanner. The heuristics discussed in Section 2.4 can serve as an aid to packer detection. However, many of them are not effective. In this chapter, we seek to improve the effectiveness of the packer analysis and detection through randomness tests.

A fast randomness test, which we shall call *local randomness test*, is introduced. In contrast to existing algorithms used in anti-malware which simply give an overall randomness value for the input sequence, the new test preserves local detail, so that a plot can be made of an entire file showing areas of high randomness and low randomness. Areas of low randomness (code and header, in the case of a PE file) show as lower $y$-values on the plot, whereas areas of high randomness (encrypted or compressed data) show as areas with high $y$-values. The plot can be a very effective clue to find packed data or even a cryptographic key in a sea of more structured data.

## 4.1 Introduction

The main idea behind the theory of Kolmogorov randomness (also called *algorithmic randomness*) for finite strings is that a string is random if and only if it is "incompressible" [65]. The Kolmogorov complexity is defined (see Chapter 14 of Cover and Thomas [24]) with respect to a universal Turing machine and cannot be computed in practice. In this thesis, we will use compression algorithms as an approximation to Kolmogorov complexity. This basically indicates that the better the compression algorithm used, the more random the compressed data, at the output of the algorithm.

A packer employs compression or encryption techniques to transform the physical appearance of the original file in order to hide it from outside scrutiny. Intuitively, the heavy use of compression or encryption should bring the bytes closer to the appearance of a sequence of random bytes. Therefore, randomness tests, sometimes called entropy tests, can be a fast way to estimate whether a file is packed or not. The inference is that the higher the randomness of the data, the more likely the executable is packed (compressed or encrypted).

Most algorithms we have seen simply give an overall randomness value for the entire file or a whole section, such as the data section. The problem with a simple global randomness value is that it will indicate that even a heavily packed PE file is not particularly "random", due to such non-random features as the PE header, and fields of zeroes that nest in between PE structures. Therefore, for a packed file, the local randomness detail is more useful than a global randomness value. *The heuristic commonly desired by the anti-malware community is a convenient answer to "how random is this data?", with the expectation that compressed or encrypted data will exhibit a higher degree of randomness than code or text.*

To address this problem, let the whole file be partitioned into windows (also called blocks), starting at the first character and continuing through to the last, then apply randomness measurement locally on each individual window instead of the whole file. In

Section 3.1, two main types of randomness measures, Shannon entropy and randomness tests, were described. Shannon entropy measures the amount of uncertainty in a random variable. Whilst this remains the formal definition of entropy, it does not particularly measure the randomness level of a block of data. Besides, the entropy for any window of data depends crucially on the probabilistic model, and the model can be reliably estimated from the data itself only for very long data segments. In our case, we don't have such a probabilistic model and in fact we don't need one.

In the research community, a large body of work has been devoted to determining whether data is random in the context of random number generators (RNG), in order to assess how good they are. Randomness tests, such as Knuth's tests [59], the DIEHARD[69] and DIEHARDER[11] battery of randomness tests, etc., are all effective suite of tools for measuring how truly "random" given binary data is.

When looking for a heuristic measure of randomness for malware packers, these randomness tests have the dual disadvantage that:

(a) they require significant amounts of data (at least 128 MB in the case of DIEHARD) that a packed file normally does not have. Most tests are based on the assumption that the size of the input sequence length is large, so that the asymptotic reference distribution can be derived and applied to carry out the tests. If a short sequence is given, the applied distribution would be inappropriate and would need to be replaced by exact distributions that would commonly be difficult to compute.

(b) they are designed to judge whether the random number generator is good. For an input sequence, the output of all these tests is only one value, random or non-random at an indicated level of significance, say 5%. Therefore these tests are not particularly useful at distinguishing highly random parts of the file from less random segments.

This section of the thesis illustrates that by adjusting some simple and well-known algorithms and data structures from the compression community, useful heuristic measures of the amount of "randomness" in different parts of a sample executable program can be derived.

### 4.1.1 Mathematical Modeling

The problem we have now is to 'fit' a model to *a small amount of data* to determine how random it is. The data, in this case, the packed file, can be examined on a binary or non-binary base.

#### 4.1.1.1 Binary Model

For binary data given as a sequence of $n$ bits, the proposed local randomness test examines the proportion of zeros and ones to determine how random the data is. The test is used to test the *null hypothesis* $H_0$ that

> *The sequence to be tested is made up of $n$ independent and identically distributed (i.i.d.) uniform Bernoulli random variables, so $Pr[X_k = 1] = Pr[X_k = 0] = 1/2$ for all $k = 1, 2, \cdots, n$.*

and the *alternative hypothesis* $H_1$ that

> *The sequence to be tested is made up of $n$ independent and identically distributed (i.i.d.) random binary biased variables with bias $\epsilon$, so $|Pr[X_k = 1] - (1/2)| = \epsilon$ for all $k = 1, 2, \cdots, n$.*

Let $X_1, X_2, \ldots, X_n$ be a sequence of $n$ i.i.d. random variables taking values in $\{0, 1\}$. Suppose for each variable, the probability of 'one' is $p$ and the probability of 'zero' is $1-p$. For any number $n$, if $S_n$ represents the number of ones that occur in the sequence, i.e. $S_n = X_1 + \ldots + X_n$, then $S_n$ is said to be a binomial random variable with parameters $(n, p)$, where $p = 1/2$ if $H_0$ is correct, and $p = 1/2 + \epsilon$ or $p = 1/2 - \epsilon$ if $H_1$ is correct.

The probability mass function of $S_n$ having parameters $(n, p)$ is given by

$$Pr[S_n = s] = \binom{n}{s} p^s \, (1-p)^{n-s}, \qquad s = 0, 1, \ldots, n \tag{4.1}$$

where

$$\binom{n}{s} = \frac{n!}{s!(n-s)!}$$

Under either hypothesis $H_0$ or $H_1$, the distribution of the data is fully specified. The likelihood ratio test statistic can be written as:

$$\Lambda(s) = \frac{L(H_0|S_n = s)}{L(H_1|S_n = s)} = \frac{Pr[S_n = s|H_0] \; Pr[H_0]}{Pr[S_n = s|H_1] \; Pr[H_1]} = \frac{Pr[S_n = s|H_0]}{Pr[S_n = s|H_1]} \tag{4.2}$$

where $Pr[H_0]$ is the a priori probability that $H_0$ is true and where $Pr[H_0] = 1 - Pr[H_1]$, and we assume this value is 1/2.

As the value of $\binom{n}{s}$ is same in both $Pr[S_n = s|H_0]$ and $Pr[S_n = s|H_1]$, Equation 4.2 can be simplified as:

$$\Lambda(s) = \frac{1}{2^{n-1}\left[\left(\frac{1}{2}-\epsilon\right)^s \left(\frac{1}{2}+\epsilon\right)^{n-s} + \left(\frac{1}{2}+\epsilon\right)^s \left(\frac{1}{2}-\epsilon\right)^{n-s}\right]} \tag{4.3}$$

For a sequence of $n$ bits, if $\Lambda(s) > 1$, we accept $H_0$. Otherwise, we accept $H_1$.

The problem of this binary approach is that the likelihood ratio test $\Lambda(s)$ now is just a function of $s$, the number of set bits in the data. As shown in Figure 4.1, when $n$ and $\epsilon$ is set, the closer the value of $s$ is to $n/2$, the bigger the $\Lambda$, and more likely the data is random.

Therefore, in a packed file, the randomness test result is only determined by the value of bytes it contains. For example, the byte ‘A’ (ASCII 65, represented as ‘01000001’) has 2 bits while the byte ‘S’ (ASCII 83, represented as ‘01010011’) has 4 bits. Using binary approach, the data of ‘S’ is assumed random but ‘A’ is not. Moreover, the test only uses

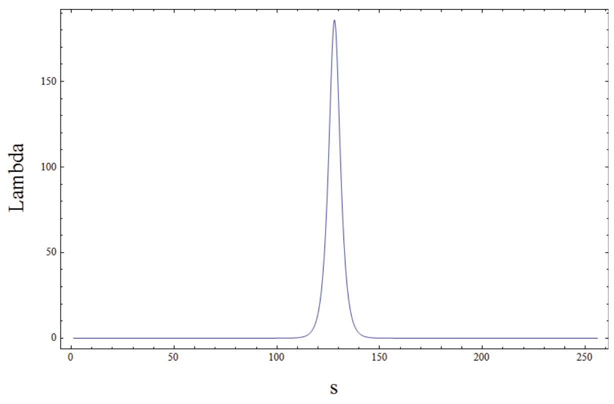


**Figure 4.1:** The likelihood function $\Lambda(s)$ as a function of $s$, where $n$ is 256 and $\epsilon$ is 0.1.

local information of each window which will be biased.

A small experiment has been carried out and the experimental results confirm the above conclusion. In the experiment, we collected one original unpacked calc.exe file, and three packed files, namely FSG 2.0, MEW 11 and Morphine 2.7 packed calc.exe. For each file, we measured likelihood ratio for every 256 bits (i.e., 32 bytes) of data of the file. As shown in Figure 4.2, the results are unsurprisingly disappointing. Even the unpacked calc.exe file is shown as highly random data. In packed file, it is impossible to identify compressed/encrypted data.

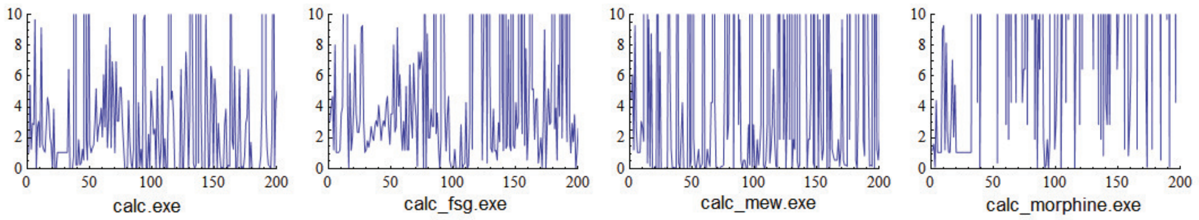


**Figure 4.2:** Binary randomness test results of the original unpacked, FSG, MEW and Morphine packed calc.exe file, where $n$ is 256 (32 bytes) and $\epsilon$ is 0.1.

#### 4.1.1.2 Non-binary Model

As discussed above, a non-binary model seems more appropriate for solving our problem. In the non-binary approach, we select byte as the base unit since the byte is the fundamental building block of most executable programs. For a non-binary data with a sequence of $n$ bytes, the proposed local randomness test examines the proportion of each byte to determine how random the data is. The test is used to test the *null hypothesis* $H_0$ that

> *The sequence to be tested is made up of $n$ bytes that have a discrete uniform distribution. Every byte has equal probability 1/256.*

If a random variable $X$ has any of 256 possible values, $x \in \mathcal{X} = \{0, 1, \cdots, 255\}$, it has the uniform distribution, $U$, if $(U_0, \cdots, U_{255}) = \left(\frac{1}{256}, \cdots, \frac{1}{256}\right)$, where $U_i = Pr[X = i]$ for all $i = 0, \cdots, 255$.

Let $X_1, X_2, \ldots, X_n$ be a sequence of $n$ bytes which follow distribution $P$, the randomness test is carried by measuring the $\mathcal{L}_1$ distance between $P$ and $U$. If the distance is small, we accept $H_0$. Otherwise, we reject $H_0$.

**Definition 12** *The $\mathcal{L}_1$ distance between any two distributions $P_1$ and $P_2$, with support in a finite set $\mathcal{X}$, is defined as*

$$||P_1 - P_2||_1 = \sum_{x \in \mathcal{X}} |P_1(x) - P_2(x)|.$$

**Definition 13** *For probability distributions $P_1$ and $P_2$ of a discrete random variable $X$, the divergence of $P_1$ from $P_2$ is defined to be*

$$D(P_1||P_2) = \sum_{x \in \mathcal{X}} P_1(x) \ \log\left(\frac{P_1(x)}{P_2(x)}\right) \tag{4.4}$$

It has been proved that for two distributions, $P_1$ and $P_2$, the divergence $D(P_1||P_2)$ has the relationship below with the $\mathcal{L}_1$ distance:

$$D(P_1||P_2) \geq \frac{1}{2\ln 2}||P_1 - P_2||_1^2,$$

see, for example, Lemma 11.6.1 in [24].

In our case, the divergence of $P$ from $U$, $D(P||U)$, is measured. Since U is the uniform distribution, i.e., $U_i = \frac{1}{256}$ for $i = 0, \cdots, 255$, we get:

$$\begin{aligned} D(P||U) &= \sum_{x \in \mathcal{X}} P(x) \log \left( \frac{P(x)}{1/256} \right) \\ &= \sum_{x \in \mathcal{X}} P(x) \log P(x) - \log \left( \frac{1}{256} \right) \\ &= 8 + \sum_{x \in \mathcal{X}} P(x) \log P(x) \\ &= H(U) - H(X), \end{aligned} \tag{4.5}$$

and

$$D(P||U) \geq \frac{1}{2 \ln 2} ||P - U||_1^2.$$

Therefore, we can use the entropy, $H(X)$, directly to test the randomness level (equivalently, the non-uniformity) of the data. The larger the entropy, the closer it is to its maximum value of $\log_2(256) = 8$ bits, which means that the smaller the $\mathcal{L}_1$ distance between $P$ and the uniform distribution $U$. As entropy is well-estimated by Huffman compression, this thesis uses an adjusted Huffman coding (Section 4.1.2) to measure the randomness value of a given file. Our experimental results described in Section 4.3 show that this model works in practice.

### 4.1.2 Huffman Coding

Huffman coding [48] is one of the best-known and easy to implement lossless data compression algorithms. It was developed by a postgraduate student at MIT by the name of David A. Huffman for determining minimum-redundancy codes. The basic idea of the algorithm is to express the most common characters using shorter strings of bits than are used for less common source symbols. It works by constructing a bottom-up mechanism, an extended binary tree, based upon the frequency of symbols. For example, for the following unreasonably famous string:

'This program cannot be run in DOS mode'

The frequency table for the symbols is shown in Figure 4.3, with the space character the most represented, having seven occurrences, and the least represented character, such as 't', 'D' and 'T', only have one occurrence.

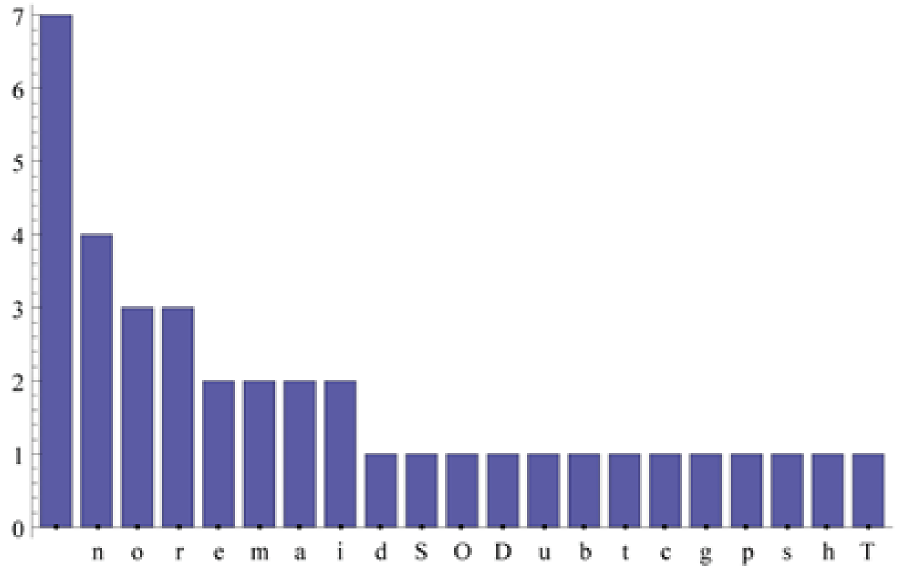


**Figure 4.3:** The frequency histogram of 'This program cannot be run in DOS mode'

A tree is constructed by starting only with leaf nodes. Each leaf node contains the symbol itself and has the weight of its respective frequency. At each stage of the algorithm, the two lowest weight nodes are combined into a parent node whose weight is the sum of its two children's weights. So, leaf nodes 't' and 'D' of weight 1 are made the children of a new internal node of weight 2. The process is then repeated, using the modified set of nodes and node weights. When there is only one new node remaining in the tree, the process terminates. An example tree is shown in Figure 4.4.

The Huffman encoding for a character is simply a traversal of the tree to the leaf-character, where bit '0' represents a left branch, and bit '1' represents a right branch. In our algorithm, each character is represented by the *length of codewords* rather than

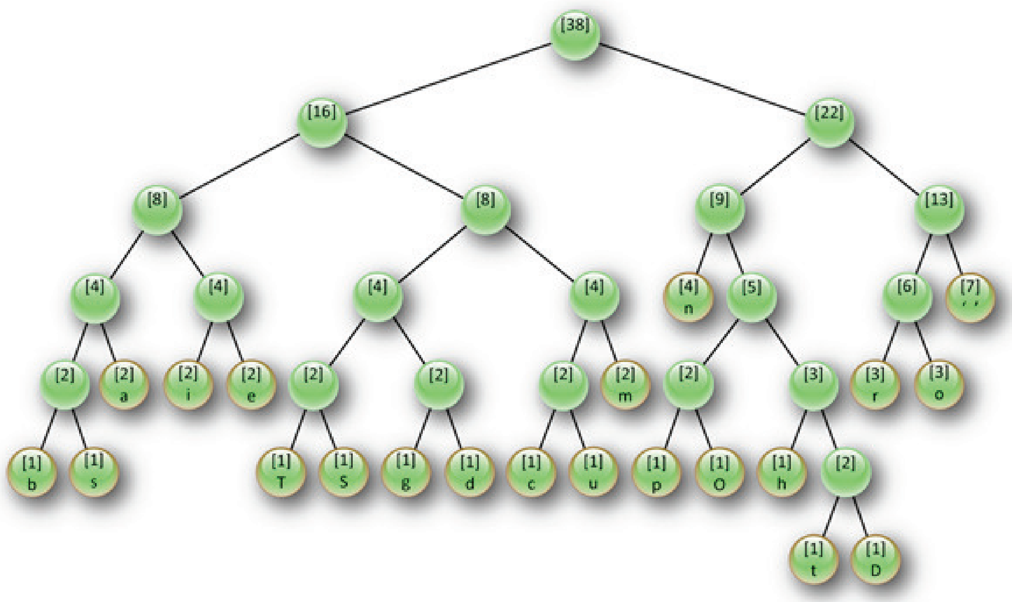


**Figure 4.4:** Huffman tree of 'This program cannot be run in DOS mode'

the codewords itself. That is, the distance from each character to the root is the number of bits needed to encode this character. Table 4.1 lists the final Huffman code for the Huffman tree in Figure 4.4. As shown, the binary encoding for the least represented character, 't', is 101110, and a common character, such as a space, is 111. So, the final encoding for a space character is three bits, whilst a 't' character requires six.

## 4.2 Algorithms

Two versions of the local randomness test algorithm are presented. The first one generates a fixed number of sample points and the second one is a "sliding-window" approach. Both algorithms first require that the Huffman tree be constructed for the entire file, using bytes as symbols. Operating on bytes as symbols appears the most intuitive approach as it causes the tree to remain small, and the byte is the fundamental building block of most executable programs. For the case of a finite alphabet ($2^8$ in our case) and symbols are sorted, there is a linear-time ($O(n)$) method to construct a Huffman tree using two

| Character | Frequency | Codewords | Length of codewords (bits) |
|---|---|---|---|
| space | 7 | 111 | 3 |
| n | 4 | 100 | 3 |
| r | 3 | 1100 | 4 |
| o | 3 | 1101 | 4 |
| a | 2 | 0001 | 4 |
| i | 2 | 0010 | 4 |
| e | 2 | 0011 | 4 |
| m | 2 | 0111 | 4 |
| b | 1 | 00000 | 5 |
| s | 1 | 00001 | 5 |
| T | 1 | 01000 | 5 |
| S | 1 | 01001 | 5 |
| g | 1 | 01010 | 5 |
| d | 1 | 01011 | 5 |
| c | 1 | 01100 | 5 |
| u | 1 | 01101 | 5 |
| p | 1 | 10100 | 5 |
| O | 1 | 10101 | 5 |
| h | 1 | 10110 | 5 |
| t | 1 | 101110 | 6 |
| D | 1 | 101111 | 6 |

**Table 4.1:** Huffman encoding for 'This program cannot be run in DOS mode'.

queues. The time required for file $I/O$ should completely dominate the time required for construction of the Huffman tree.

### 4.2.1 Fixed Sample Count Algorithm

The fixed sample count algorithm sets a fixed number of sample points, equally spaced throughout the file. In order to preserve detail as much as possible, all consecutive windows overlap.

Set the sample count $s$, let a file be denoted by $f = \{b_1, \ldots, b_j, \ldots, b_n\}$ where $b_j$ is the byte at the file position $j$, $b_j \in \{B_0, B_1, \ldots, B_{255}\}$, and $n$ is the file size. Then the file can produce $s$ randomness values, listed as $r_1, r_2, \ldots, r_s$. If the Huffman encoding (length of codewords) for the array of bytes $B_0, B_1, \ldots, B_{255}$ is $e_{B_0}, \ldots, e_{B_{255}}$ respectively, the randomness value $r_i$ in the window $i$ is calculated as the total length of codewords of the corresponding data. That is, for the window positioned at $x$ to $y$ in the file,

$$r_i = \sum_{j=x}^{y} e_{b_j}$$

For example, the previous 'This program cannot be run in DOS mode' string has total 38 bytes. With the sample count set to 4, Figure 4.5 shows that the file is partitioned into four windows. Each window contains 15 bytes which are bytes 1-15, 9-23, 16-30 and 24-38 respectively. In the graph, the length of codewords of each byte is calculated and listed below the corresponding byte, such as 5 bits for 'T'. The randomness value of each window is also calculated. For the first window,

$$r_1 = 5+5+4+5+3+5+4+4+5+4+4+4+3+5+4 = 64.$$

Similarly, $r_2 = 60, r_3 = 56$ and $r_4 = 61$.

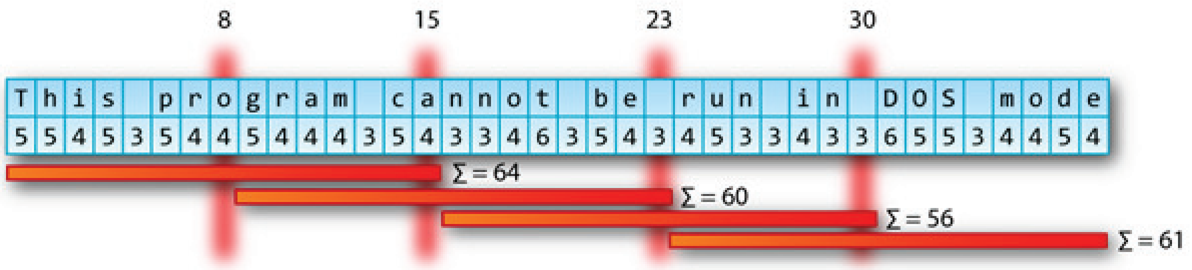


**Figure 4.5:** Illustration of fixed sample count algorithm, where sample count is 4

The fixed sample count algorithm is useful for comparative and storage purposes as it outputs a fixed set of samples for different length files. The visual impact is that the window size depends on the file size, so a big window in a big file can mask detail, while a small window in a small file can over represent detail. Similar features on different sized files appear as if they were dilated or contracted when plotted together.

#### 4.2.1.1 Implementation

As described in Algorithm 4.1, five steps are involved in a fixed sample count version of local randomness test. Firstly, for each file, all bytes are counted and a byte-frequency histogram is built. Secondly, using the global bytes information, the Huffman tree is constructed for the entire file by inserting bytes into the tree in the order of increasing frequency. Then, a "length-encoding" array $e_{B_0}, \ldots, e_{B_{255}}$ is constructed where $B_i$ is the corresponding byte. The entries of the array give the distance to the root of the tree for each element. Thus $e_{B_0}$ gives the distance to the root from the byte $B_0$, $e_{B_1}$ gives the distance to the root from the byte $B_1$, and so on. This distance is also the number of bits needed to encode this byte, in the prefix-free Huffman code. Thirdly, each window's offset needs to be determined as windows overlap. It is calculated as $o_1 = 1, o_{s+2} = n$ and $o_{i+1} =$ round $(\frac{n \times i}{s+1})$ for $i = 1, \ldots, s$. Then, the randomness value $r_i$ in each window is calculated as $\sum_{j=o_i}^{o_{i+2}} e_{b_j}$, i.e. as the total code length of the corresponding data. At the end, the $r_i$ are divided by the difference between the maximum and minimum value, divisor = maximum randomness value - minimum randomness value, so that the new minimum value becomes 0.0 and the new maximum value becomes 1.0. This is because a big window contains more bytes than a small window has, therefore its randomness value (total Huffman encoding) is larger than the one of a small window. With the scaling, files of different sizes can be compared in the same scope.

As mentioned before, it is possible to construct a Huffman tree in linear time, by using, e.g. bucketsort[74]. All other steps are also linear in the length of the file, hence the whole algorithm is linear in the number of total bytes $n$.

```
Data  : The packed file in the form of bytes $b_1, \ldots, b_j, \ldots, b_n$ where $b_j \in$
        $\{B_0, B_1, \ldots, B_{255}\}$ and a sample count $s$
Result: An array of $s$ samples of the randomness, ranging from 0.0 to 1.0
begin
1.    Construct the Huffman tree for the input bytes ;
2.    Construct an array $e_{B_0}, \ldots, e_{B_{255}}$ containing the encoding length for each of
      the input bytes.
3.    Compute sample offsets $o_1, \ldots, o_{s+2}$ as the following: ;
      Set $o_1 \longleftarrow 1$ ;
      Set $o_{s+2} \longleftarrow n$ ;
      for $i$ from 1 to $s$ do
          $o_{i+1} \longleftarrow \text{round}\ (\frac{n \times i}{s+1})$ ;
      endfor
4.    for $i$ from 1 to $s$ do
          Set the randomness value $r_i \longleftarrow \sum_{j=o_i}^{o_{i+2}} e_{b_j}$
      endfor
5.    for $i$ from 1 to $s$ do
          Rescale $r_i$ between 0.0 and 1.0, where $\min(r_i) = 0.0$ and $\max(r_i) = 1.0$
      endfor
end
```

**Algorithm 4.1:** The fixed sample count algorithm: generate fixed number of randomness measurements for a file

### 4.2.2 Sliding Window Algorithm

The sliding window algorithm sets a fixed window size and slides down windows crossing the whole file with a certain skip size. If the skip size is the half of the window size, then all consecutive windows overlap.

Set the window size $w$ and skip size $a$ for a file $f = \{b_1, \ldots, b_j, \ldots, b_n\}$ where $b_j$ is the byte at the file position $j$, $b_j \in \{B_0, B_1, \ldots, B_{255}\}$, and $n$ is the file size, then $\lceil \frac{n-w}{a} \rceil$ randomness values are generated, listed as $r_1, r_2, \ldots, r_{\lceil \frac{n-w}{a} \rceil}$. If the Huffman encoding (length of codewords) for the array of bytes $B_0, B_1, \ldots, B_{255}$ is $e_{B_0}, \ldots, e_{B_{255}}$ respectively, the randomness value $r_i$ in the window $i$ is calculated as the total length of codewords of the corresponding data. That is, for the window positioned at $x$ to $y$ in the file,

$$r_i = \sum_{j=x}^{y} e_{b_j}$$

For example, set the window size 15 and skip size 2 for the same string used before, the file is partitioned into $\lceil \frac{n-w}{a} \rceil = \lceil \frac{38-15}{2} \rceil = 12$ windows. In the graph (see Figure 4.6), the length of codewords of each byte is calculated and listed below the corresponding byte, such as 5 bits for 'T'. The randomness value of each window is also calculated. For the first window,

$$r_1 = 5 + 5 + 4 + 5 + 3 + 5 + 4 + 4 + 5 + 4 + 4 + 4 + 3 + 5 + 4 = 64.$$

For the second window, $r_2 = 60$, and so on.

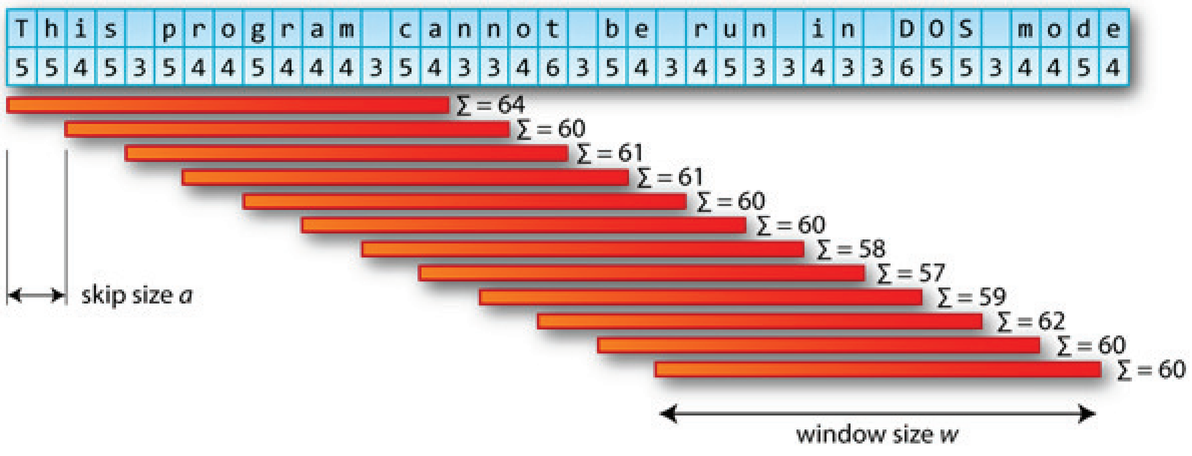


**Figure 4.6:** Illustration of sliding window algorithm where window size is 15 and skip size is 2

This version of the algorithm produces an output length dependent on the input file size. The advantage of this approach is that it does not dilate or contract features if the window size is appropriately set. It can be used to look for features of a specific size, for example cryptographic keys, by setting the window size $w$ to the size of the interesting feature. However, it is less convenient for comparative purposes, and can produce a lot of data if the file is big or the skip size is small.

#### 4.2.2.1 Implementation

There are four steps involved in a sliding window version of local randomness test (see Algorithm 4.2). Firstly, for each file, all bytes are counted and a byte-frequency histogram is built. Secondly, using the global bytes information, the Huffman tree is constructed for the entire file by inserting bytes into the tree in the order of increasing frequency. Thirdly, a "length-encoding" array $e_{B_0}, \ldots, e_{B_{255}}$ is constructed where $B_i$ is the corresponding byte. The entries of the array give the distance to the root of the tree for each element. Thus $e_{B_0}$ gives the distance to the root from the byte $B_0$, $e_{B_1}$ gives the distance to the root from the byte $B_1$, and so on. This distance is also the number of bits needed to encode this byte, in the prefix-free Huffman code. At the end, we set a window size $w$ and a skip size $a$, so that there are a total $\lceil \frac{n-w}{a} \rceil$ windows (indexed by $1, 2, \ldots, \lceil \frac{n-w}{a} \rceil$) for the whole file. The randomness value $r_i$ in each window is calculated as $\sum_{j=a(i-1)+1}^{a(i-1)+w+1} e_{b_j}$, i.e. as the total code length of the corresponding data. At the end, the $r_i$ are scaled so that the minimum value is 0.0 and the maximum value is 1.0. Though windows' sizes are fixed, the rescaling process minimises the difference between the randomness values of different files.

```
Data   : The packed file in the form of bytes $b_1, \ldots, b_j, \ldots, b_n$ where $b_j \in$
         $\{B_0, B_1, \ldots, B_{255}\}$, a window size $w$ and a skip size $a$
Result: An array of $(n-w)/a$ samples of the randomness, ranging from 0.0 to 1.0
begin
1.    Build a byte-frequency histogram for all bytes $B_0, B_1, \ldots, B_{255}$ in the entire
      file ;
2.    Construct the Huffman tree by inserting bytes into the tree in the order of
      their frequency ;
3.    Construct an array $e_{B_0}, \ldots, e_{B_{255}}$ containing the encoding length for each of
      the input bytes.
      for $i$ from 1 to $(n-w)/a$ do
          Set the randomness value $r_i \longleftarrow \sum_{j=a(i-1)+1}^{a(i-1)+w+1} e_{b_j}$ ;
      endfor
4.    for $i$ from 1 to $s$ do
          Rescale $r_i$ between 0.0 and 1.0, where $\min(r_i) = 0.0$ and $\max(r_i) = 1.0$ ;
      endfor
end
```

**Algorithm 4.2:** The sliding window algorithm: generate randomness measurements for a file, output proportional to file length

## 4.3 Experiments and Results

### 4.3.1 Randomness Scanning

#### 4.3.1.1 Data Setup

Experiments have been carried out intensively on six packers with the whole collection of UnxUtils binaries[112]. Six packers used are FSG 2.0, Mew 11, Morphine 2.7, RLPack 1.19, Upack 0.399 and UPX 2.03w. The UnxUtils collection is selected because files in the collection are utilities of reasonable complexity, and since they are command line based, they do not block on user input. This is particularly useful when automating debugging. Furthermore, the collection contains a good diversity of large, medium and small programs. The collection has 116 executable files whose file size range from 3 to 191 KB.

Three steps are involved in the randomness scanning experiment. Firstly, for each packer, each file in the UnxUtils set is packed with this packer. Then for each packed file, randomness values in different parts of the file are measured using both the fixed sample count algorithm and the sliding window algorithm. The final step is to plot the output.

#### 4.3.1.2 Fixed Sample Count Algorithm

In the fixed sample count algorithm, the sample count $s$ is set to 512 in order to balance the effect of over-representing detail of the small file and under-representing detail of the big file.

For illustrating the idea, Figures 4.7-4.13 only show plots of three clean files basename.exe, gsar.exe and logname.exe and their corresponding packed files using the fixed sample count algorithm. Here, both basename.exe and logname.exe are size of 8kb and gsar.exe is 15kb, nearly double size of the other two. Plots of other tested files display similar characters as shown.

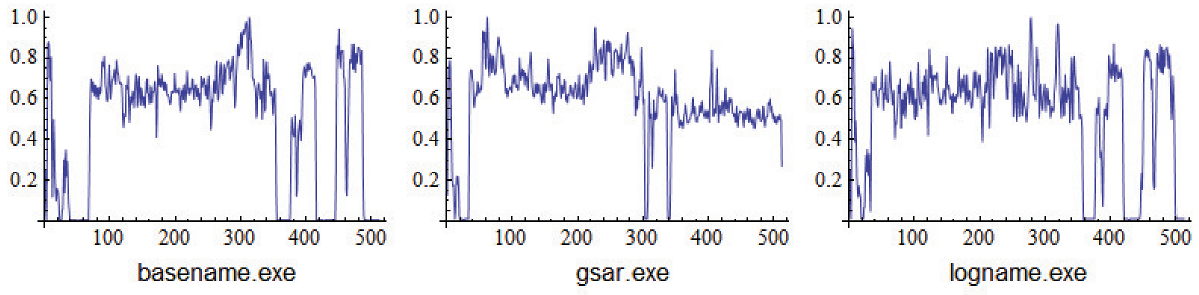


**Figure 4.7:** Fixed sample count randomness scan for unpacked clean files

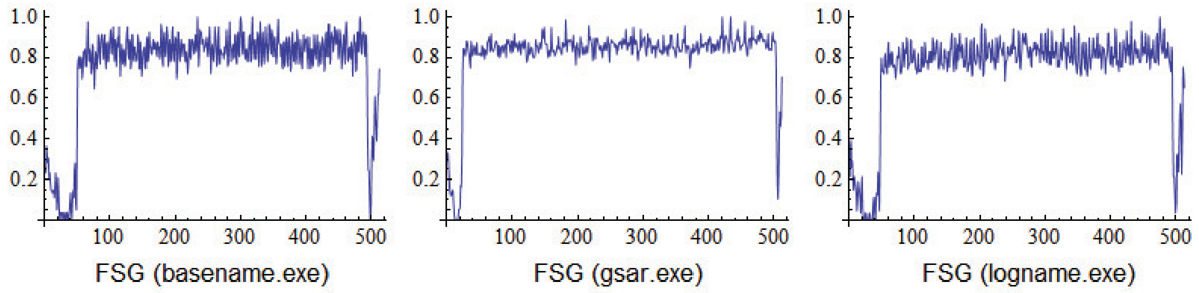


**Figure 4.8:** Fixed sample count randomness scan for FSG 2.0 packed files

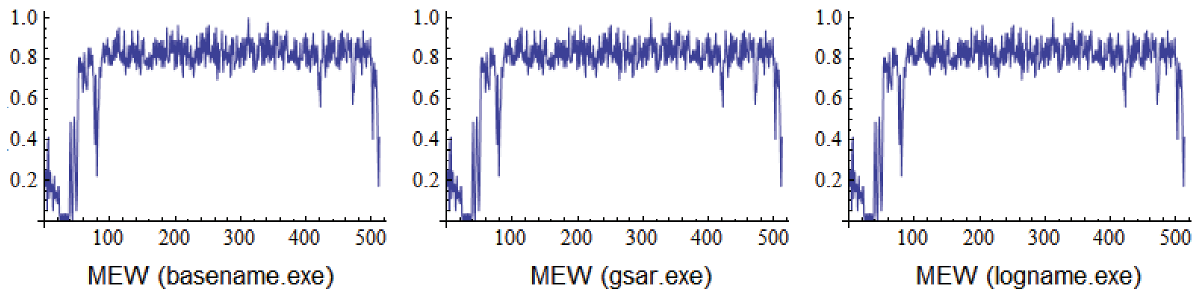


**Figure 4.9:** Fixed sample count randomness scan for Mew 11 packed files

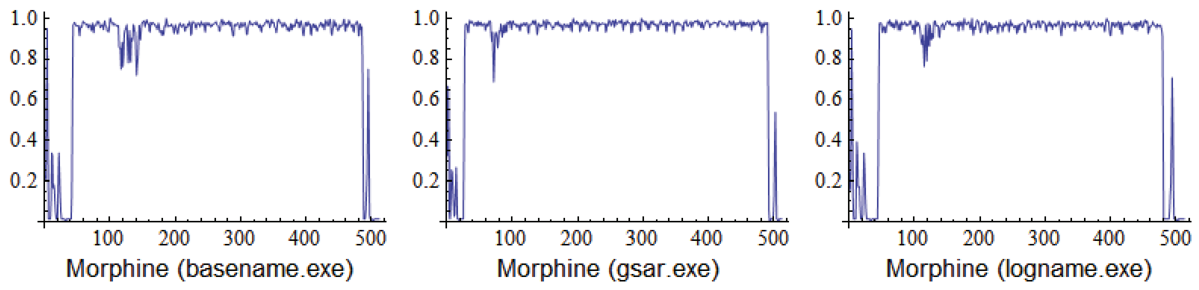


**Figure 4.10:** Fixed sample count randomness scan for Morphine 2.7 packed files

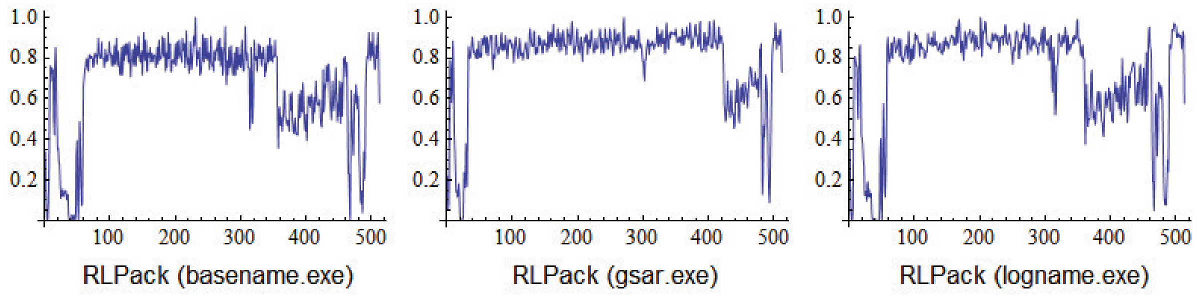


**Figure 4.11:** Fixed sample count randomness scan for RLPack 1.19 packed files

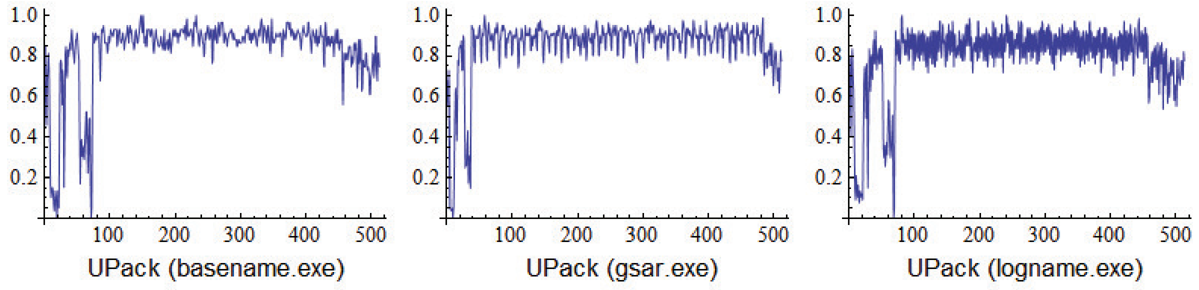


**Figure 4.12:** Fixed sample count randomness scan for UPack 0.399 packed files

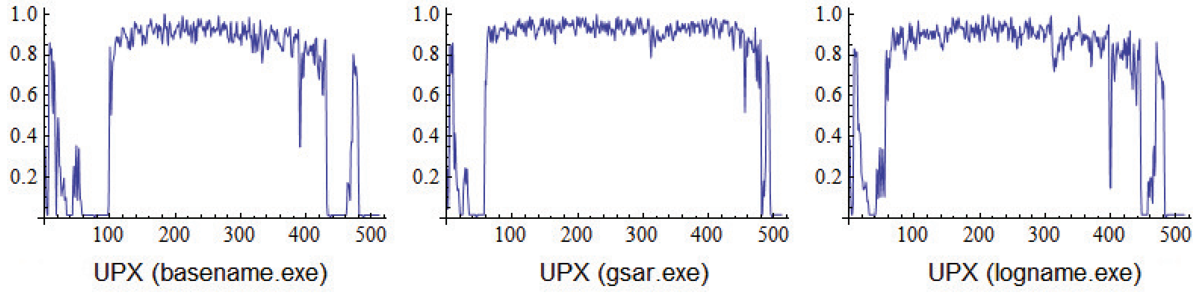


**Figure 4.13:** Fixed sample count randomness scan for UPX 2.03w packed files

The fixed sample count algorithm preserves local detail by showing areas of relatively high and low randomness across the file. Where a plot bottoms out is usually a region filled with zero (0x00) bytes. Compare unpacked files (Figures 4.7) with packed files (Figures 4.8-4.13), it is obvious that all plots of packed files contain large, high, flat segments. These segments tend to indicate highly random data, often indicating compressed or encrypted data. The relatively low randomness segment are usually produced by PE header, code and other resources information.

The size of the original file gsar.exe is nearly double of the size of basename.exe or logname.exe. For most packers, the packed gsar.exe is also bigger than packed basename.exe and logname.exe, e.g. the FSG packed gsar.exe is 8kb while the other two are 4kb. This relationship is reflected in the fixed sample count randomness scanning plots. In the relatively low segments, similar features (such as PE header) are dilated in the small files.

It is also noted that similar plots are obtained from files packed with the same packer, even when compared by eye. It is clear that by using this technique, some packers seem to produce a distinctive plot. This seems to be because compressed data and the code that unpacks it tend to be placed in the same relative location in the packed file, leading to a kind of 'randomness signal' which may be used for packer classification. A detailed discussion will be given in Chapter 5.

#### 4.3.1.3 Sliding Window Algorithm

In the sliding window algorithm, the window size $w$ is set to 32 bytes (256 bits) so that the detail can be reasonably displayed. The skip size $a$ uses 16 which means samples overlap.

Figures 4.14-4.20 show plots of three clean files basename.exe, gsar.exe and logname.exe and their corresponding packed files using the sliding window algorithm.

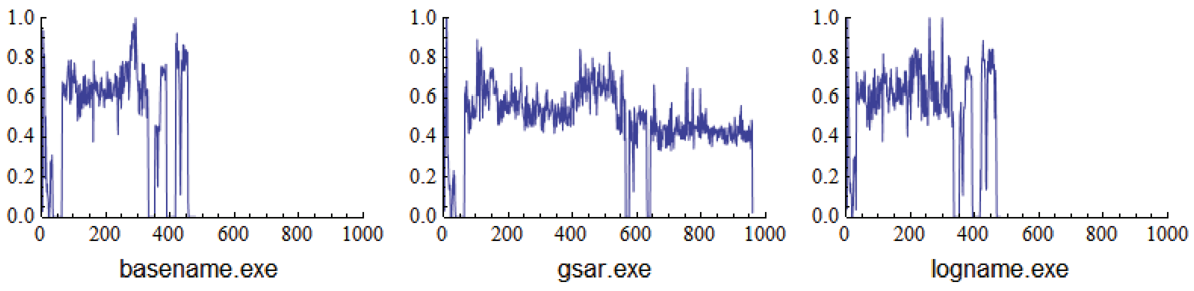


**Figure 4.14:** Sliding window randomness scan for unpacked clean files

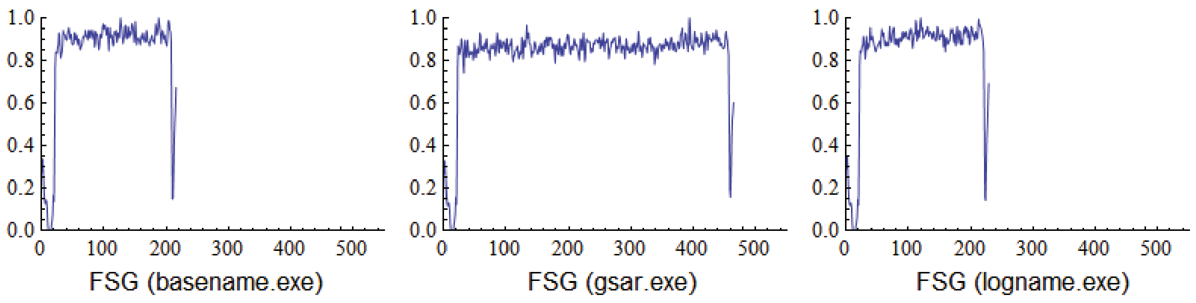


**Figure 4.15:** Sliding window randomness scan for FSG 2.0 packed files

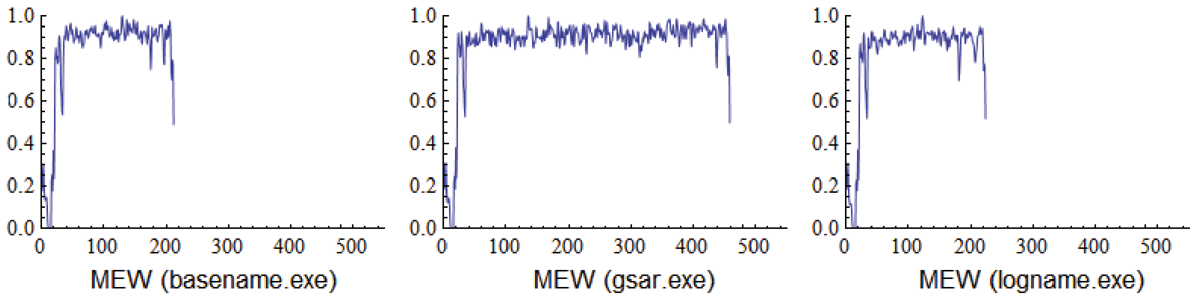


**Figure 4.16:** Sliding window randomness scan for MEW 1.1 packed files

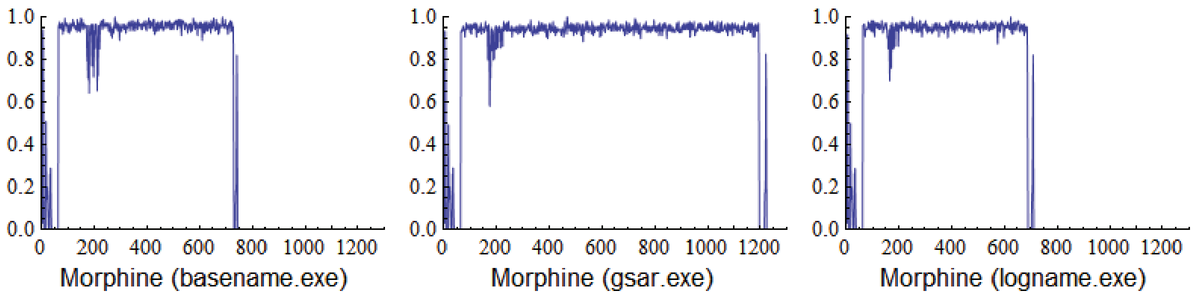


**Figure 4.17:** Sliding window randomness scan for Morphine 2.7 packed files

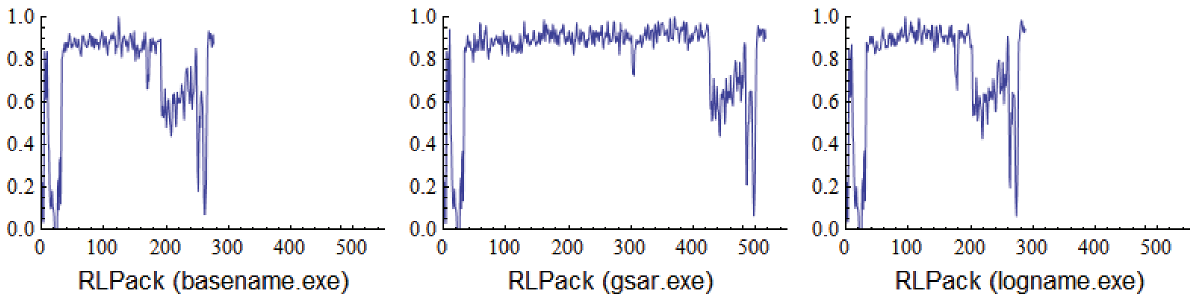


**Figure 4.18:** Sliding window randomness scan for RLPack 1.19 packed files

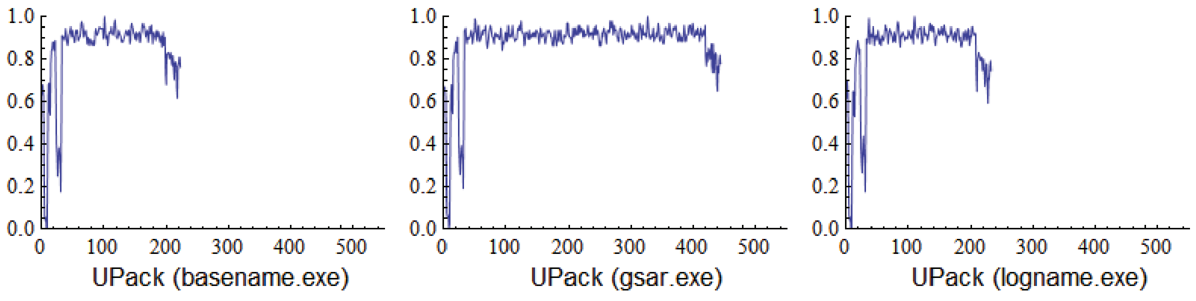


**Figure 4.19:** Sliding window randomness scan for UPack 0.399 packed files

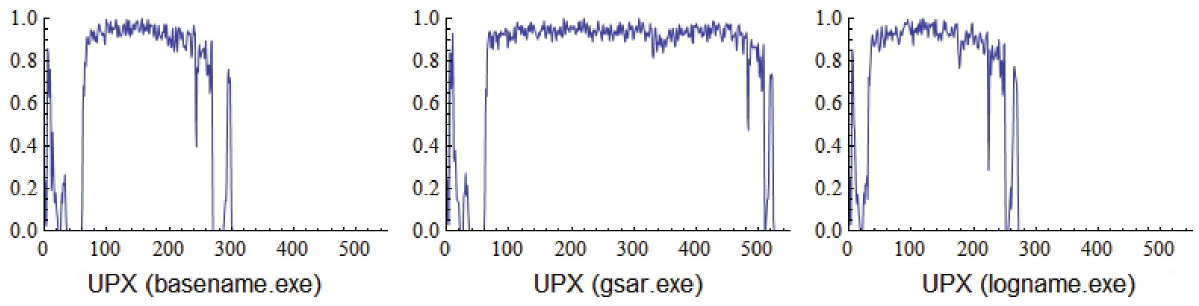


**Figure 4.20:** Sliding window randomness scan for UPX 2.03w packed files

Similar to the fixed sample count algorithm, the sliding window algorithm preserves local detail too. Note also that compressed/encrypted data in the packed file produce high, flat segments in the plot and each packer exhibits a distinctive plot.

The number of outputs in packed gsar.exe is nearly double as the one in packed basename.exe or logname.exe (see Figures 4.15-4.20), this confirms that the outputs of the sliding window algorithm depend on the file size. These plots also show that same feature on different sized files displays in the similar manner in the plot.

### 4.3.2 Unpacking Animation

As the result of the memory change caused by decompression, decryption and memory overwriting, the corresponding file's randomness outputs will change. Therefore, the local randomness test algorithm is also useful to visualize the work of a packed program as it unpacks itself. This is achieved by placing breakpoints on main loops during unpacking, dumping memory, then performing the detail-preserving randomness analysis on the dump.

The dumping tool, Multi-dumper, described here is a C++ plugin for IDA Pro [47]. This tool allows the user to set breakpoints on the loops. When the program is running in an IDA debugger, every time the breakpoint is trigged, the memory of the program is automatically dumped. Interesting loops are identified using the "Hump-and-Dump" method [111] which is based on an ordered address execution histogram. An unpacking (decompression/decryption) activity or a memory copying loop normally involves a massive recurrence. It will generate a “hump” in an ordered address execution histogram and be easily identified.

Once dumps are collected, the local randomness test described before is applied on each dumped file. Using these randomness values, a Mathematica [94] notebook is used to generate an animation of the randomness distribution during unpacking.

Experiments were carried out on five packers, ASPack 2.12, FSG 2.0, MEW 11, Morphine 2.7 and UPX 2.03w. These animations visualize each packer’s unpacking routine. By viewing an unpacking animation, anti-virus researchers can easily get a feel about how a packer is operating. Below are two examples.

Figure 4.21 displays six images captured during a FSG 2.0 unpacking animation. The local randomness test used is the fixed sample point algorithm with sample count 512. The test is run without the re-scaling process as the test is applied on the dumped memory of the same file, i.e., the dumping size is same. The first image is the randomness scanning result when the file firstly loaded into the memory. The high, flat segments of the plot are generated by the compressed data while the bottom area is a big segment of memory filled with zero (0x00) bytes usually created by the loader. The relatively low randomness segments tend to indicate the location of the PE header and codes and other resources information. The last image is the randomness scanning result when the decompression is finished and the other images are the randomness scanning plots of the dumped memory at different stages of the unpacking process. As we can see, during the unpacking, the empty segment’s randomness values change from the lowest to relatively high values. We believe that FSG decompresses the compressed original file into a new segment.

Figure 4.22 displays six images captured during a Morphine 2.7 unpacking animation. The local randomness test used is the fixed sample point algorithm with sample count 512. The test is run without the re-scaling process as the test is applied on the dumped memory of the same file, i.e., the dumping size is same. The first image is the randomness

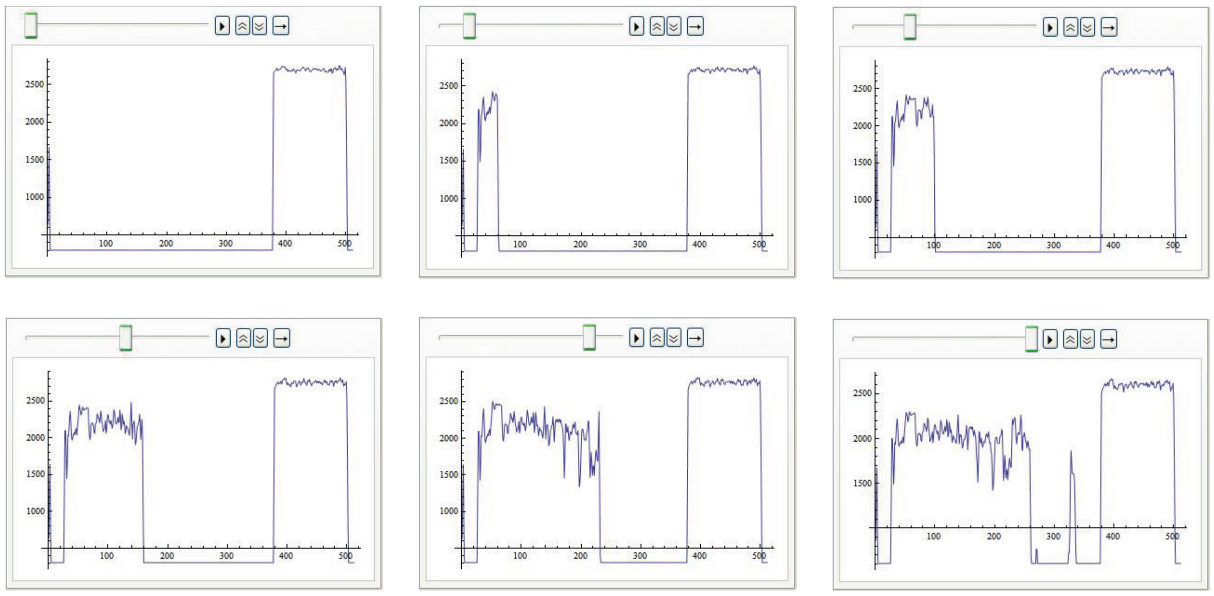


**Figure 4.21:** Snapshots from FSG 2.0 unpacking animation

scanning result when the file firstly loaded into the memory. The high, flat segments of the plot are generated by the encrypted data while the bottom segment indicates memory filled with zero (0x00) bytes usually created by the loader. The relatively low randomness segments tend to indicate PE header and codes and other resources information. The last image is the randomness scanning result when the decryption is finished and the other images are the randomness scanning plots of the dumped memory at different stages of the unpacking process. As we can see, during the unpacking, the highly random data is overwritten by the relatively low randomness data in the backwards manner. We believe that Morphine decrypts the encrypted original file backwards and writes them back to their original addresses.

## 4.4 Summary and Discussion

This chapter presents a novel local randomness test with two versions, namely fixed sample count and sliding window algorithm. Both algorithms are fast and are able to preserve local detail of a packer and distinguish highly random data from relativly low random-

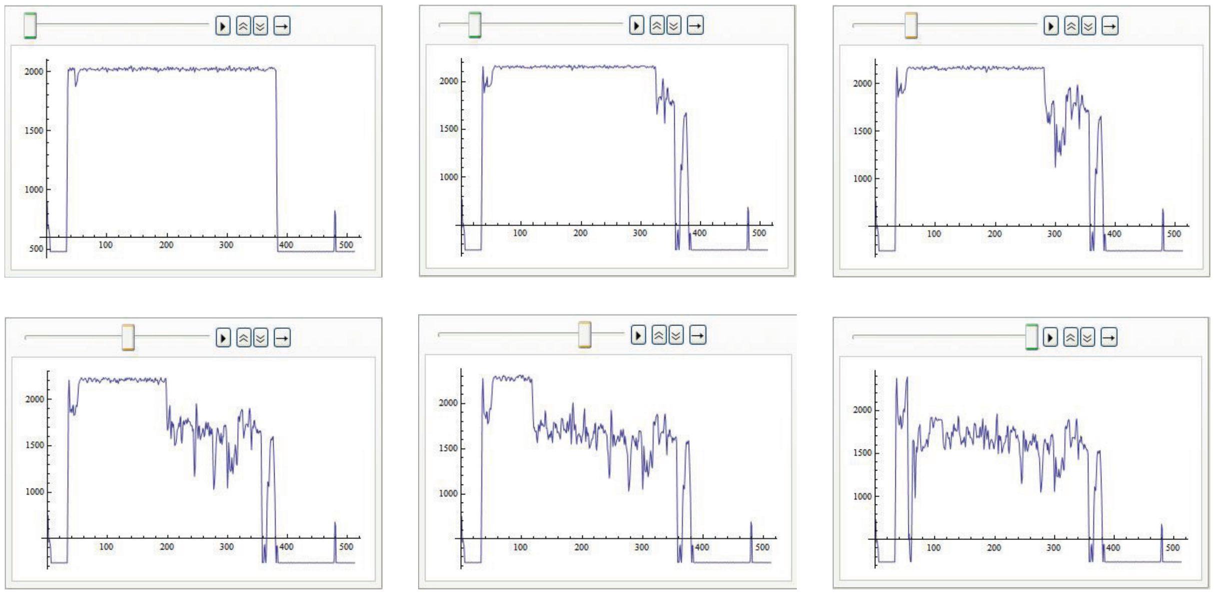


**Figure 4.22:** Snapshots from Morphine 2.7 unpacking animation

ness data. Compressed/encrypted data usually generates high points on the randomness scanning plot and can be easily detected. This is useful for packer detection and packer analysis. Moreover, each packer displays randomness plot in a distinctive way, therefore the randomness outputs can be useful for comparative measurements.

While they have common features, the two algorithms also differ significantly. The fixed sample count algorithm produces the same length of outputs thereby enabling easy comparison. Using the sliding window algorithms results in same size windows and features remain in different-sized files.

What remains to do is a comprehensive comparison between the two algorithms in terms of effective feature extraction. Such a comparison is important to decide the best feature that would be recommended for practical use in packer classification. This is left to Chapter 5, when the classification approach have been described fully. In that chapter, the two versions of local randomness test are embedded into the packer classification system in order to compare the feature extraction effectiveness of the algorithms.

# Chapter 5

# Packer Classification Using Randomness Profile

Packer classification is an important step in malware analysis, if it can be designed so that it gives a reliable approach to detect packed files. In this chapter, we provides a fast, automatic and effective method for visually assessing which packer used to pack the file.

The previous chapter describes a randomness test that measures the local randomness level of a packed file. This method is able to distinguish highly random data (compressed or encrypted data) from data with low randomness (such as header, code, etc.). It also presents a kind of an extracted 'randomness signal' or profile describing the packer that may be used for the purpose of packer classification. This Chapter investigates whether this randomness profile contains sufficient information for accurately classifying packers.

The investigation begins by extracting feature vectors from packers' randomness profiles generated by the local randomness test. The two versions of the algorithms introduced in Chapter 4, both the fixed sample count algorithm and the sliding window algorithm, are evaluated in this chapter and the algorithm parameters which give the best performance are determined. In order to classify packers using the extracted randomness features, two different classification approaches are employed. The first one is the randomness signature approach. The packer signatures are automatically generated by calculating the centroid vectors (i.e., the average) of the training data. Then the distances between the test file and the candidate packer signatures are measured. The packer with the shortest distance is assigned to the test file. The second approach applies pattern recognition techniques which build a classification model for each packer and then apply the model to classify the test file based on its features.

Experimental results demonstrate that the pattern recognition approach outperforms the randomness signature approach. All six evaluated pattern classification algorithms achieve extremely high effectiveness ($> 99\%$). The experiments also confirm that the randomness profile used in the system is a very strong feature for packer classification.

## 5.1 Introduction

The objective of packer classification is to quickly detect and identify the packer, allowing AV researchers to easily and correctly unpack the file and retrieve original payload for further malware detection and analysis. An efficient and effective packer classification system can yield benefits to both back-end AV researchers and real time anti-malware engines, running live on the client machine

The traditional packer classification approach is mainly based on matching the packer's signature. The signature is typically a unique set of bytes which occur in a specific packer. In this approach, a packer signature needs to be created for each packer, and even for each variant of the packer. An incoming packed file is checked against a packer signature file/database which contains the signatures of different packers. If there is an exact match, then the packer is considered to be identified.

The main disadvantage of the traditional signature scanning approach is that the signatures obtained from byte sequences are not flexible enough to cover the carefully crafted variants of old packers. Recently, there has been a dramatic increase in the number of new packers and variants of existing ones combined with packers employing increasingly sophisticated anti-unpacker tricks and obfuscation methods (see Section 2.3.1.2). The large diversity of packers, and the number of different variants of each packer, severely undermine the effectiveness of classical signature based detection.

Moreover, the traditional signature scanning approach is expensive as the signature detection and updating need to be performed accurately by AV experts. As the number of packers and their complexity increase, more human effort is needed and therefore more time and cost are required.

As the number of new packers keeps growing, *a reliable computer-based packer classification scheme would simplify the identification and characterising process and reduce cost significantly.* Here, an automatically generated randomness profile based scheme is proposed as a replacement for the inefficient approach of using manually created signatures, which is the current approach, to the packer classification problem.

Figure 5.1 shows the architecture of the proposed classification system. Firstly, it extracts a unique feature set from each packed file's randomness profile. Then it passes the extracted feature to a discriminant calculator which measures the difference between the sample file and a packer's signal. If the chosen distance measure is effective, the correct packer will be identified.

## 5.2 Feature Extraction

Feature extraction retrieves the 'characteristic' of the packer which represents the packer in a distinctive way. This permits the comparison of the packed file with the candidate packers.

The randomness test described in Chapter 4 measures the amount of "randomness" in different parts of a sample executable program. The encouraging results obtained show that the novel randomness test preserves local detail by showing areas of relatively high and low randomness across the file. It was also noted that the randomness distribution of each packer family exhibits a distinctive pattern (see Figure 4.8-4.13 and Figure 4.15-4.20).

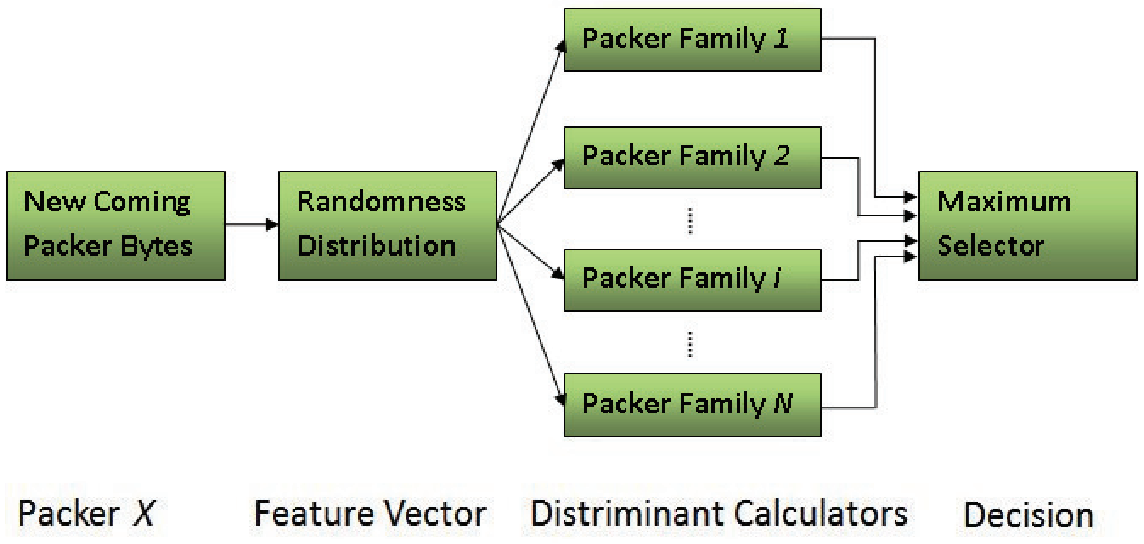


**Figure 5.1:** A computer-based packer classification framework

In order to only use significant features in a packer classification system, scaling and pruning processes have been applied on the packer's randomness profile to extract useful data with low randomness. These two processes are explained below.

### 5.2.1 Scaling

The fixed sample count and sliding window algorithms developed in Chapter 4 rescale outputs at the last step (see Algorithm 4.1 & 4.2). The scaling process is a linear transformation that normalises the randomness outputs of one file by the same scale factor. In this case, the scale factor is the difference between the maximum and minimum value, so that all outputs are in the range between 0.0 and 1.0. The intuition is that scaling will minimize the effect caused by the file sizes. However, whether this step is necessary for feature extraction is still questionable. Thus, we investigate the scaling process below.

Figure 5.2 and 5.3 show that the scaling process is important in both the fixed sample count and sliding window algorithms for comparative purposes. In the fixed sample count algorithm, though the number of outputs ($x$-axis) are same, the randomness values of each window ($y$-axis) vary with the input file size before the scaling process (Figure 5.2 (a)).

The larger the size of the file, the bigger the size of each randomness test window and the higher the value of the randomness. The scaling process standardizes the randomness outputs along the y-axis though division (where we use the divisor = (maximum randomness value - minimum randomness value)). This brings the graphs into the same $y$-axis range $(0.1 - 1.0)$ while retains the shape of the pattern of the randomness distribution (Figure 5.2 (b)).

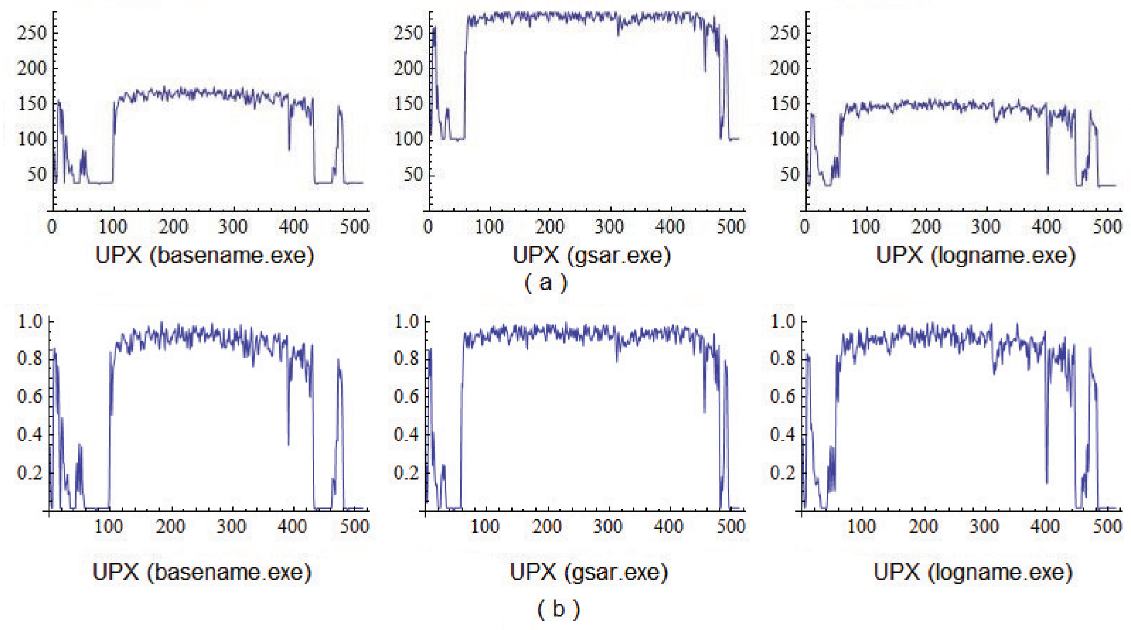


**Figure 5.2:** The importance of scaling in the fixed sample count algorithm. (a) Fixed sample count algorithm before scaling. (b) Fixed sample count algorithm after scaling.

In the sliding window algorithm, we have observed that without scaling, though the shapes of the randomness distribution pattern of same packer look similar, the randomness values along the $y$-axis are different. Using the most popular packer UPX for demonstration, as in Figure 5.3(a), the range of high randomness values of UPX packed basename.exe and gsar.exe file are very close, but the lowest value in basename.exe is around 50 while in gsar.exe is around 100. The scaling process in the sliding window algorithm retains the shape of the pattern of the randomness distribution by standardizing the randomness outputs along the y-axis though division (with divisor = (maximum randomness value -

minimum randomness value)). After scaling (Figure 5.3(b)), the randomness values are in the same $y$-axis range $(0.0 - 1.0)$. Comparing to graph (a), it is clear that corresponding randomness values, especially in the low randomness sections, are closer for different files in graph (b).

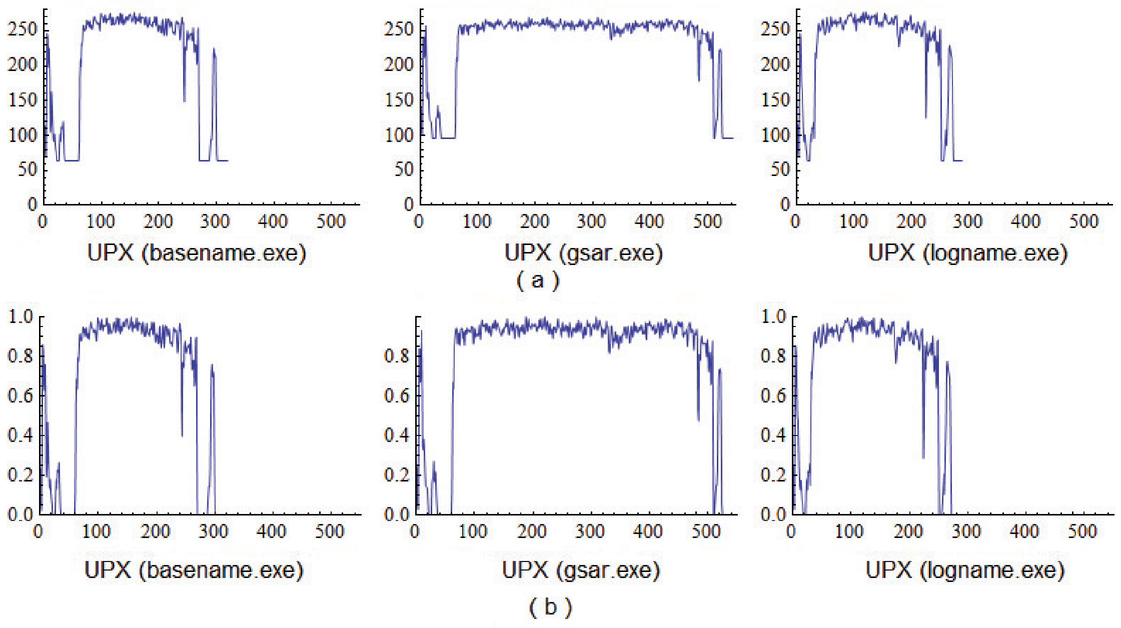


**Figure 5.3:** The importance of scaling in the sliding window algorithm. (a) Sliding window algorithm before rescaling. (b) Sliding window algorithm after rescaling.

The remaining problem is that the total number of randomness values still vary along $x$-axis. As we can see, the differences among files, are mainly with respect to the length of the highly random sections, which is determined by the size of the original unpacked file. The bigger the original file, the larger the highly random section, and the higher the total number of values. A process that solves this problem, by removing some values along x-axis, will be discussed in the next section (Section 5.2.2).

### 5.2.2 Pruning

In the sliding window algorithm, the number of the randomness outputs is dependent on the input file size. In order to create feature vectors of standardized size, these outputs need to be pruned.

Pruning is a post-processing method that extracts an $n$-dimensional vector from an input. The method simplifies comparison between input samples of different lengths by standardizing all vectors to the same length. For distinguishing packers, the low randomness segments produced by the unpacking code are generally more significant than the high randomness segments, which corresponding to encrypted or compressed data. Therefore pruning intends to retain segments of the data with low randomness and eliminate those with high randomness, i.e., keeps the unpacking code and removes the packed data.

We propose four different types of pruning, namely 'First', 'Smallest', 'Ordered smallest' and 'Trunk'. For a list of $i$ values:

**First:** retrieves the first $n$ values from the output.

**Smallest:** firstly sorts the output, then gets the $n$ smallest values.

**Ordered smallest:** firstly sorts the output, then gets the $n$ smallest values and lists them *in their original order.*

**Trunk:** removes the middle part of the output. In other words, it keeps $n/2$ values from the beginning and $n/2$ values from the end of the output.

Consider the input data of ten values, $I_{10} = 1, 3, 4, 9, 8, 10, 6, 7, 2, 5$, the pruning results of the First, Smallest, Ordered smallest and Trunk pruning heuristics are listed as $O_f, O_s, O_o, O_t$ respectively. If the pruned data has $n = 6$ values, then

$$\begin{aligned} O_f &= 1, 3, 4, 9, 8, 10; \\ O_s &= 1, 2, 3, 4, 5, 6; \\ O_o &= 1, 3, 4, 6, 2, 5; \text{ and} \\ O_t &= 1, 3, 4, 7, 2, 5. \end{aligned}$$

## 5.3 Classification

The scaling and pruning processes introduced in the previous section map the randomness profile of the packed file into an $n-$dimensional vector space in which various classification techniques can be applied. Classification refers to the method with which the packed file is examined and assigned to a predefined class. In this thesis, randomness signature scanning technique and pattern recognition techniques are employed to classify packers. The first one creates a signature for each packer and then uses the distances between the test file and the packer signatures to identify the packer. The second one constructs a classification model for each packer and then applies the model to classify the test file. Section 5.3.1 and 5.3.2 describe these two approaches in detail.

### 5.3.1 Randomness Signature Scanning

The randomness signature scanning developed is based on matching the file with the packer's *randomness signature*. Instead of the traditional signature which is usually a byte sequence manually defined by AV researchers, the *randomness signature* is a sequence of significant randomness values automatically generated from a set of training data by calculating their average value. For an incoming packed file, it is checked against a packer signature collection which contains the signatures of different packers. The check is not trying to find an exact match, instead, the distance between the file and each signature is calculated. The packer with the shortest distance is assigned to be a match.

#### 5.3.1.1 Randomness Signature

In randomness signature scanning method, a file is represented as an $n$-dimensional vector of randomness values

$$f = \{r_{f1}, r_{f2}, \ldots, r_{fn}\}$$

where $r_{fi}$ is the randomness value at position $i$ in the file.

For each family, we generate a packer's signature, which is represented as an $n$-dimensional vector

$$s = \{r_{s1}, r_{s2}, \dots, r_{sn}\},$$

to use in comparing against the entire dataset as a means of classification. We obtain this signature by calculating each term $r_{si}$ as *the average value of a set of training data which are packed with this packer.* For a set of $N$ training files $F = \{f_1, \dots, f_j, \dots, f_N\}$, where $f_j$ is a single file, each value in the signature is computed as follows:

$$r_{si} = \frac{\sum_{j=1}^{N} r_{f_j i}}{N}. \tag{5.1}$$

#### 5.3.1.2 Distance Measure

The main component in the randomness signature scanning method is the distance measure. A distance measure numerically estimates how closely each test file matches the given packer signature. If the measure is effective, the signature of the packer that was used to pack the file will be the closest to the test file profile and the packer will be identified. For example, if the distances between the file and the signature of packer A, packer B, packer C are 2, 1, 3 respectively. Then the file will be identified having been packed with packer B. For an unknown packer, the randomness profile of the file will not be close to any signature, i.e., the lowest distance value between the file and all signatures will be above a certain threshold.

A wide range of distance measures have been suggested and experimented with. Here, two commonly used measures, namely Euclidean distance and Cosine distance measures, are evaluated. In both measures, an 'entropy' (randomness) file is represented as a vector $f$ and a packer's signature is represented as $s$ as described above.

#### 5.3.1.2.1 Euclidean Distance Measure

The Euclidean distance measures the distance between each point of the file vector $f$ and the signature vector $s$, The smaller the distance between two vectors, the greater the presumed similarity between the file and the packer. It is defined as follows:

$$d(f, s) \quad = \quad \sqrt{\sum_{i=1}^{n} (r_{fi} - r_{si})^2} \tag{5.2}$$

In a packer classification system, the feature vector $f$ belongs to the class $s$ with the lowest distance value. For comparative purpose, it is possible to use monotone functions of the distance. Therefore, in the Euclidean distance, the process that computes the square root of the sum of squares can be ignored for efficient computing. In other words, the Equation 5.2 can be replaced by:

$$d(f, s) = (r_{f1} - r_{s1})^2 + (r_{f2} - r_{s2})^2 + \cdots + (r_{fn} - r_{sn})^2, \tag{5.3}$$

with no change in performance.

Algorithm 5.1 describes the implementation of the Euclidean distance measure. Firstly, an accumulator is created for each packer signature in the signature collection. The initial value is set to zero. Then, for each packer, the randomness vector of the testing file and the packer signature are retrieved and processed. In other words, at each vector point, the corresponding accumulator is updated with the new randomness contribution. The contribution is calculated as the difference between the file randomness value and the signature randomness value, $(r_{fi} - r_{si})^2$. Once all of the vector points have been processed, the accumulator is set. Finally, the packer with the lowest accumulator value is chosen. One efficient way to sort the distances is by using merge sort [74].

```
Data : Packed file $f$ and packer signature collection $S$
Result: Packer name with shortest Euclidean distance
begin
1.  foreach $s \in S$ do
        Set accumulator $A_s \longleftarrow 0$ ;
    end
2.  foreach $s \in S$ do
        Retrieve the file vector $f = \{r_{f1}, r_{f2}, \ldots, r_{fn}\}$ ;
        Retrieve the signature vector $s = \{r_{s1}, r_{s2}, \ldots, r_{sn}\}$ ;
        foreach pointer $i$ in the vector do
            Set $A_s \longleftarrow A_s + (r_{f1} - r_{s1})^2$ ;
        end
    end
3.  Identify the lowest distance values ;
    Return corresponding packer name ;
end
```

**Algorithm 5.1:** Implementation of the Euclidean distance measure

#### 5.3.1.2.2 Cosine Distance Measure

The Cosine distance algorithm measures the angle between two vectors $f$ and $s$. The smaller the angle, the greater the presumed similarity between the file and the packer. It is represented as follows:

$$\begin{aligned} c(f,s) &= \arccos\left(\frac{f \cdot s}{|f||s|}\right) \\ &= \arccos\left(\frac{\sum_{i=1}^{n}(r_{fi} \cdot r_{si})}{\sqrt{\sum_{i=1}^{n} r_{fi}^2}\sqrt{\sum_{i=1}^{n} r_{si}^2}}\right) \end{aligned} \tag{5.4}$$

The Cosine distance is computed as the inner product of the two vectors, normalized (divided) by the product of the vector lengths. As the file vector length, $\sqrt{\sum_{i=1}^{n} r_{fi}^2}$, is same for all packers, this normalization is usually ignored in the calculation. Therefore, in Equation 5.4, the inner product of the file vector and signature vector is only normalized by the packer signature length, $\sqrt{\sum_{i=1}^{n} r_{si}^2}$. That is, we can use:

$$c(f, s) = \arccos\left(\frac{\sum_{i=1}^{n}(r_{fi} \cdot r_{si})}{\sqrt{\sum_{i=1}^{n} r_{si}^2}}\right). \tag{5.5}$$

As discussed in Section 5.2.1, both randomness values, $r_{fi}$ and $r_{si}$, are in the range between 0.0 and 1.0, so

$$\frac{\sum_{i=1}^{n}(r_{fi} \cdot r_{si})}{\sqrt{\sum_{i=1}^{n} r_{si}^2}} \geq 0 \text{ and } c(f, s) = \begin{cases} [0, \dfrac{\pi}{2}] \\ [\dfrac{3\pi}{2}, 2\pi] \end{cases} .$$

In the Cosine distance algorithm, we measure the angle between two vectors. Therefore, $c(f, s)$ can be simplified in the range between 0 and $\pi/2$, where the cosine function is monotone. For comparative purpose, the function arccos can further be ignored in the calculation. Then, the problem becomes calculating the Cosine similarity, the cosine of the angle (see Equation 5.6). The larger this cosine value, the smaller the angle between two vectors, and the more likely the file is packed with this packer, so we can use:

$$c(f, s) = \frac{\sum_{i=1}^{n}(r_{fi} \cdot r_{si})}{\sqrt{\sum_{i=1}^{n} r_{si}^2}} \tag{5.6}$$

Therefore, the implementation of the Cosine distance measure can be simplified as described in Algorithm 5.2. Firstly, an accumulator is created for each packer signature in the signature collection. The initial value is set to zero. The length of the signature vector is also initialized to zero. Then, at each point, the randomness vector of the test file and the packer signature are retrieved and processed. In other words, for each vector point, the corresponding accumulator is updated with the new randomness contribution. The contribution is calculated as the product of the file randomness value and the packer randomness value $r_{fi} \cdot r_{si}$. The signature length is also updated. Once all of the vector points have been processed, each accumulator is then normalized by the signature vector length $l_s$. Finally, the packer with the highest accumulator value is chosen. One efficient way to sort the distances is by using merge sort [74].

```
Data  : Packed file $f$ and packer signature collection $S$
Result: Packer name with shortest cosine distance
begin
1.  foreach $s \in S$ do
        Set accumulator $A_s \longleftarrow 0$ ;
        Set signature length $l_s \longleftarrow 0$ ;
    end
2.  foreach $s \in S$ do
        Retrieve the file vector $f = \{r_{f1}, r_{f2}, \ldots, r_{fn}\}$ ;
        Retrieve the signature vector $s = \{r_{s1}, r_{s2}, \ldots, r_{sn}\}$ ;
        foreach pointer $i$ in the vector do
            Set $A_s \longleftarrow A_s + r_{f1} \cdot r_{s1}$ ;
            Set $l_s \longleftarrow l_s + r_{si}^2$ ;
        end
    end
3.  foreach non-zero accumulator $A_s$ do
        Calculate cosine distance $c_{f,s} \longleftarrow \frac{A_s}{\sqrt{l_s}}$ ;
    end
4.  Identify the highest distance values ;
    Return corresponding packer name ;
end
```

**Algorithm 5.2:** Implementation of the Cosine distance measure

### 5.3.2 Pattern Recognition Techniques

Pattern recognition techniques have been employed by researchers to effectively detect malware (Section 3.2.2). This thesis incorporates the pattern recognition techniques for packer classification. Experimental results of randomness test described in Chapter 4 demonstrate that each packer displays a unique pattern of its randomness profile, especially the low randomness part. This suggests that it is worthwhile to investigate the possibility of classifying packer though pattern recognition techniques.

Using the pattern recognition techniques, a file is represented as an $n$-dimensional feature vector of randomness values

$$f = \{r_{f1}, r_{f2}, \ldots, r_{fn}\}$$

where $r_{fi}$ is the randomness value at position $i$ in the file.

For a set of training data, a classifier employs a learning algorithm to build a classification model that maps each feature vector into one of the predefined class. For an incoming file, the same classifier applies the model to classify the file based on its feature vector.

Various classifiers have been developed and proved effective on certain data sets [119]. In this thesis, four classifiers are evaluated. They are Naive Bayes, Sequential Minimal Optimization, $k-$Nearest Neighbor and Best-first Decision Tree. These classifiers are selected since they are relatively fast. This is very important for a client-side AV scanner which needs to scan millions of files in a short user-tolerable time frame. The detail of these classifiers are described in Section 3.2.1.

## 5.4 Experiments and Results

### 5.4.1 Data Sets

Experiments have been intensively carried out on three types of data set, namely packed clean data set, malware sample data set and mixed data set. The packed clean data set consists of purely packed clean files. The malware sample data set contains only real malware samples which have reliable predefined class. The mixed data set have both packed clean files and real malware samples. Packers used in this data set are mixed with low complex packers and sophisticated packers. Below are the detail description of these three data sets.

#### 5.4.1.1 Packed Clean Data Set

The packed clean data set is used in the initial experiments to determine the feature vectors. The data set comprises of a total of 708 packed clean files of six packers (see Figure 5.1). For each packer, each file in the UnxUtils binaries [112] is packed with this packer. The UnxUtils collection is chosen because files in the collection have reasonable complexity and are all command line based. This is particularly useful when automating debugging since they do not block on user input. The selected collection contains 118 executable files whose file size ranged from 3 to 1058 KB, though most files (116 out of 118) are in the range of 3 – 191 KB. Six packers used are FSG 2.0, Mew 11, Morphine 2.7, RLPack 1.19, UPack 0.399 and UPX 2.03w.

| Packer Name | Number of files |
|---|---|
| FSG 2.0 | 118 |
| Mew 11 | 118 |
| Morphine 2.7 | 118 |
| RLPack 1.19 | 118 |
| UPack 0.399 | 118 |
| UPX 2.03w | 118 |

**Table 5.1:** Data set one: packed clean files.

#### 5.4.1.2 Malware Sample Sets

##### 5.4.1.2.1 Sample Preparation

All malware samples have been prepared by an independent third party, Computer Associates (CA) Threat Management Team in Melbourne, Australia. To construct the data set, real malware downloaded over January and February 2009 by CA are collected. Each file is scanned by three AV scanners for packer labelling. These three scanners are Microsoft (Microsoft Malware Protection, version 1.1.1391.0), Kaspersky (Kaspersky Anti-Virus for DOS32, version 3.0) and CA (CA's internal tool vetrs.exe, version 31). In addition, CA's VET engine and anti-virus Arclib Archive Library are used to determine whether the file can be unpacked. Files that reported by Microsoft, Kaspersky or CA, or that can be unpacked by either the VET engine or Arclib are identified as packed file. As a result of this method, we got a total of 103,392 packed files. The top five packers detected are UPX, ASPACK, FSG, UPACK and NSIS installer. They comprise a total of 91.35% of packed files.

For the collection of 103,392 packed files, each file is assigned as having been packed by one packer. This is done by combining the packer scanning results of all three scanners. The packer name is set if it is identified by any scanner and is 100% confirmed if it is identified by all three scanners. If there is conflicting information, the result taken is the one given by the two scanners which agree. If all three scanners disagree, PEiD is further applied to identify packer. PEiD is a byte-signature based packer scanner [51] which is supported by a large number of packer signatures. However, it is so popular that many packers start to use fake signatures to hide from PEiD detection. Therefore, PEiD's scanning results are not reliable and are only used by us for confirming information.

All scanners contain a different packer signature schema. For some packers, some scanners might only provide the packer family name without the version information. When collecting the version information, if any scanner obtains the version detail, this information is used. If there is no version information, PEiD is used to retrieve the version detail.

Table 5.2 lists four packer detection results extracted from our database. The first file is detected as Aspack by all three scanners. PEiD also confirms the packer. Therefore, it belongs to the Aspack family. Though Kaspersky doesn't provide the version information, all other scanners indicate it is Aspack 2.12. The second file is detected as Aspack by CA and Kaspersky, and CA provides its version number,namely version 2.0, while Kaspersky doesn't. So PEiD is further used to confirm that it is Aspack 2.0. CA identifies the third file as PC Shrinker while the other two do not have any scanning result. In this case, PEiD is used. It not only confirms the packer family, but also gives out the version 0.71. In the last example, there is information conflict between the scanning results. CA says Petite 2.1 and Microsoft gives Petite 2.3. Again, PEiD is used. Its result is Petite 2.1 or 2.2. All results are adjusted and Petite 2.1 is finally assigned to this file.

| No | CA | Microsoft | Kaspersky | PEiD | Family | Detail |
|---|---|---|---|---|---|---|
| 1 | ASPack 2.12 | ASPack v2.12 | ASPack | ASPack v2.12 | ASPack | ASPack 2.12 |
| 2 | ASPack 2.0 | NULL | ASPack | ASPack v2.001 | ASPack | ASPack 2.0 |
| 3 | PC Shrinker | NULL | NULL | PC Shrinker v0.71 | PC Shrinker | PC Shrinker 0.71 |
| 4 | Petite 2.1 | Petite 2.3 | NULL | Petite v2.1 (2) | Petite | Petite 2.1 |

**Table 5.2:** Determination of packer name for malware samples.

#### 5.4.1.2.2 Malware Sample Data Set

As discussed in the previous section, the packer names assigned to the samples are not 100% precise, especially when there is conflicting information between the scanning results from different scanners. In order to get a reliable training data set for the packer classification experiment, two criteria are used to select the packers and files in the malware sample set. These two constraints are:

- *Only confirmed cases are used.* In other words, the file's packer name has been identified as same by all three scanners, e.g., the first file in Table 5.2.
- *Only packers with a sufficient number of confirmed cases are chosen.* In this thesis, each packer should have more than 100 confirmed packed files to be chosen.

According to the selection conditions above, 6 packers of 17,336 files, whose file sizes ranging from $2 - 6880KB$, are chosen from the sample collection described in Section 5.4.1.2.1. The details of the set is presented in Table 5.3. As the top packer in the collection, UPX has 39,799 confirmed packed files. However, to balance the distribution of the sample set, only 9,931 randomly selected samples from these files are used. Note that each packer contains samples with different versions. For example, packer NsPack has cross versions of 2.x, 2.9, 3.4, 3.5, 3.6 and 3.7.

| **Packer** | **Versions** | **Total Number** |
|---|---|---|
| FSG | 1.33 and 2.0 | 5,105 |
| NSPACK | 2.x, 2.9, 3.4, 3.5, 3.6 and 3.7 | 256 |
| PECOMPACT | 2.xx | 1,058 |
| PETITE | 2.1 and 2.2 | 152 |
| UPACK | 0.2x-0.3x | 834 |
| UPX | UPX, UPX(LZMA), UPX(Delphi), 2.90, 2.92(LZMA), 2.93 and 3.00 | 9,931 |
| | | 17,336 |

**Table 5.3:** Data set two: malware sample set.

#### 5.4.1.2.3 Mixed Sample Data Set

The malware sample data set consists of six popular packers. Though each packer contains cross version samples, its complexity is relatively low. Therefore, the mixed sample data set is used to assess the robustness of the system effectiveness drawn from the results of malware sample set experiments. This data set contains not only popular but also sophisticated packers.

One problem is that our sample collection does not have a sufficient number of reliable samples of the sophisticated packers. To address this problem, the selection conditions used for malware sample data set are relaxed. The new selection criteria are:

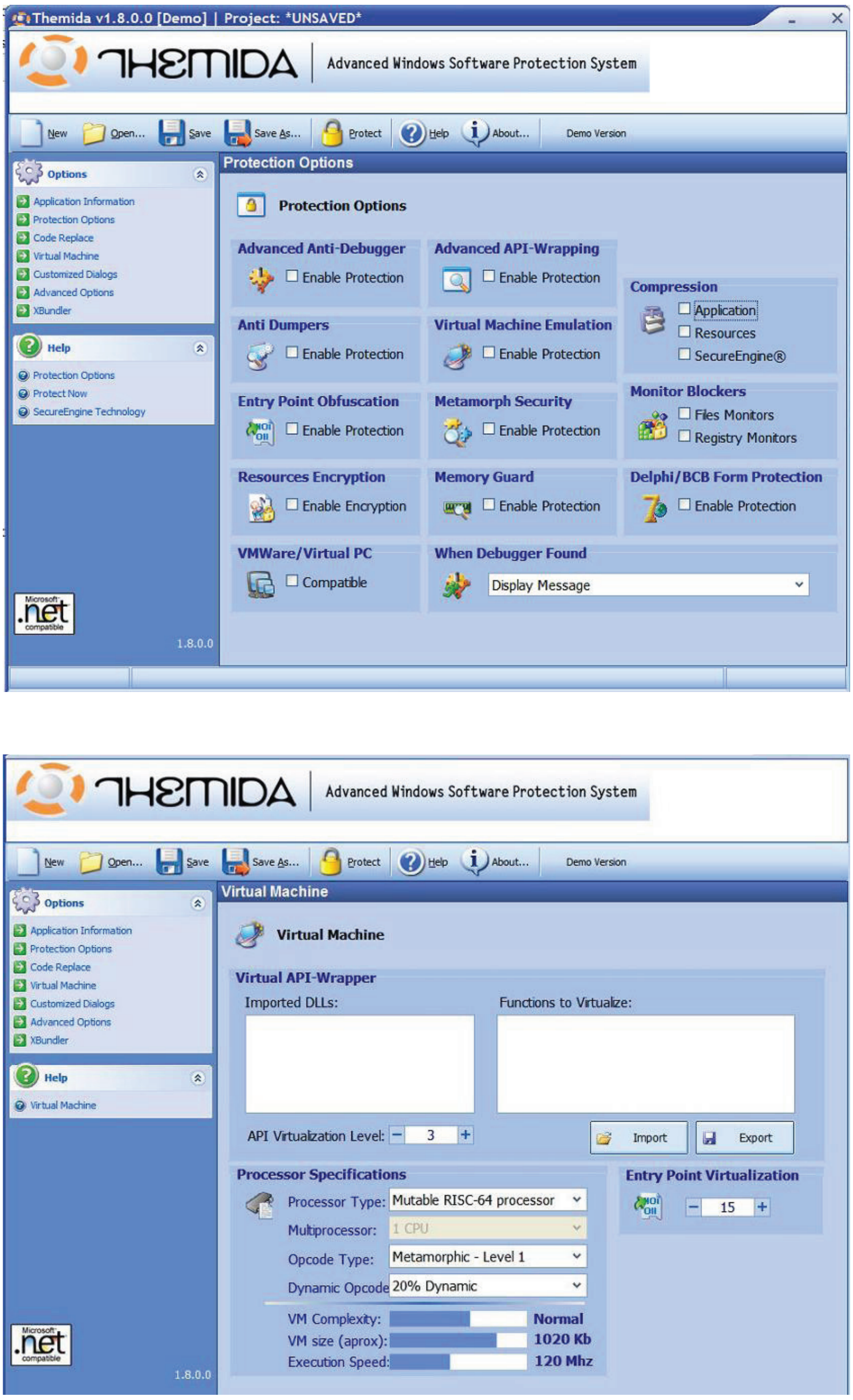
- The file's packer name is identified by two scanners, or is identified by one scanner and confirmed by PEiD.
- Packers with a sufficient number of classified cases (more than 100) are chosen.

466 files of two packers, Asprotect and Mew, that match the above criteria have been added into the data set. In addition, a popular and sophisticated packer, Themida, is chosen for this data set. As most samples of Themida in the database contain conflicting packer information, Themida packed clean files are used instead. 117 clean files in the UnxUtils binaries are packed with Themida v1.8.0.0 demo version. Figure 5.4 shows that Themida provides various protection options and the user is allowed to configure the protection level of the packer. Consequently, the number and the complexity of Themida variants are dramatically increased. In this thesis, six different combinations of packing options are evenly applied on these files.

The details of the set are presented in Table 5.4. Though this data set is not as reliable as the previous malware sample set, the experimental results on this data set will still provide an overall score of the system effectiveness.

### 5.4.2 Randomness Signature Scanning Results

The performance of two distance measures are compared in terms of effectiveness. Metrics used are the true positive (TP), the number of positive records correctly identified as specific class, and the false negatives (FN), the number of positive records which belong to the class but were incorrectly identified as other classes. All experiments are run in VMware 5.5.3 and the host operating system is Windows XP Professional 2002, service pack 2.



Protection Options

Advanced Anti-Debugger: ☐ Enable Protection
Advanced API-Wrapping: ☐ Enable Protection
Compression: ☐ Application ☐ Resources ☐ SecureEngine®
Anti Dumpers: ☐ Enable Protection
Virtual Machine Emulation: ☐ Enable Protection
Entry Point Obfuscation: ☐ Enable Protection
Metamorph Security: ☐ Enable Protection
Monitor Blockers: ☐ Files Monitors ☐ Registry Monitors
Resources Encryption: ☐ Enable Encryption
Memory Guard: ☐ Enable Protection
Delphi/BCB Form Protection: ☐ Enable Protection
VMWare/Virtual PC: ☐ Compatible
When Debugger Found: Display Message

Virtual Machine

Virtual API-Wrapper
Imported DLLs:
Functions to Virtualize:
API Virtualization Level: 3

Processor Specifications
Processor Type: Mutable RISC-64 processor
Multiprocessor: 1 CPU
Opcode Type: Metamorphic - Level 1
Dynamic Opcode 20% Dynamic
VM Complexity: Normal
VM size (aprox): 1020 Kb
Execution Speed: 120 Mhz

Entry Point Virtualization: 15

**Figure 5.4:** Themida protection options

| Packer | Versions | Total Number |
|---|---|---|
| ASPROTECT | unknown, 1.2 and 1.23 | 205 |
| FSG | 1.33 and 2.0 | 5,105 |
| MEW | 11 and 11 SE 1.2 | 261 |
| NSPACK | 2.x, 2.9, 3.4, 3.5, 3.6 and 3.7 | 256 |
| PECOMPACT | 2.xx | 1,058 |
| PETITE | 2.1 and 2.2 | 152 |
| THEMIDA | v1.8.0.0 with 6 option sets | 117 |
| UPACK | 0.2x-0.3x | 834 |
| UPX | UPX, UPX(LZMA), UPX(Delphi), 2.90, 2.92(LZMA), 2.93 and 3.00 | 9,931 |
| | | 17,919 |

**Table 5.4:** Data set three: mixed sample set.

Experiments have been carried out to compare the performance of packer classification systems with various implementations. As we discussed in Chapter 4, parameters such as the sample count $s$, the window size $w$, the skip size $a$ and the pruning size $n$ are initially determined empirically. However, whether the extracted feature set could gain the best performance with these settings is still questionable. Thus, various $s$, $w$, $a$ and $n$ values are firstly tested to determine a suitable feature vector for successful classification.

#### 5.4.2.1 Performance of the Fixed Sample Count Algorithm

Two sets experiment was carried out to evaluate two distance measures for the fixed sample count algorithm. The data set used was the packed clean data set. In each set of experiments, the system with various sample count $s$ values are experimented using corresponding distance measure.

If the window count $s$ is too large, the small file will have very small windows. If it is too small, the big file will have very big windows. Consider the size of testing data which are mainly in the range $3 - 191$ KB, $s$ is set in the range of $128 - 2048$. In that case, a 3 KB file will have window sizes ranged from around $2 - 24$ bytes and a 191 KB file will have window sizes ranged from around $96 - 1528$ bytes.

Table 5.5 lists the results of all runs. It is disappointing that for the fixed sample count algorithm, no matter which $s$ is chosen, both the Euclidean and Cosine distance measure's performance are unsatisfactory. This is mainly due to the fact that files of different sizes are compared together, which means a feature dilated in a small window is compared with the same feature contracted in a big window.

| Sample count | Distance measure | Total files | TP | FN | TP rate |
|---|---|---|---|---|---|
| 128 | Euclidean | 708 | 372 | 336 | 52.54% |
| | Cosine | 708 | 391 | 317 | 55.23% |
| 256 | Euclidean | 708 | 398 | 310 | 56.21% |
| | Cosine | 708 | 396 | 312 | 55.93% |
| 512 | Euclidean | 708 | 405 | 303 | 57.20% |
| | Cosine | 708 | 401 | 307 | 56.64% |
| 1024 | Euclidean | 708 | 445 | 263 | 62.85% |
| | Cosine | 708 | 419 | 289 | 59.18% |
| 2048 | Euclidean | 708 | 418 | 290 | 59.04% |
| | Cosine | 708 | 444 | 264 | 62.71% |

**Table 5.5:** Performance of the fixed sample count algorithm with various sample count $s$.

#### 5.4.2.2 Pruning Method Selection

Results of the sliding window algorithm need to be pruned into vectors of same size for characteristic representation. In this thesis, the effects of different pruning strategies and distance measures are compared. The data set used is the packed clean data set. For the algorithm, the window size is set to 32 and the skip size is 16. For all four pruning methods described above, the system uses 100 low randomness values of the file, that is, $n$ is set to 100. These values are chosen as a compromise between the information extracted from small file and big file.

Table 5.6 illustrates the results of experiments. The Cosine distance measure outperforms the Euclidean distance measure in all classification experiments as a distance measure. Among four pruning heuristics, the Trunk pruning is the most effective one followed by the Ordered smallest pruning. This is probably due to the fact that most packers store code at the beginning or the end of the file. This pattern is also displayed in the randomness scanning plots shown in the last chapter. Please note that these results are excellent, especially when Trunk pruning is used.

| **Pruning method** | **Distance measure** | **Total files** | **TP** | **FN** | **TP rate** |
|---|---|---|---|---|---|
| First | Euclidean | 703 | 568 | 135 | 80.79% |
| | Cosine | 703 | 577 | 126 | 82.07% |
| Smallest | Euclidean | 702 | 579 | 123 | 82.47% |
| | Cosine | 702 | 646 | 56 | 92.02% |
| Ordered smallest | Euclidean | 702 | 629 | 73 | 89.60% |
| | Cosine | 702 | 683 | 19 | 97.29% |
| Trunk | Euclidean | 703 | 685 | 18 | 97.43% |
| | Cosine | 703 | 690 | 13 | 98.15% |

**Table 5.6:** Evaluation of two distance measures and four pruning strategies using the sliding window algorithm

#### 5.4.2.3 Determination of Window Size $w$ and Skip Size $a$

Two sets of experiments have been initially carried out to determine the window size $w$ and the skip size $a$ for the sliding window algorithm. As shown in the last section, the Cosine measure combined with Trunk pruning gives the best performance. We therefore use this combination in the subsequent experiments to find the best parameter settings. Five files are removed from the data set as they produce more than $n$ same smallest randomness values, which are all rescaled to 0. As the result, there are total 703 files in each experiment.

In each set of experiments, the total number of original bytes used for the feature vector remains roughly the same. In other words, similar features of the file are used for packer classification. For example, when test the window size $w$, the skip size is set to $w/2$ and the pruning size $n$ will vary as $w$ changes. Consider a run with $w = 32, a = 16$ and $n = 100$, the number of bytes used is around $100 \times 16 + 32 = 1632$. In another experiment, if $w$ is set to 16, then $a = 8$ and $n$ will be set to 200 ($200 \times 8 + 16 = 1616$).

The results are illustrated in Table 5.7 and 5.8. For $w$, the true positive rate of window size of 8 and 16 are slightly higher than 32. However, windows of small size carry less flexible information than big size does. Besides, feature extraction efficiency is also a factor when selecting the window size. The smaller the window size, the more randomness outputs are generated and the more time-consuming it is. Therefore, 32 was chosen as a suitable window size in the algorithm for packer classification purposes.

| Window size | Skip size | Pruning size | Total files | TP | FN | TP rate |
|---|---|---|---|---|---|---|
| 8 | 4 | 400 | 703 | 691 | 12 | 98.29% |
| 16 | 8 | 200 | 703 | 691 | 12 | 98.29% |
| 32 | 16 | 100 | 703 | 690 | 13 | 98.15% |
| 64 | 32 | 50 | 703 | 684 | 19 | 97.29% |

**Table 5.7:** Determination of the window size $w$, where the skip size $a = w/2$, the distance measure is *cosine* measure and the pruning method is *Trunk*. The pruning size varies with different window size so that that the extracted features used for comparison are roughly same.

| Skip size | Pruning size | Total files | TP | FN | TP rate |
|---|---|---|---|---|---|
| 2 | 800 | 703 | 689 | 14 | 98.00% |
| 4 | 400 | 703 | 690 | 13 | 98.15% |
| 8 | 200 | 703 | 690 | 13 | 98.15% |
| 12 | 134 | 703 | 690 | 13 | 98.15% |
| 16 | 100 | 703 | 690 | 13 | 98.15% |
| 20 | 80 | 703 | 689 | 14 | 98.00% |
| 32 | 50 | 703 | 691 | 12 | 98.29% |

**Table 5.8:** Determination of the skip size $a$, where the window size $w = 32$, the distance measure is *cosine* measure and the pruning method is *Trunk*. The pruning size varies with different skip size so that the extracted features used for comparison are roughly same.

If the experimental results are examined with respect to the skip size $a$, it is noted that when using a similar amount of information for comparison, the effectiveness of different systems are very close. In Table 5.8, if windows do not overlap, that is, $a = w = 32$, the system achieves slightly high TP rate than others. Moreover, the system with this implementation is the most efficient one. In this case, the total number of outputs of the randomness test and the extracted feature vector size are the smallest, so both of the feature extraction process, including the scaling and pruning, and the classification process are fast.

#### 5.4.2.4 Determination of Pruning Size $n$

So far, the most effective classification performance (TP = 98.29%) has been obtained using the settings below:

- Sliding window algorithm,
- Cosine distance measure,
- Trunk pruning,
- window size $w = 32$,
- non-overlapping windows, that is $a = w = 32$,
- pruning size $n = 50$.

This set of experiments is used to determine how detailed the information used for classification should be. This depends on the feature vector size (pruning size $n$). If $n$ is small, the classification takes less time to build the model and classify the file. However if $n$ is too small, there may not be sufficient information to distinguish between packers. If $n$ is too large, the classification process is slow and also the noise generated by compressed/encrypted data will affect system performance. Therefore, experiments are run with various pruning size $n$ in the range of $30 - 70$ and other settings given in Table 5.9. Thus, the information used are around $1 - 2$ KB. Table 5.9 suggests that $n$ should be set between $30 - 50$.

| Pruning size | Total files | TP | FN | TP rate |
|---|---|---|---|---|
| 30 | 703 | 693 | 10 | 98.57% |
| 40 | 703 | 693 | 10 | 98.57% |
| 50 | 703 | 691 | 12 | 98.29% |
| 60 | 703 | 674 | 29 | 95.87% |
| 70 | 703 | 675 | 28 | 96.01% |

**Table 5.9:** Determination of the pruning size $n$, where the window size $w = 32$, the skip size $a = 32$, the distance measure is *cosine* measure and the pruning method is *Trunk*.

#### 5.4.2.5 Malware Samples Experiments

We have seen that packer classification experiments on packed clean files have achieved good performance with TP rate over 98%. However, the packed clean files are a small data set. It only contains files ranging between 3 and 191 KB. Moreover, each test packer contains only one version of the specific packer. Therefore, we evaluated this randomness signature approach again using a large malware data set described in Table 5.3. The best combinations of settings applied in the last section have been used here as well, namely:

- Sliding window algorithm,
- Cosine distance measure,
- Trunk pruning,
- window size $w = 32$,
- non-overlapping windows, that is $a = w = 32$,
- pruning size $n = 30 - 50$.

As shown in Table 5.10 and Table 5.11, the performance of the system is still unsatisfactory. The average true positive rate in three tests ($n = 30$, 40 and 50) are all below 90%, while in the $n = 50$ test, NSPACK and PETITE only achieve 40.63% and 65.13% respectively.

The low accuracy in the results obtained by the randomness signature approach motivated us to investigate the pattern recognition techniques described in Section 3.2 for packer classification using the extracted randomness profile. In the remainder of this chapter, we evaluate four fast statistical classifiers, namely Naive Bayes, Sequential Minimal Optimization, $k-$Nearest Neighbor and Best-first Decision Tree.

| **Pruning size** | **Total files** | **TP** | **FN** | **TP rate** |
|---|---|---|---|---|
| 30 | 17336 | 14853 | 2483 | 85.68% |
| 40 | 17336 | 14782 | 2554 | 85.27% |
| 50 | 17336 | 14765 | 2571 | 85.17% |

**Table 5.10:** Performance of the sliding window algorithm using malware sample data set, where the window size $w = 32$ and the skip size is 32, the distance measure is *Cosine* measure and the pruning method is *Trunk*. The pruning size are in the range $30 - 50$.

| **Packer** | **Total files** | **TP** | **FN** | **TP rate** |
|---|---|---|---|---|
| FSG | 5105 | 5081 | 24 | 99.53% |
| NSPACK | 256 | 104 | 152 | 40.63% |
| PECOMPACT | 1058 | 802 | 256 | 75.80% |
| PETITE | 152 | 99 | 53 | 65.13% |
| UPACK | 834 | 823 | 11 | 98.68% |
| UPX | 9931 | 7856 | 2075 | 79.12% |

**Table 5.11:** Detailed performance of the sliding window algorithm using malware sample data set, where the window size $w = 32$ and the skip size is 32, the distance measure is *Cosine* measure and the pruning method is *Trunk*. The pruning size is 50.

## 5.4.3 Pattern Recognition Results

Experiments have been carried on two types of data set, the malware sample data set and the mixed sample set. Both of them contain a large number of real malware samples.

In the feature extraction process, each packed file is passed to the sliding window randomness test with a window size 32 bytes (256 bits). There is no overlap between windows, i.e., the skip size $a$ is 32 too. Then the feature vector is constructed by extracting $30 - 50$ low randomness values from the output using the Trunk pruning method.

The classification process was carried out on the machine learning package Weka (see Section 3.2.4). All selected statistical classifiers, NB, SMO, kNN (called IBk in Weka) and BFTree are implemented in Weka. In the experiments, all classifiers used the default settings defined by Weka.

In the tests, 10-fold cross validation [60] is used. For each data set, the whole set is randomly partitioned into ten equal-size subsets. There are a total of 10 runs. During each run, one subset is used for testing and the other nine subsets are used for training. Therefore, each vector is used as a test sample exactly once.

#### 5.4.3.1 Results of the Malware Sample Data Set

Experiments were firstly carried on the malware sample data set with feature vector size (pruning size) 50. The performance, in terms of effectiveness and efficiency, of various statistical classifiers are listed in Table 5.12. All these classifiers work very well with high true positive rate ($> 93\%$) and low false positive rate. This provides very strong evidence that the randomness profile plays a significant role in a packer classification system.

Among all classifiers, the kNN classifier achieves the best overall performance. Its TP rate is 99.6% and FP rate is only 0.1%. Moreover, it takes least time to build a model on training data (*Model building time* in Table 5.12).

| **Classifier** | **TP rate** | **FP rate** | **Model building time (s)** |
|---|---|---|---|
| Bayes.NaiveBayes | 93.9% | 1.1% | 0.91 |
| Functions.SMO | 98.9% | 0.8% | 72.11 |
| Lazy.kNN(k=1) | 99.6% | 0.1% | 0.02 |
| Trees.BFTree | 99.3% | 0.5% | 16.45 |

**Table 5.12:** Comparison of statistical classifiers. The feature vector contains 50 points

Two other sets of experiments are used to determine the $k$ values used in the kNN classifier and the size of the extracted feature vector. Table 5.13 shows that $k = 1$

outperforms other two $k$ values, 3 and 5. Table 5.14 shows feature vectors of 30-50 points all achieve high effectiveness (TP > 99%) while feature vector of 50 points yields the best result.

| k | TP rate | FP rate | Model building time (s) |
|---|---|---|---|
| 1 | 99.6% | 0.1% | 0.02 |
| 3 | 99.5% | 0.2% | 0.02 |
| 5 | 99.4% | 0.3% | 0.02 |

**Table 5.13:** Comparison of kNN with different $k$ values. The feature vector contains 50 points

| Vector size | TP rate | FP rate | Precision | Recall |
|---|---|---|---|---|
| 30 | 99.4% | 0.3% | 99.4% | 99.4% |
| 40 | 99.6% | 0.1% | 99.6% | 99.6% |
| 50 | 99.6% | 0.1% | 99.8% | 99.7% |

**Table 5.14:** Comparison of kNN (k = 1) with different feature vector size.

#### 5.4.3.2 Results of the Mixed Sample Data Set

The results of malware sample data set shows that our packer classification system can achieve extremely effective performance using the pattern recognition techniques. With the kNN classifier, the TP rate achieved is 99.6% while its FP rate is only 0.1%. Through the experiments, it is proved that this novel packer classification technique works well with different packer variants. However, it is unknown whether the conclusions drawn in previous sections can be applied to packers with different sophistication. To address this, an experiment was run on the mixed sample data set. This data set contains not only packed clean files and malware samples, but also lowly complex packers and highly complex packers.

50 randomness values were extracted from the file to construct the feature vector. In the classification process, the kNN (k=1) classifier is used. The results in Table 5.15

prove that sophisticated packers, such as Asprotect and Themida, can also be effectively classified by applying the pattern recognition techniques on packer's randomness profile. As shown, the average TP rate is 99.4%. Among 9 packers, the TP rates of Upack and FSG obtain nearly perfect while all other packers achieve more than 90%.

| **Packer** | **TP rate** | **FP rate** | **Precision** | **Recall** |
|---|---|---|---|---|
| ASPROTECT | 92.7% | 0.1% | 95.5% | 92.7% |
| FSG | 99.9% | 0.0% | 99.9% | 99.9% |
| MEW | 99.6% | 0.0% | 100.% | 99.6% |
| NSPACK | 91.0% | 0.1% | 90.7% | 91.0% |
| PECOMPACT | 98.5% | 0.1% | 98.2% | 98.5% |
| PETITE | 98.0% | 0.0% | 98.7% | 98.0% |
| THEMDIA | 92.3% | 0.1% | 86.4% | 92.3% |
| UPACK | 100.% | 0.0% | 99.4% | 100.% |
| UPX | 99.7% | 0.3% | 99.8% | 99.7% |
| *Weighted Avg* | 99.4% | 0.2% | 99.5% | 99.4% |

**Table 5.15:** Detailed accuracy by class using the mixed sample data set. The classifier is kNN with $k = 1$ and the feature vector contains 50 points

## 5.5 Summary and Discussion

This chapter has presented a *fast yet effective* packer classification technique which uses features extracted from the low randomness profile of the packer. Two approaches have been developed to map the test file to the candidate packer. These two approaches are randomness signature scanning and pattern recognition approach. While the first approach generates a randomness signature to represent each packer, the second approach builds a classification model for each packer.

Our work demonstrates that the pattern recognition approach performs better than randomness signature scanning approach does. The randomness profile combined with strong pattern recognition algorithms can be used to produce a highly accurate packer classification system on real life data. Such a system identifies the packer automatically and therefore is essential to keeping up with the accelerating growth in packer varieties.

The system has been tested on more than $17,000$ malware samples from the wild. These malware samples are packed by various packers including not only different versions of packers, but also packers of different complexity. We evaluated four popular fast statistical classifiers, namely Naive Bayes, Sequential Minimal Optimization, $k-$Nearest Neighbor and Best-first Tree. All four classifiers were extremely effective, and three of the four algorithms achieved an average true positive rate of around 99% or above, Naive Bayes was the lowest, with a true positive rate of around 94%. *The $k-$Nearest Neighbor classifier with $k = 1$ obtains the best overall performance. Its true positive rate is* 99.6% *while false positive rate is* 0.1%. *Moreover, it is the fastest classifier.* The system also reveals that the low randomness profile of the packed file, normally produced by the PE header and unpacking stub, contains significant packer's information. Thus it is very useful in distinguishing between families of packers.

# Chapter 6

# A Statistical Approach to Generic Unpacking

This chapter focuses on another important task in packer analysis, namely, unpacking. A packed file must be unpacked completely before being sent to a malware scanner for malicious code detection. Here, we introduce a novel statistical solution, which is named *Hump-and-Dump*, for generic unpacking, and analyse its performance.

This unpacking technique relies upon creating a histogram of the addresses of executed instructions (EIP on x86). Whilst other researchers [16, 64] have done this, the trick is to order the histogram by the last time an address is executed. The algorithm is based upon the dual observations that

**(a)** even in a packed program, the OEP bytes are almost always only executed once, and

**(b)** most packers unpack the original program to an area of memory which has not been previously executed.

The Hump-and-Dump technique effectively locates the original entry point (OEP) area of a packed program. In an ordered address execution histogram, decryption, decompression and copying appear as large spikes (*hump*) at the start of the histogram, followed by a flat section, of height one, which is usually the region corresponding to the OEP instructions. This unique shape of the chart suggests a good opportunity to stop and *dump* the memory.

Furthermore, incorporating the Hump-and-Dump technique with the section information can precisely locate the OEP. A dumped memory at the OEP can usually be restored to its original status before the file is packed and run without penalty.

This technique is extremely efficient to implement. Two implementations, namely off-line and on-line algorithms, are developed. The off-line algorithm locates the OEP from an execution trace after the program has finished running, and the on-line algorithm computes the OEP of a live running program, i.e., "on-the-fly" in an emulator.

## 6.1 Introduction

With the employment of compression and encryption by malware writers, the packed file is made unrecognizable to an outside observer, be it a human or another program such as an anti-virus scanner. In order for an anti-virus scanner to determine whether a packed program is a piece of malware or legitimate software, it must usually unpack it to see what is underneath. Unpacking is a vital step in achieving a successful malware analysis and detection.

For a packed file, unpacking is the process that unwraps the layers of packers and restores the original executable. The unpacked file then normally executes from its original entry point, i.e., the OEP, without penalty. In other words, during the whole execution, the OEP is the boundary between the execution of the packed file and the unpacked file. At this point, the unpacking has finished and the unpacked memory can be dumped for further analysis.

Locating the OEP is not a simple task as it sounds. If the packer is identified and its corresponding OEP finding module is available, the OEP can be easily located either using manual unpacking or static unpacking method (Secion 2.5.1 and Section 2.5.2). As most new malware is now designed for illicit financial gain [37, 67] or political missions [33], more attention is paid to providing a hard protective shell for malware. An increasing number of anti-unpacker tricks and obfuscation methods are now being applied to malware packers. The targets include the common packers in existence as well as newly emerging packers. This means that the number and the sophistication of new packer strains and variants have been dramatically increased. Consequently, unpacking has become a real challenge for anti-virus researchers and AV software. *A generic unpacking technique, that obtains fully decrypted memory for every type of packer without knowing which PE packer is used and what encryption/compression algorithm it utilizes, is essential to keeping up with the accelerating growth in packed malware.*

Many attempts have been carried out with the aim to deliver a generic unpacking solution [9, 40, 45, 53, 70, 92, 97, 106, 109, 110]. As described in Section 2.5.3, the run-and-dump approach is the simplest unpacking method, but it does not stop the running program at the right point (which would be after unpacking) and dump the memory. The run-and-dump action can also easily miss processes that terminate quickly, i.e., the process terminates itself before it is stopped and dumped manually. The main problem with the existing automatic unpacking tools, such as PolyUnpack [97], Renovo [53], OmiUnpack [70], etc, is that they significantly increase computational overhead. Besides, most of these unpacking tools are mainly based on tracking the packer's written-then-executed memory and are not able to locate the OEP in all cases.

Computing emulation (see Section 2.5.3.2) is a very powerful unpacking tool which has attracted attention from a lot of researchers [9, 40, 106, 109]. However, similar to the automatic unpacking tools, the disadvantage associated with emulation is the high computational overhead. More importantly, it is hard to determine when to stop the emulator.

We present a new and efficient generic unpacking algorithm which effectively locates the OEP of a packed program. Although the dumped unpacked binary may no longer execute on its own, many of its features will be exposed, thereby greatly simplifying analysis.

## 6.2 Ordered Address Execution Histogram

An ordered address execution histogram, as shown in Figure 6.2, is a graphical display of tabulated frequencies of the addresses of executed instructions ordered by the last time an address is executed.

The proposal of using an ordered address execution histogram for unpacking is based on the heuristic rule that (a) even in a packed program, the OEP bytes are almost always only executed once, and (b) most packers unpack the original program to an area of memory which has not been previously executed, the addresses that belong to the OEP instructions will only be executed once. Meanwhile, the main decompression and/or decryption routines during the unpacking will have a high frequency.

In a packed file execution, the unpacking activity happens before the original file execution. Therefore, when the histogram is ordered by the last time an instruction address is executed, the instructions of main loops, such as decryption, decompression and copying, must be listed before the OEP. Moreover, these main loops usually appear as large spikes at the start of the histogram, followed by a flat section, of height one, which is usually the OEP. This pattern of the histogram clearly indicates the boundary between unpacking activity and the original executable file in memory. From the histogram, the OEP area is easy to identify. This is a good time to take a memory dump of the process for further analysis.

## 6.3 Implementation

The technique is extremely efficient to implement, and can compute the OEP "on the fly" in an emulator, or off-line from a trace of EIP.

### 6.3.1 Off-line Hump-and-Dump Implementation

Four steps are involved in the off-line computation (Algorithm 6.1). Firstly, an execution trace is run to collect all executed EIPs. The detail of the tracer will be discussed in Section 6.3.1.1.

```
   Data  : Packed application
   Result: Unpacked application
   begin
1.     Run the program in the trace mode and collect all executed EIPs ;
2.     Create the ordered address execution histogram: ;
       Initialize a hash table $addrHist$ of which the key is the $eip$ ;
       Set the counter $counter \longleftarrow 0$ ;
       foreach $eip$ do
           $counter+ = 1$ ;
           if $addrHist.has\_key(eip)$ then
               $addrHist[eip] = (addrHist[eip][frequency] + 1, counter)$ ;
           endif
           else
               $addrHist[eip] = (1, counter)$ ;
           endif
       end
       Sort $addrHist$ by the $counter$ Draw the histogram using $addrHist$;
3.     Identify the pattern that large spikes followed by the flat section of height one
       and locate the OEP;
4.     Run the program to the OEP and dump the memory ;
   end
```

**Algorithm 6.1:** Implementation of off-line Hump-and-Dump algorithm

Once addresses are collected, a Python script is then used to perform basic analysis of the address trace to produce a histogram showing address frequency, in the order of when the address was last executed. In the script, a hash table, of which the key is the address, is built. For each entry of the hash table (an address structure), only three DWORDs are needed to store the required information: address, execution frequency and incrementing counter (execution time). For each EIP in the trace result, if the address is already in the hash table, the corresponding entry's frequency is increased by one and the current counter is recorded as the point at which it was last executed. If it is a new address which hasn't been executed before, a new entry is added to the hash table. The frequency of this address is set to one and its execution time is set to the current counter. After all EIPs have been processed, the whole hash table is sorted by each entry's incrementing counter. The graph is produced using the hash table data. The $x$-axis is the addresses in the order of the time it was last executed and the $y$-axis is the frequency.

The next step is to search the histogram for a pattern of a large spike followed by a flat section of height one. Once the pattern is found, the boundary between the large spikes and the flat section is regarded as the OEP area. At that point, the unpacking has finished and the original file will be represented in the memory.

Lastly, the program is run again until it reaches the OEP and then the program is dumped.

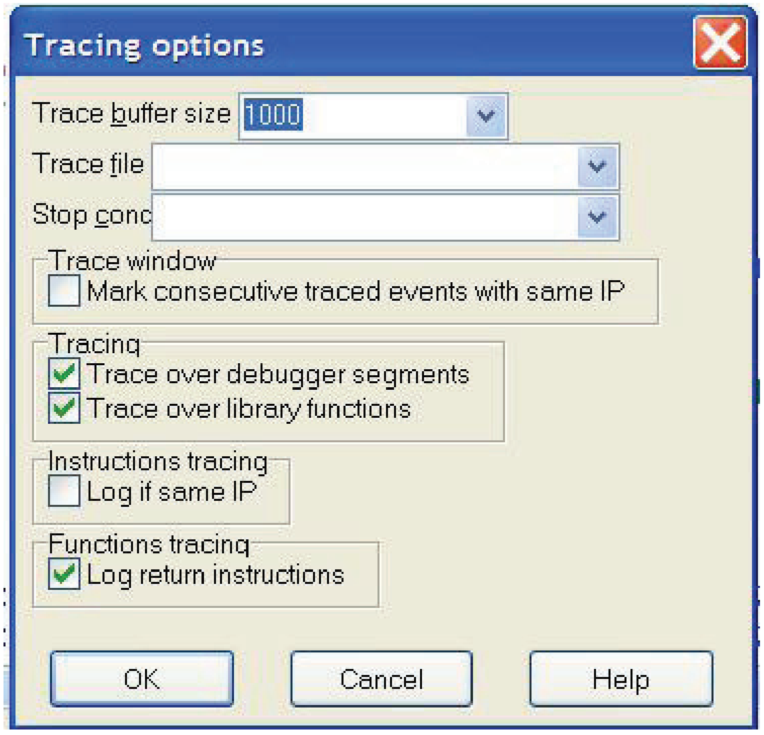
#### 6.3.1.1 Tracer

The main trace tool, proposed here is a C++ plugin of the commercial Interactive Disassembler Pro (IDA). Before running the plugin, the tracing option of IDA needs to be configured as in Figure 6.1, so that the tracer runs over the debugger segment and the library functions. That means, only addresses of user code will be recorded.

Other open source tools, PIN[86] and PyDbg[88], can also be modified to serve the same purpose as the IDA tracer plugin.

PIN is an instrumentation tool for Linux or Windows executables for a wide range of Intel processors, including the XScale, ia-32, ia-32e (64 bit x86), and Itanium processors. It dynamically adds the code while the executable is running. PIN is able to handle multi-threaded programs and self-modifying code. Moreover, it provides a module, itrace, for address traces which fits our needs.

PyDbg is a debugger engine developed in Python. Its abstracted interface allows for customized debugging scripts. In this thesis, a Python script is written to run PyDbg to have an execution trace of the address.


Tracing options

Trace buffer size 1000

Trace file

Stop conc

Trace window
- [ ] Mark consecutive traced events with same IP

Tracing
- [x] Trace over debugger segments
- [x] Trace over library functions

Instructions tracing
- [ ] Log if same IP

Functions tracing
- [x] Log return instructions

**Figure 6.1:** Tracing option of IDA

### 6.3.2 On-line Hump-and-Dump Implementation

Algorithm 6.2 is an implementation of the on-line version of “Hump-and-Dump” which computes the OEP “on-the-fly” in an emulator. In this thesis, an IDA plugin is implemented using this algorithm to test the concept.

The on-line algorithm is similar to the off-line algorithm except that the address histogram is built dynamically at each step of tracing during the program execution.

Two tuning parameters, $T_{\text{hump}}$ and $T_{\text{flat}}$, are used to determine the extent to which hump and the flat section affect the OEP finding, respectively. Both parameters’ values depend on the nature of the packer and possibly on the size of the original file. During the tracing, once both $T_{\text{hump}}$ and $T_{\text{flat}}$ are exceeded, the unpacking is assumed to be finished. The program will then be stopped and the memory of unpacked file will be dumped.

```
Data   : Packed application
Result: Unpacked application
begin
    Start emulating the application ;
    Initialize a hash table $addrHist$ of which the key is the $eip$ ;
    Set hump flag $hump \longleftarrow false$ ;
    Set flat section counter $flatCount \longleftarrow 0$ ;
    Set threshold of the hump $T_{hump}$ ;
    Set threshold of the flat section which is repeated number of new addresses of
    height 1 $T_{flat}$ ;
    foreach $eip$ do
        $counter += 1$ ;
        if $addrHist.has\_key(eip)$ then
            $flatCount = 0$ ;
            $addrHist[eip] = (addrHist[eip][frequency] + 1, counter)$ ;
            if $addrHist[eip][frequency] > T_{hump}$ then
                $hump = True$ ;
            endif
        endif
        else
            $addrHist[eip] = (1, counter)$ ;
            if $hump$ then
                $flatCount += 1$ ;
                if $flatCount > T_{flat}$ then
                    Stop the application ;
                    Dump the memory ;
                endif
            endif
        endif
    end
end
```

**Algorithm 6.2:** Implementation of on-line Hump-and-Dump algorithm

## 6.4 Experiments and Results

### 6.4.1 Off-line Experiments and Results

#### 6.4.1.1 Test Data Set

Off-line experiments have been carried out intensively with ten packers applied to two types of executable file, Win32 GUI (Graphic User Interface) application (calc.exe) and Win32 console application (date.exe). The top four packers in the WildList [113] have all been tested. They are UPX, Morphine, Mew and FSG. A detailed testing data set is listed in Figure 6.1. All experiments were run in VMware Workstation 5.5.3 with the host operating system as Windows XP Professional 2002, service pack 2.

Modern malware is usually wrapped in layers of compression and/or encryption. In order to examine whether the Hump-and-Dump pattern exists in a multi-packed file or not, a multi-packer (UPX 2.03w+ Morphine 2.7) has also be tested (see last row in Figure 6.1).

| Packer Name | Executable 1 (calc.exe) | Executable 2 (date.exe) |
|---|---|---|
| ASPack 2.12 | `calc_aspack.exe` | `date_aspack.exe` |
| FSG 2.0 | `calc_fsg.exe` | `date_fsg.exe` |
| Mew 11 | `calc_mew.exe` | `date_mew.exe` |
| Morphine 2.7 | `calc_morphine.exe` | `date_morphine.exe` |
| PC Shrinker 0.71 | `calc_pcshrinker.exe` | `date_pcshrinker.exe` |
| PECompact 2.5 | `calc_pecompact.exe` | `date_pecompact.exe` |
| PE Diminisher v0.1 | `calc_pediminisher.exe` | `date_pediminisher.exe` |
| RLPack 1.19 | `calc_rlpack.exe` | `date_rlpack.exe` |
| UPack 0.399 | `calc_upack.exe` | `date_upackexe` |
| UPX 2.03w | `calc_upx.exe` | `date_upx.exe` |
| UPX 2.03w + Morphine 2.7 | `calc_upx_morphine.exe` | `date_upx_morphine.exe` |

**Table 6.1:** Unpacking strategy testing data.

#### 6.4.1.2 Off-line Algorithm Results

The off-line algorithm is very fast and effective. Figures 6.2 - 6.11 illustrate the address histogram of the traced applications listed in Table 6.1 on both linear and log scales. In the graphs, the addresses are organised by their last execution time.

As can be seen in all of the graphs, the first part of the execution, i.e., the unpacking activity, involves much more recurrence compared with the second part, which is the original file execution. This pattern can be used to guess when unpacking has finished. For example, in Figure 6.2, the pattern of the massive hump followed by the flat section with one count is evident and the actual OEP of the file is just on the boundary area between the hump and the flat section. A memory dump taken at this time, will contain the original unpacked file and can be used for further analysis.

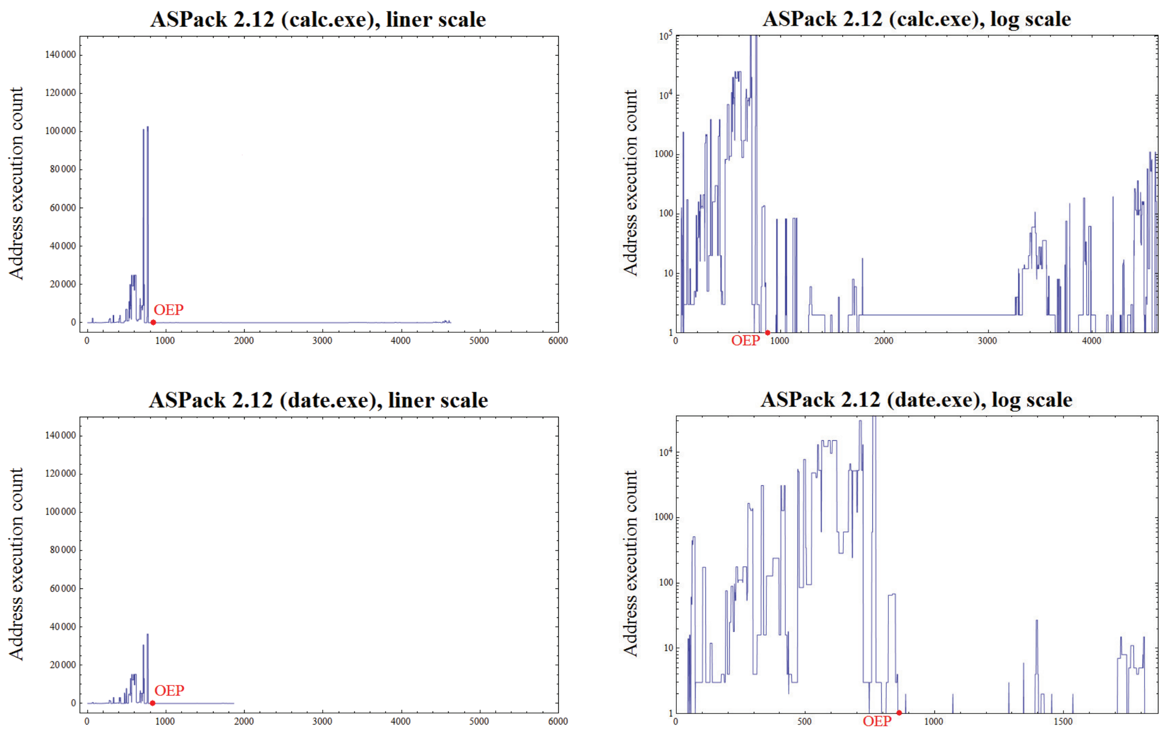


**Figure 6.2:** ASPack 2.12 unpacking trace address histogram

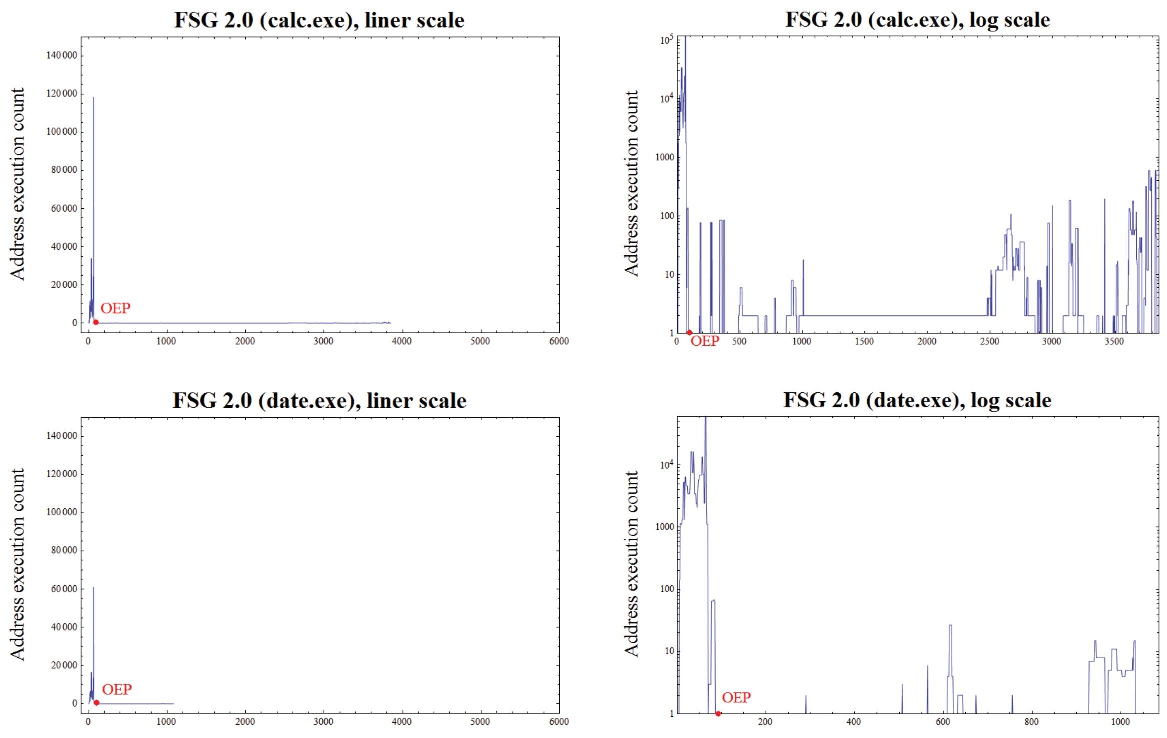


**Figure 6.3:** FSG 2.0 unpacking trace address histogram

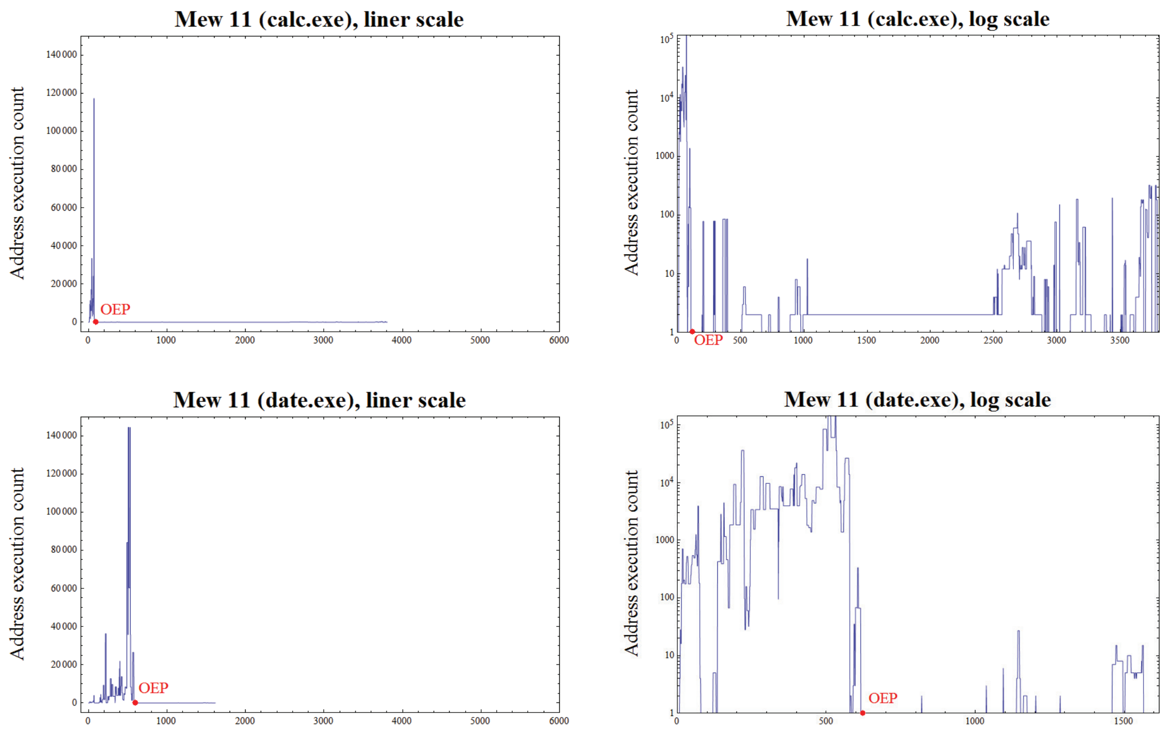


**Figure 6.4:** Mew 11 unpacking trace address histogram

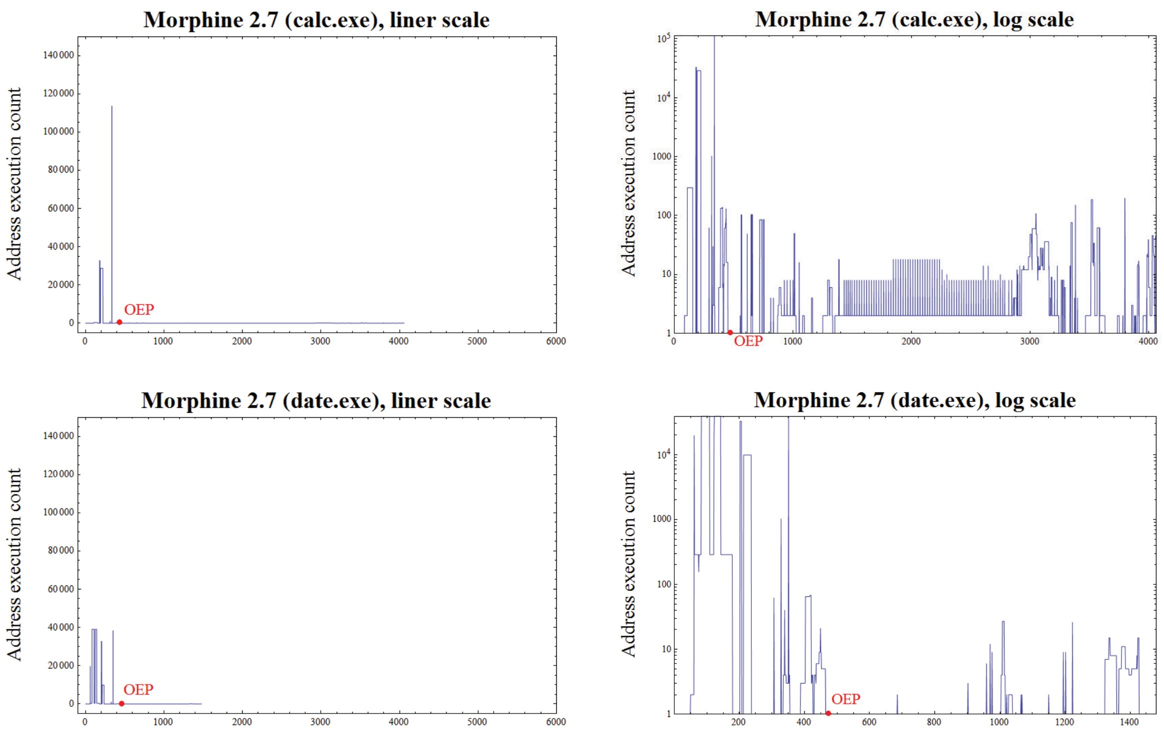


**Figure 6.5:** Morphine 2.7 unpacking trace address histogram

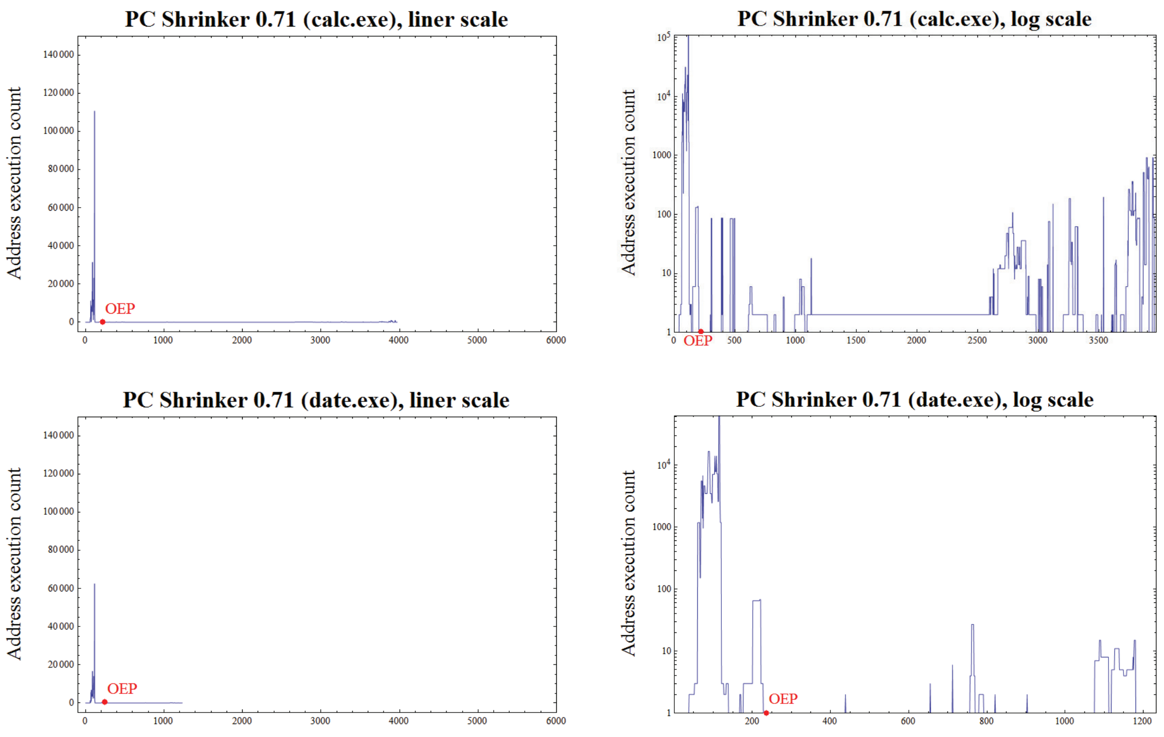


**Figure 6.6:** PC Shrinker 0.71 unpacking trace address histogram

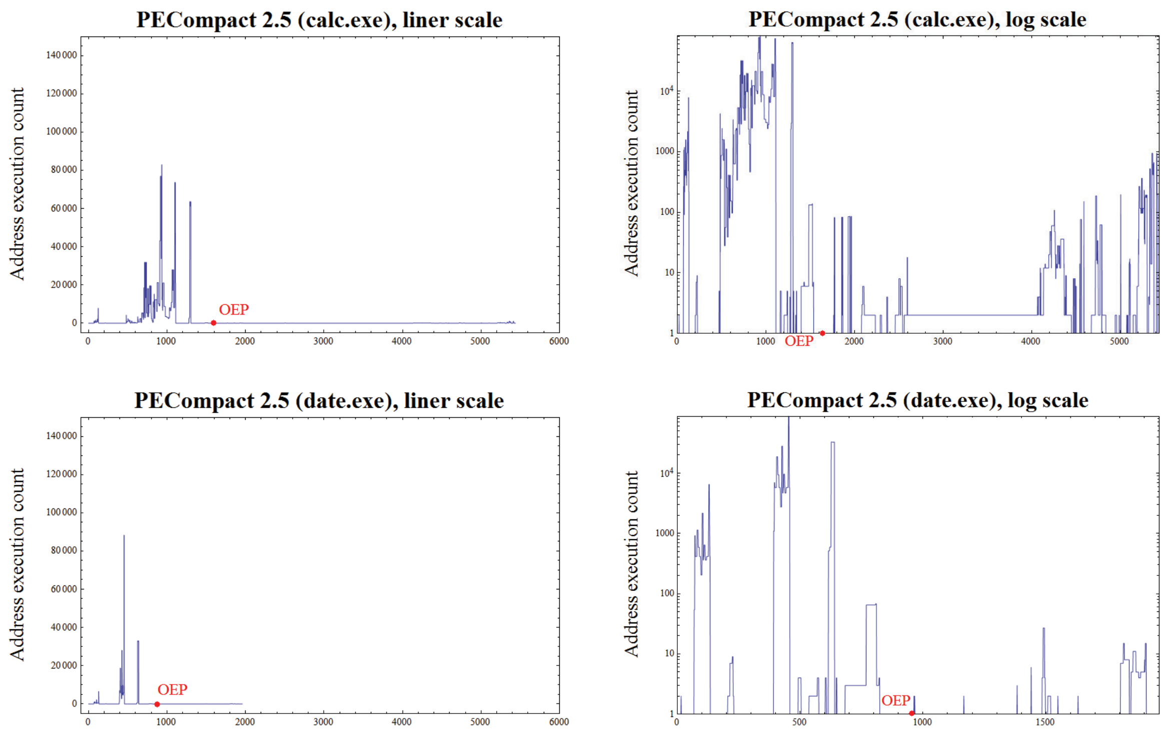


**Figure 6.7:** PECompact 2.5 unpacking trace address histogram

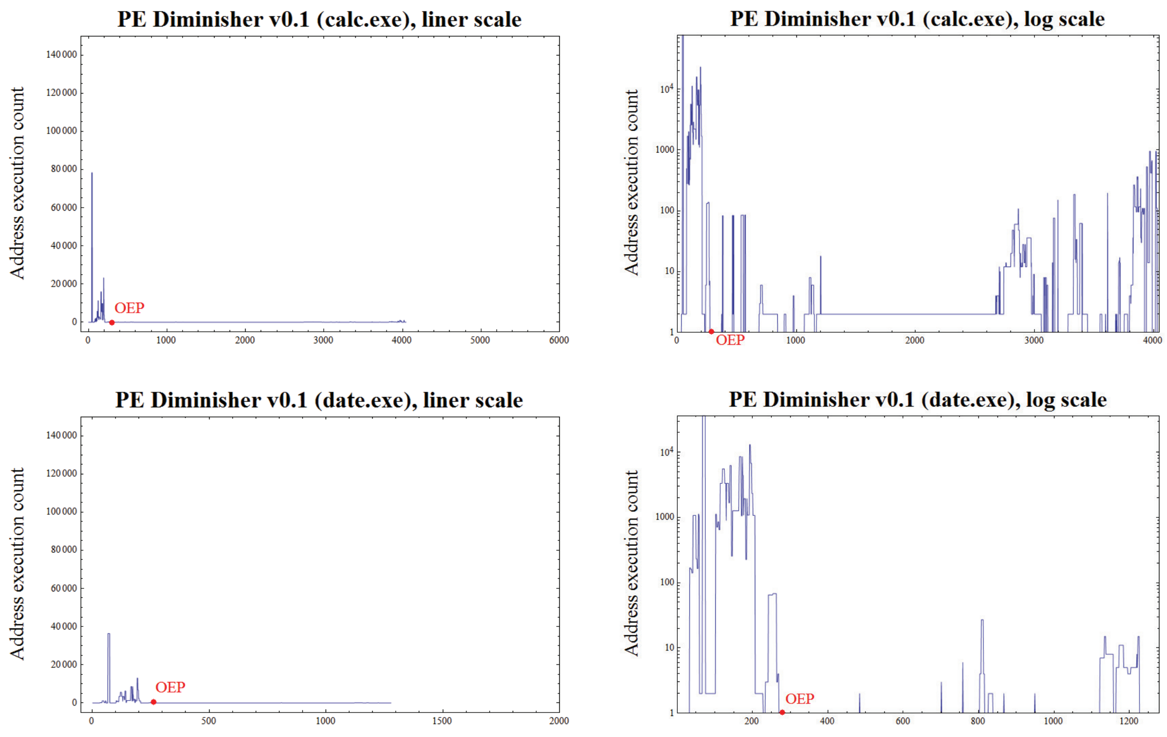


**Figure 6.8:** PE Diminisher v0.1 unpacking trace address histogram

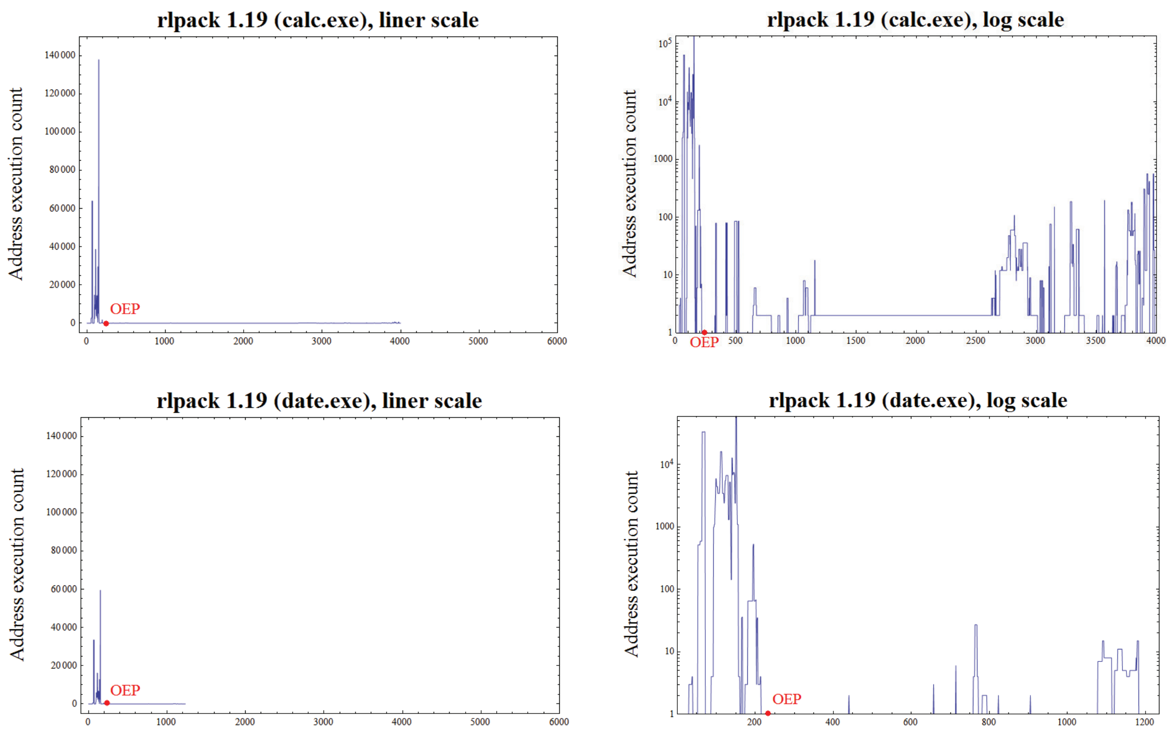


**Figure 6.9:** RLPack 1.19 unpacking trace address histogram

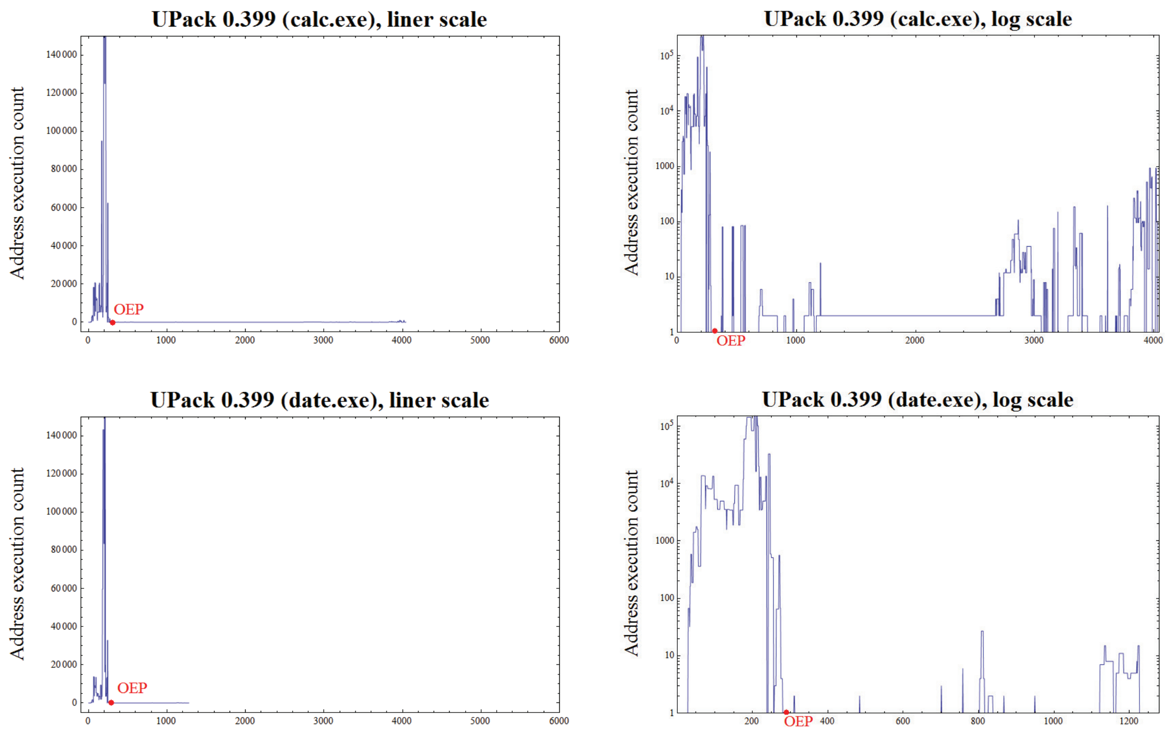


**Figure 6.10:** UPack 0.399 unpacking trace address histogram

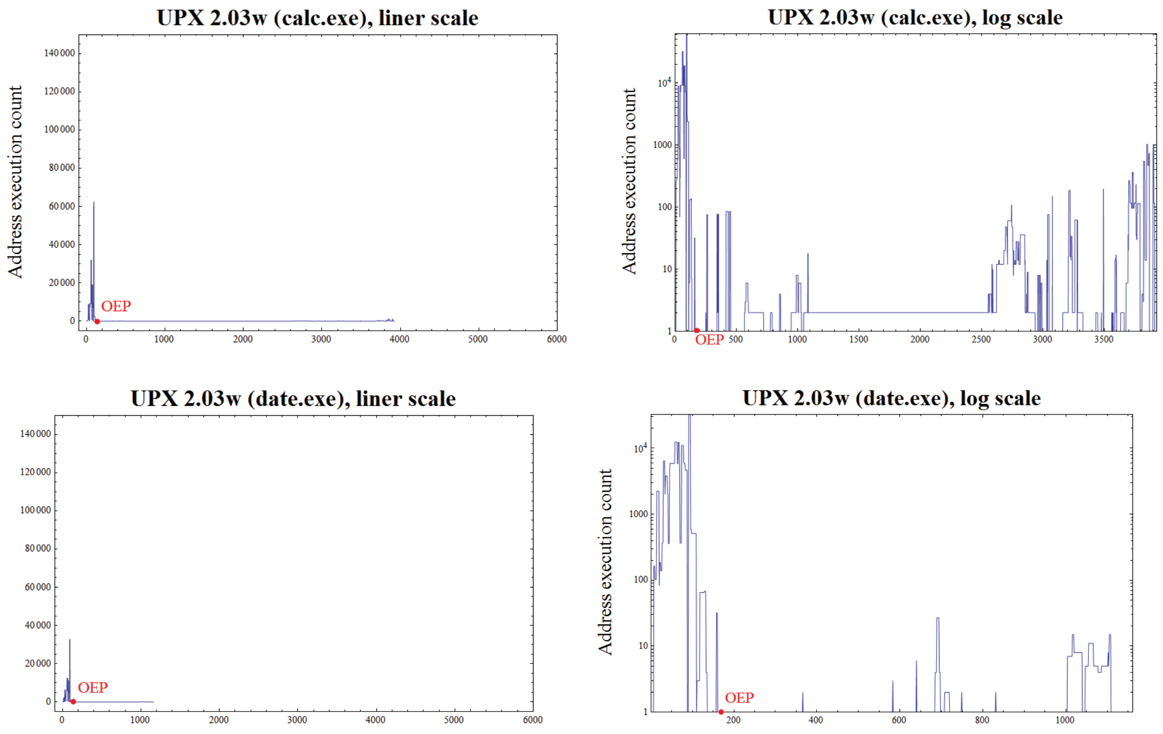


**Figure 6.11:** UPX 2.03w unpacking trace address histogram

### 6.4.1.3 Multi-packer Results

An example of the multi-packer (Morphine + UPX) result, shown in Figure 6.12, clearly illustrates the OEP flat section after the massive unpacking humps. However, compared to patterns from the single packed files, such as Figure 6.5 (Morphine) or Figure 6.11 (UPX), there are more humps. Moreover, the spikes generated by the outer packer seem much less humpy, as most inner packed files are smaller than the original unpacked files.

### 6.4.1.4 Incorporating the Off-line Hump-and-Dump Technique with the Section Information

The Hump-and-Dump patterns displayed in Figures 6.2 - 6.11 indicate the OEP area. However, it is not clear where the OEP's exact position is. To address this question, this research further improves the Hump-and-Dump technique by including PE file's section information (refer to Section 2.2.2.3).

Most packers unpack the original file and store them in a different section to where the

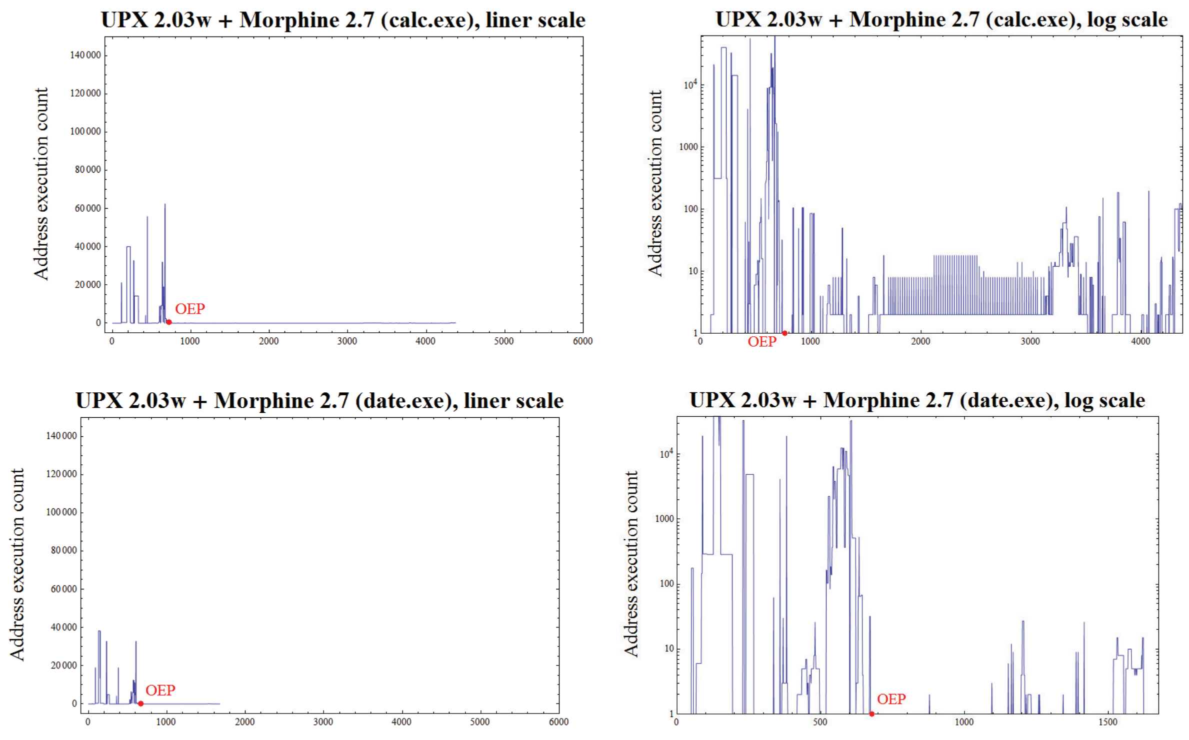


**Figure 6.12:** Multi-packed UPX 2.03w + Morphine 2.7 unpacking trace address histogram

unpacking stub is stored. Thus, after unpacking, the program needs to take an intersection jump (also called Far jump) to transfer control to the original file. Therefore, an OEP instruction is usually executed after an intersection jump though an intersection jump does not necessarily mean jumping to an OEP.

Figures 6.13 - 6.23 display the ordered address execution histograms with the section information of the data described in the Table 6.1. In the graphs, EIPs of different sections are displayed in different colors and line styles. For example, in Figures 6.13, the section of the unpacker is displayed in the purple thick line while the section of the original file is displayed in a blue dashed line.

Figures 6.13 - 6.23 demonstrate that the OEP can be precisely located by incorporating the Hump-and-Dump technique with the section information. In these graphs, the OEP occurs not only in the Hump-and-Dump pattern but also at the start of the new section.

For multi-packed files, as shown in Figures 6.23, the OEP is located in the last section after the last large spike.

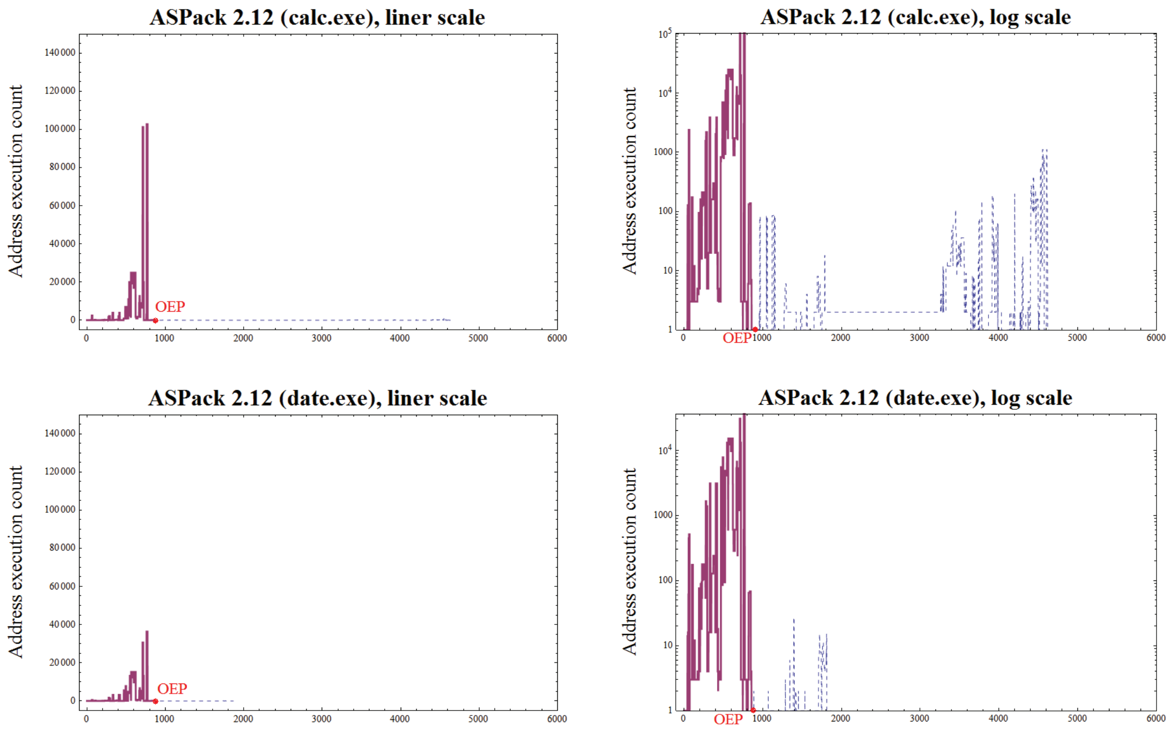


**Figure 6.13:** ASPack 2.12 unpacking trace address histogram with section information

## 6.4.2 On-line Experiments and Results

To illustrate the idea behind the on-line approach, the on-line algorithm has been implemented as a C++ IDA plugin. An experiment has been carried out on the UPX packer. During the experiment, we define the "hump" as when the address frequency is above 30000 and the "flat section" as where there are 20 consecutive addresses which only occur once. A screen shot of the plugin execution, Figure 6.24, shows that this unpacking strategy can efficiently and effectively locate the OEP area as the plugin stops at address 0x01020E4D, an important POPA instruction which is just five instructions before the final jump to the OEP. By then, 155 instructions have been executed. In other words, there are only 155 records stored in the memory. We need less than 1KB of memory to hold the necessary data structures, and computation is similarly cheap (and compatible with dynamic-translation emulators).

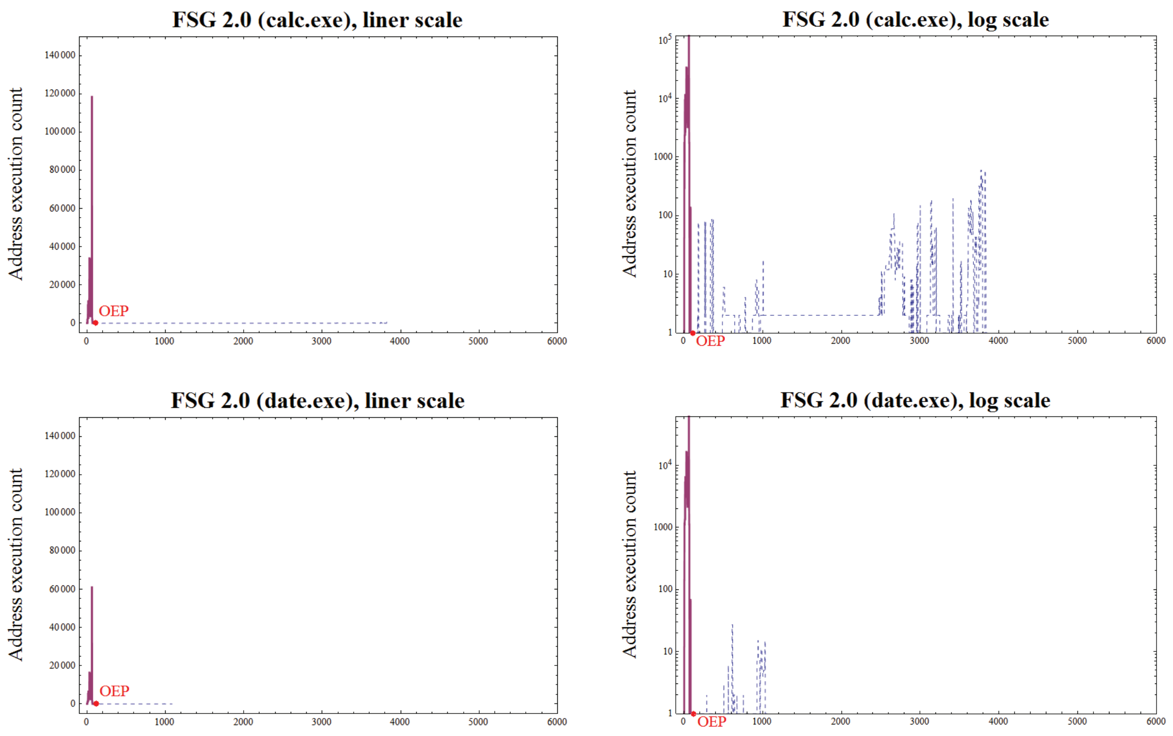


**Figure 6.14:** FSG 2.0 unpacking trace address histogram with section information

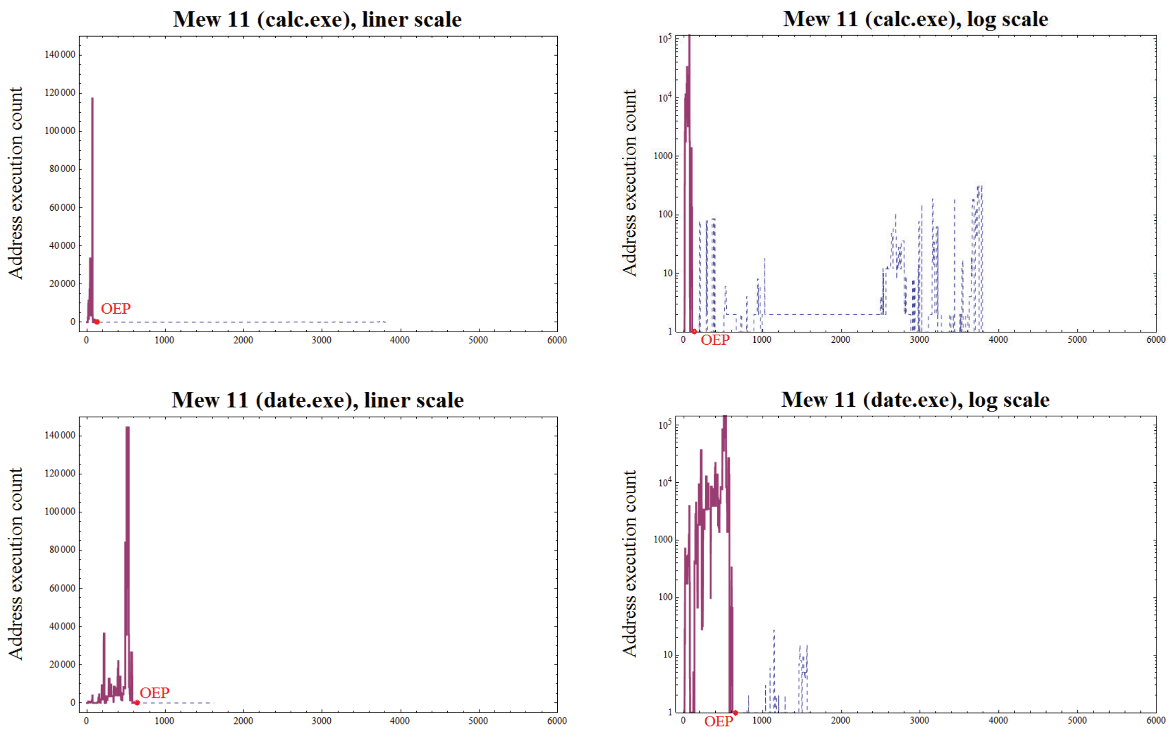


**Figure 6.15:** Mew 11 unpacking trace address histogram with section information

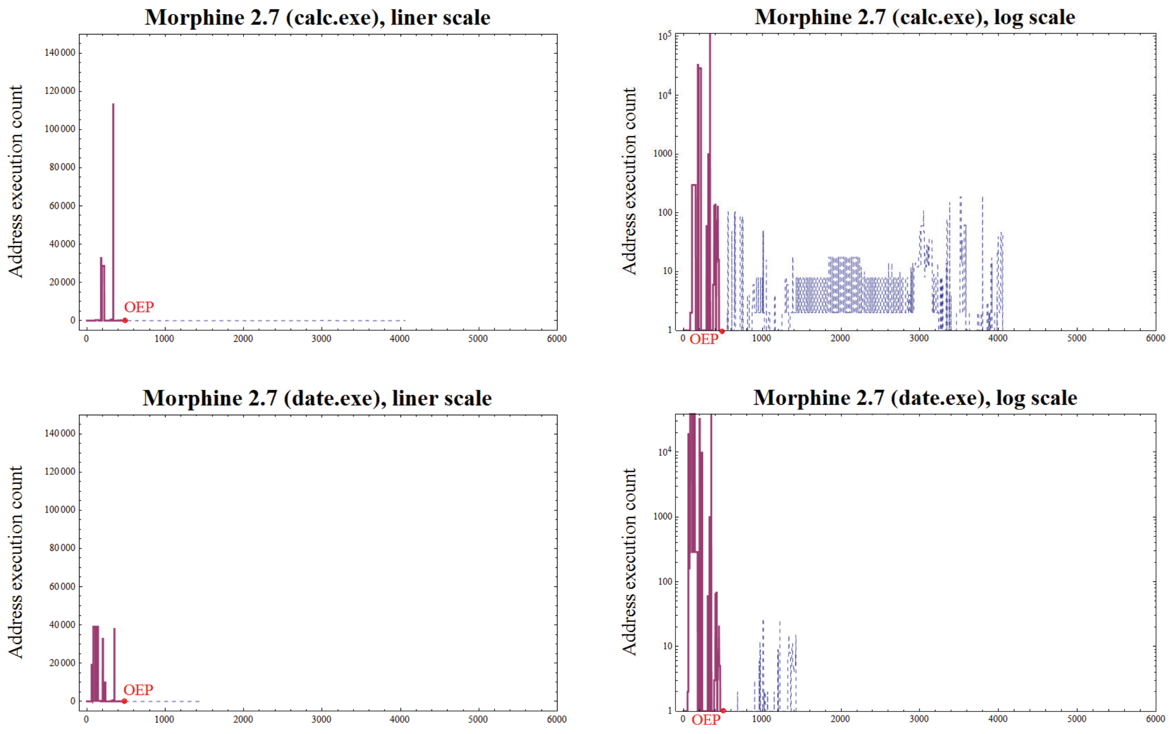


**Figure 6.16:** Morphine 2.7 unpacking trace address histogram with section information

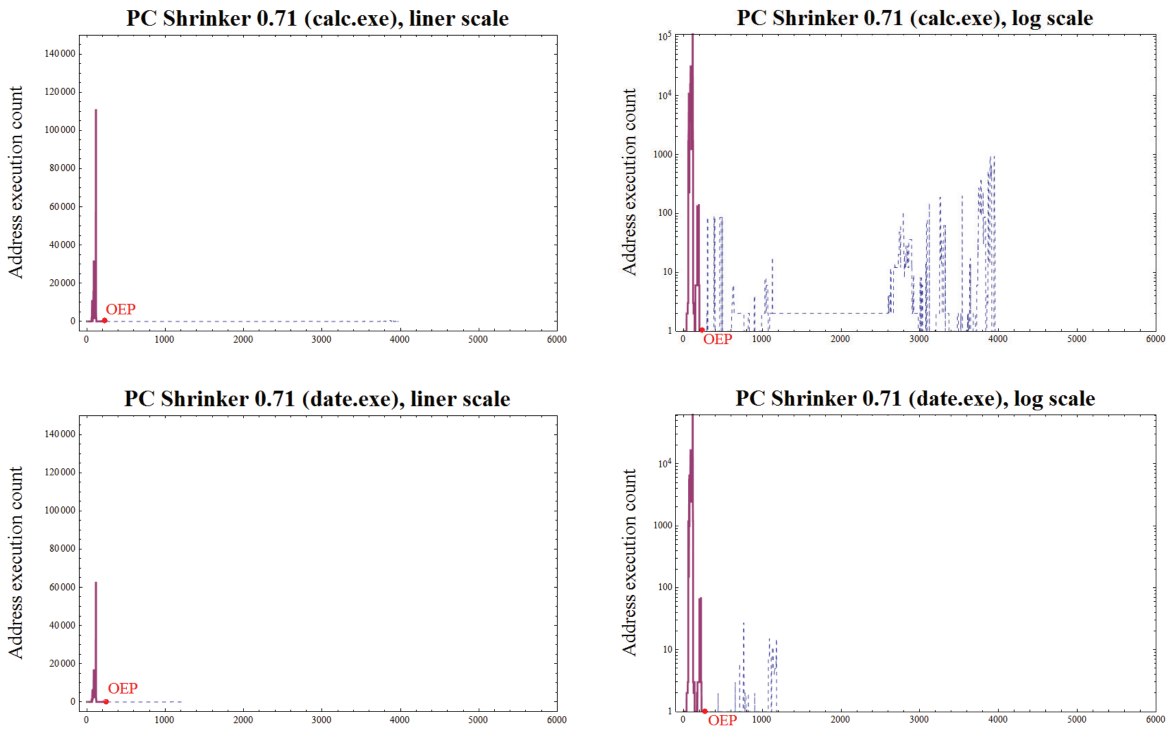


**Figure 6.17:** PC Shrinker 0.71 unpacking trace address histogram with section information

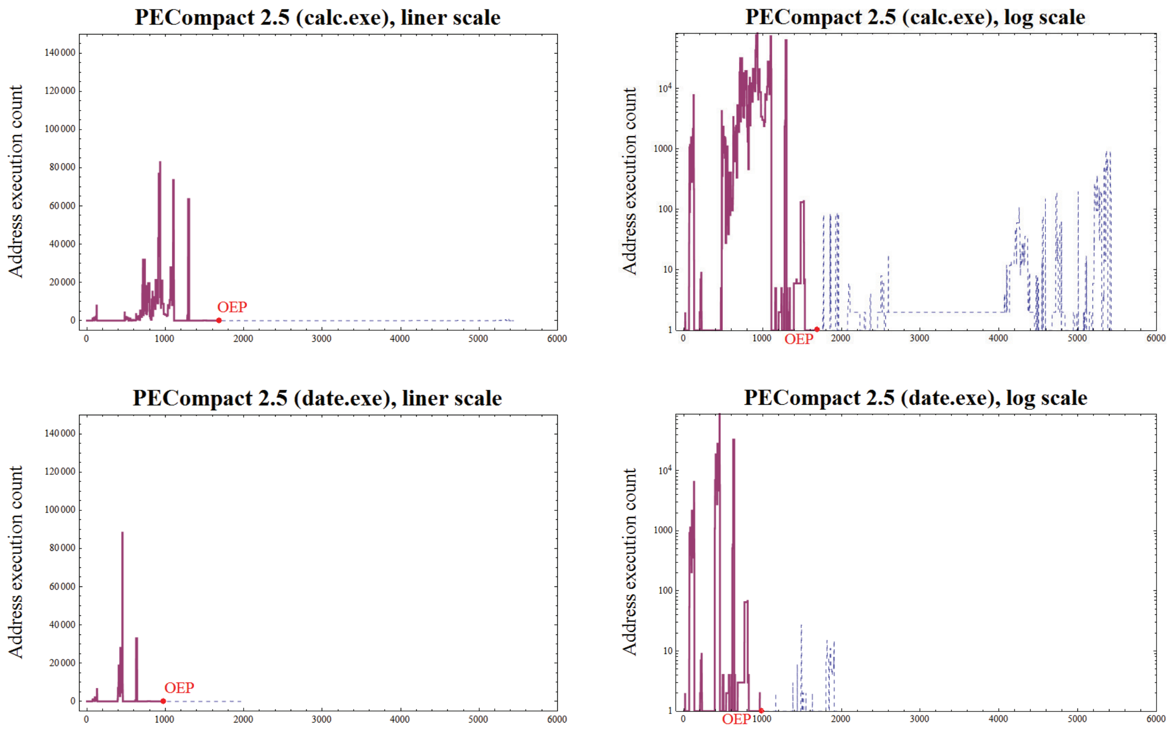


**Figure 6.18:** PECompact 2.5 unpacking trace address histogram with section information

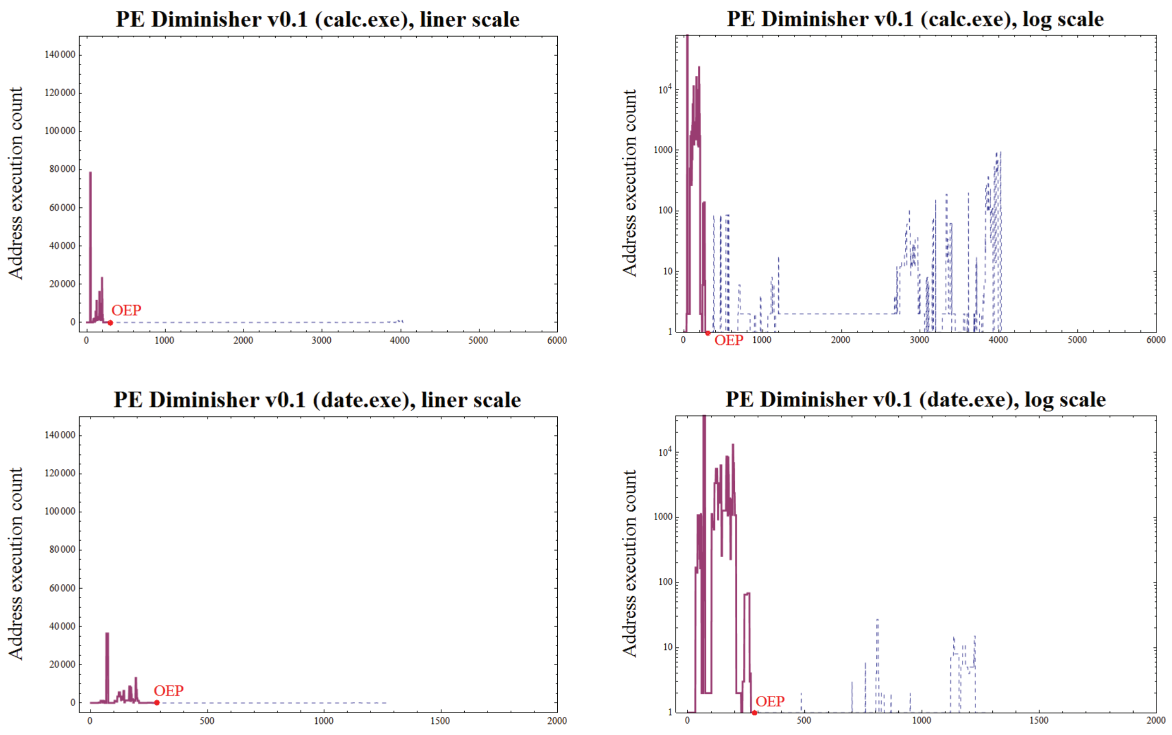


**Figure 6.19:** PE Diminisher v0.1 unpacking trace address histogram with section information

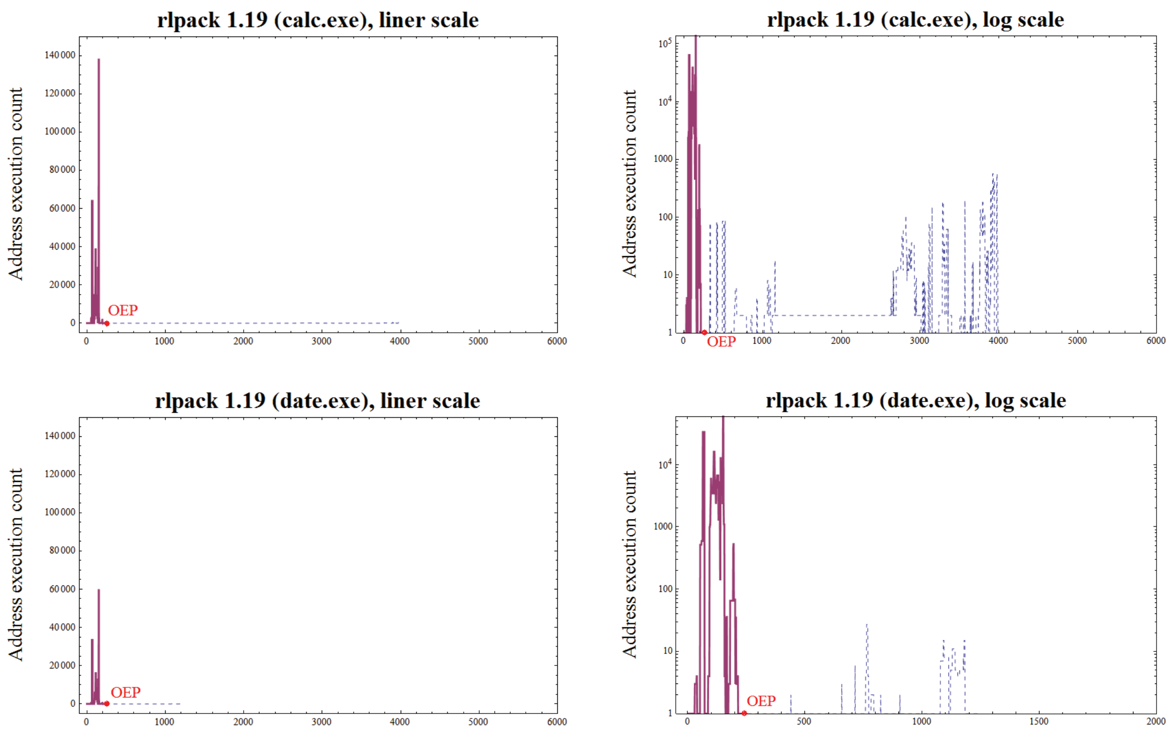


**Figure 6.20:** RLPack 1.19 unpacking trace address histogram with section information

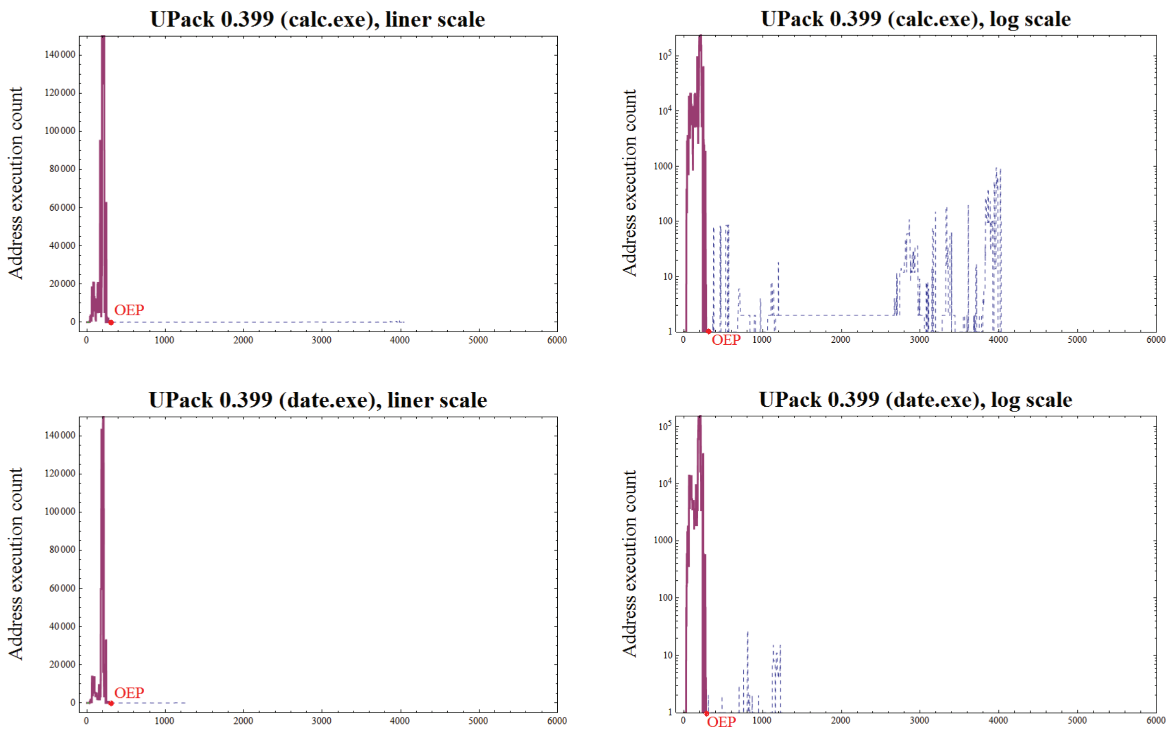


**Figure 6.21:** UPack 0.399 unpacking trace address histogram with section information

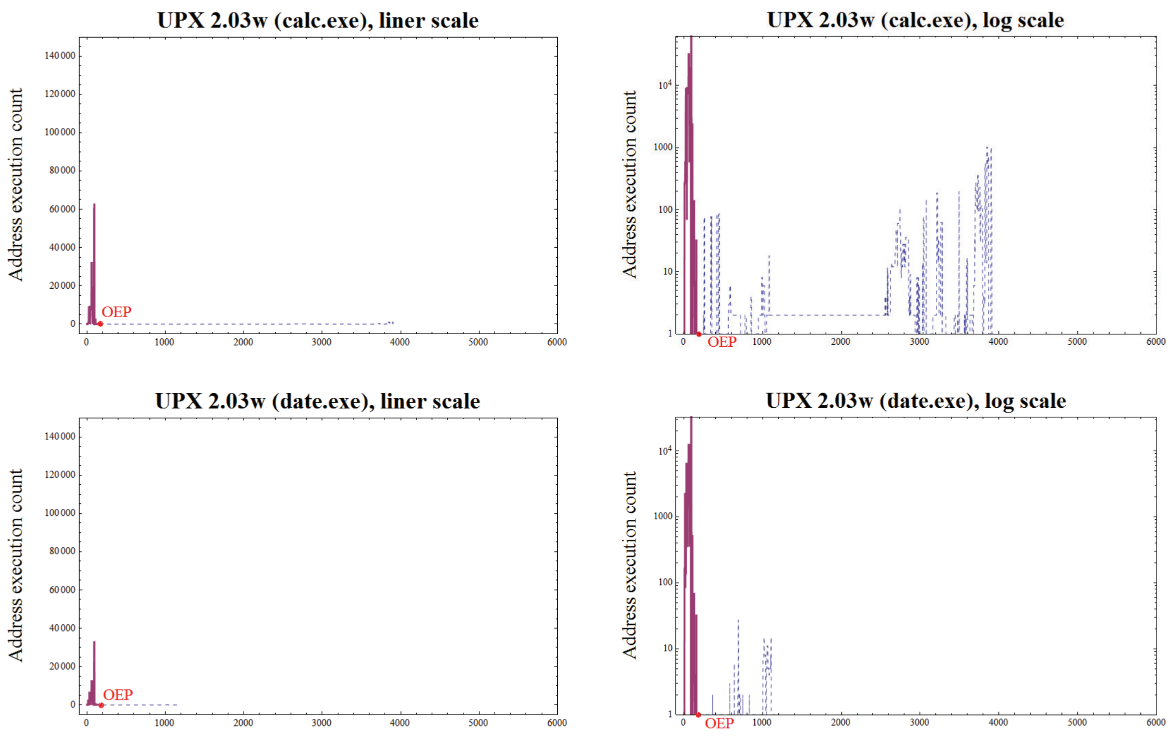


**Figure 6.22:** UPX 2.03w unpacking trace address histogram with section information

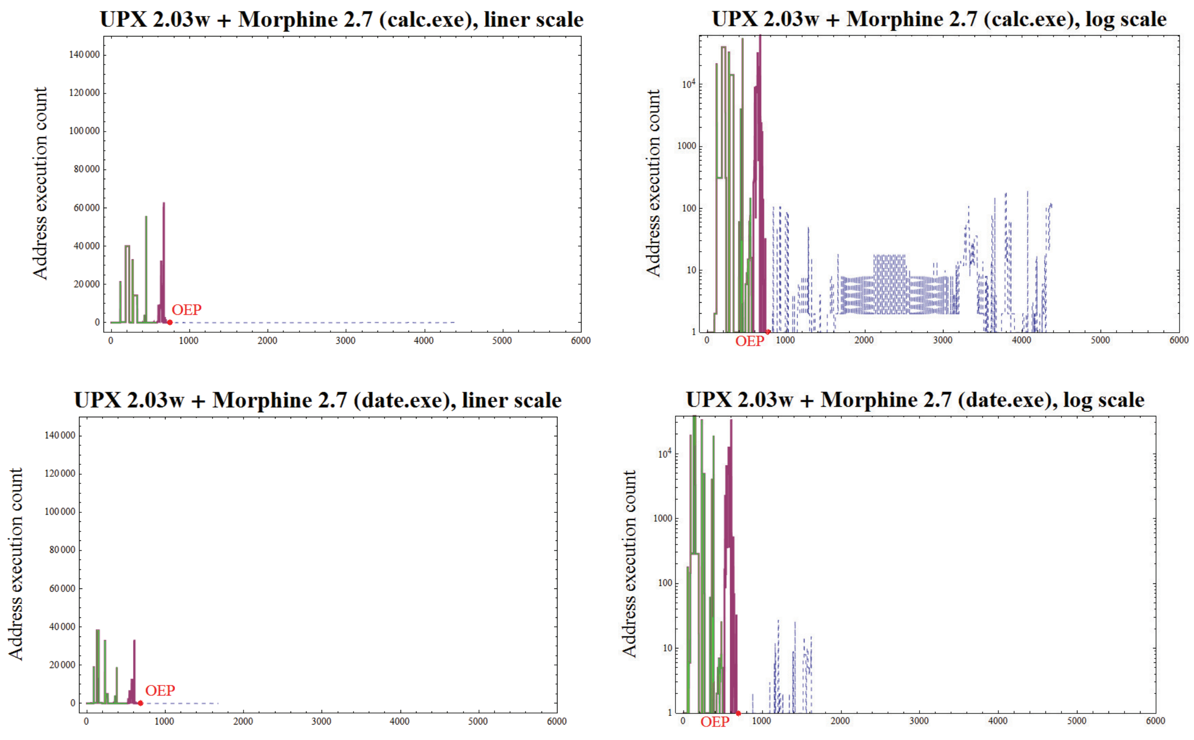
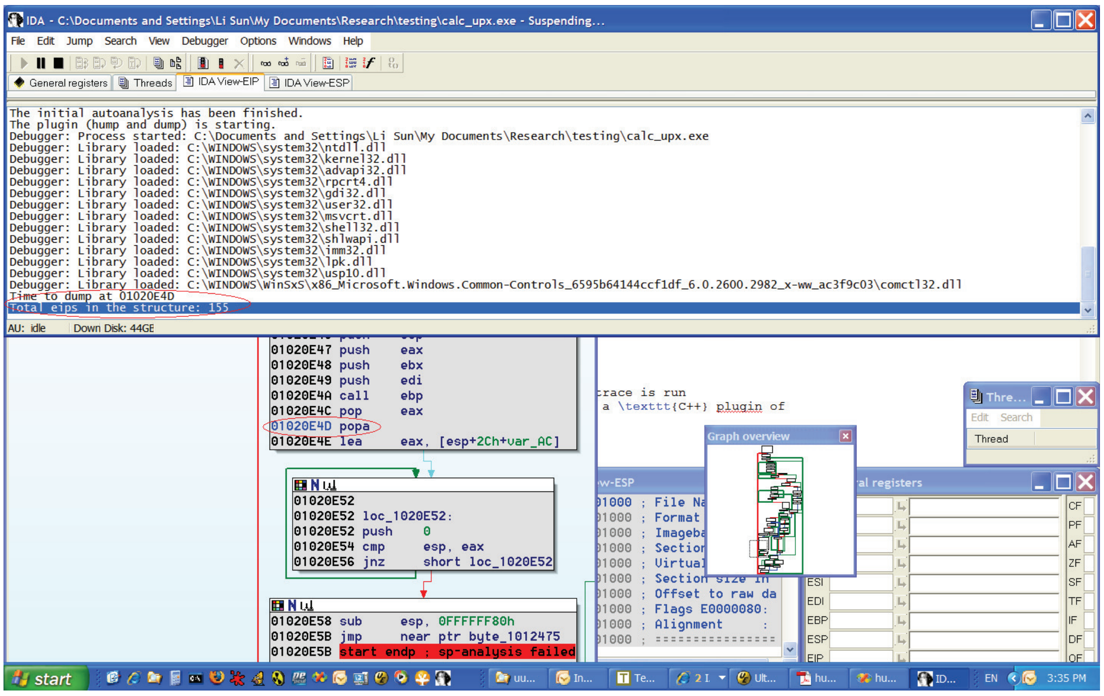


**Figure 6.23:** Multi-packed UPX 2.03w + Morphine 2.7 unpacking trace address histogram with section information



```
The initial autoanalysis has been finished.
The plugin (hump and dump) is starting.
Debugger: Process started: C:\Documents and Settings\Li Sun\My Documents\Research\testing\calc_upx.exe
Debugger: Library loaded: C:\WINDOWS\system32\ntdll.dll
Debugger: Library loaded: C:\WINDOWS\system32\kernel32.dll
Debugger: Library loaded: C:\WINDOWS\system32\advapi32.dll
Debugger: Library loaded: C:\WINDOWS\system32\rpcrt4.dll
Debugger: Library loaded: C:\WINDOWS\system32\gdi32.dll
Debugger: Library loaded: C:\WINDOWS\system32\user32.dll
Debugger: Library loaded: C:\WINDOWS\system32\msvcrt.dll
Debugger: Library loaded: C:\WINDOWS\system32\shell32.dll
Debugger: Library loaded: C:\WINDOWS\system32\shlwapi.dll
Debugger: Library loaded: C:\WINDOWS\system32\imm32.dll
Debugger: Library loaded: C:\WINDOWS\system32\lpk.dll
Debugger: Library loaded: C:\WINDOWS\system32\usp10.dll
Debugger: Library loaded: C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.2982_x-ww_ac3f9c03\comctl32.dll
Time to dump at 01020E4D
Total eips in the structure: 155
```

```
01020E47 push    eax
01020E48 push    ebx
01020E49 push    edi
01020E4A call    ebp
01020E4C pop     eax
01020E4D popa
01020E4E lea     eax, [esp+2Ch+var_AC]

01020E52
01020E52 loc_1020E52:
01020E52 push    0
01020E54 cmp     esp, eax
01020E56 jnz     short loc_1020E52

01020E58 sub     esp, 0FFFFFF80h
01020E5B jmp     near ptr byte_1012475
01020E5B start endp ; sp-analysis failed
```

```
trace is run
 a \texttt{C++} plugin of
```

```
01000 ; File Na
01000 ; Format
01000 ; Imageba
01000 ; Section
01000 ; Virtual
01000 ; Section size in
01000 ; Offset to raw da
01000 ; Flags E0000080:
01000 ; Alignment       :
01000 ; ================
```

IDA - C:\Documents and Settings\Li Sun\My Documents\Research\testing\calc_upx.exe - Suspending...
File Edit Jump Search View Debugger Options Windows Help
General registers
Threads
IDA View-EIP
IDA View-ESP
AU: idle
Down Disk: 44GE
Graph overview
Thread
ESI EDI EBP ESP EIP
CF PF AF ZF SF TF IF DF OF
start
3:35 PM

**Figure 6.24:** Using the Hump-and-Dump IDA plugin, without section check, to find the OEP for a UPX packed file

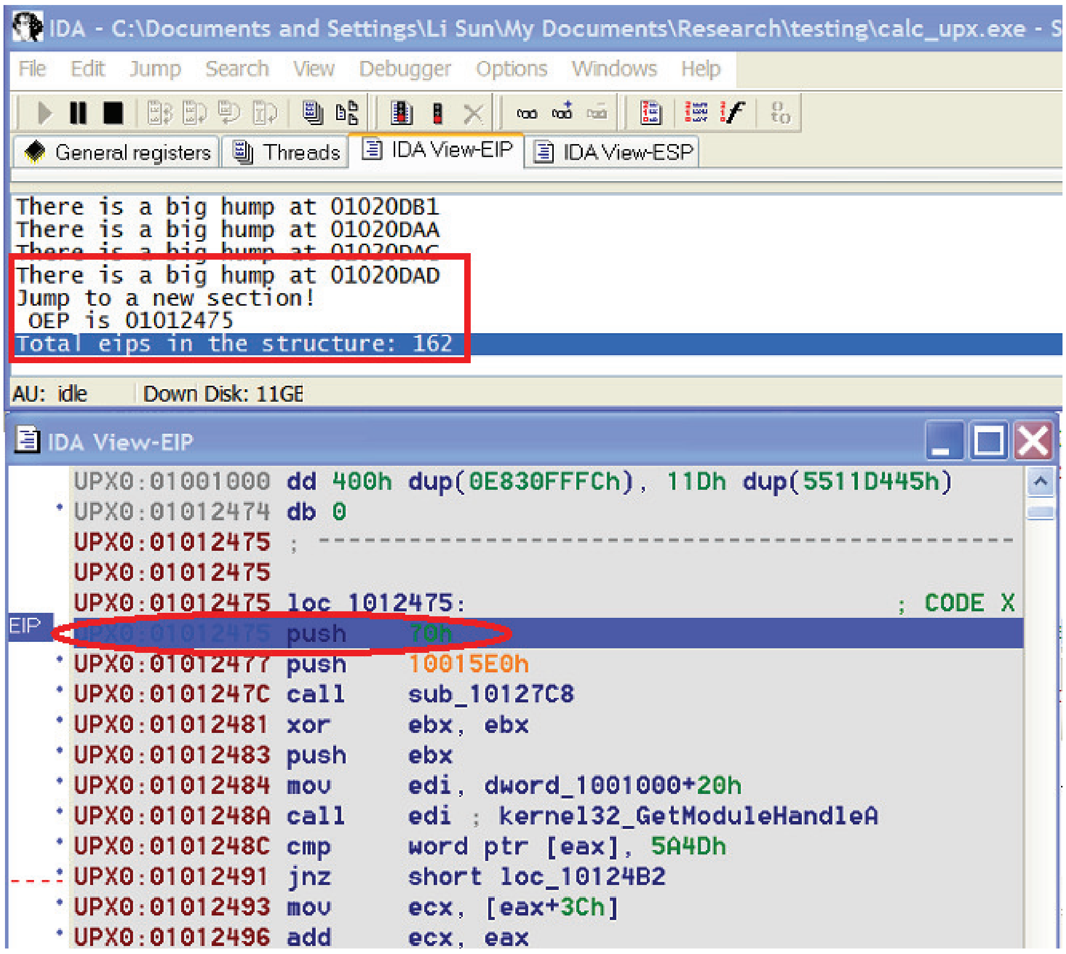
#### 6.4.2.1 Incorporating the On-line Hump-and-Dump Technique with the Section Information

The promising results obtained from the off-line experiments (Section 6.4.1.4) motivates us to incorporate the on-line Hump-and-Dump algorithm with the section information to locate the exact OEP. As shown in Algorithm 6.3, once the hump (main loop) occurs, instead of using a threshold of the flat section length, $T_{\text{flat}}$, to find OEP, the algorithm checks whether the new executed address belongs to the unpacking code section or not. If not, then the program must jump to a new section. As an intersection (Far) jump after a main loop usually indicates a control transferring to the OEP, this is a good time to stop the program and dump the memory.

```
Data   : Packed application
Result: Unpacked application
begin
    Start emulating the application ;
    Initialize a hash table $addrHist$ of which the key is the $eip$ ;
    Set hump flag $hump \longleftarrow false$ ;
    Set flat section counter $flatCount \longleftarrow 0$ ;
    Set threshold of the hump $T_{hump}$ ;
    Set threshold of the flat section which is repeated number of new addresses of
    height 1 $T_{flat}$ ;
    foreach $eip$ do
        $counter += 1$ ;
        if $addrHist.has\_key(eip)$ then
            $flatCount = 0$ ;
            $addrHist[eip] = (addrHist[eip][frequency] + 1, counter)$ ;
            if $addrHist[eip][frequency] > T_{hump}$ then
                $hump = True$ ;
                Retrieve the section information ;
            endif
        endif
        else
            $addrHist[eip] = (1, counter)$ ;
            if $hump$ then
                if $eip$ is in a new section then
                    Stop the application ;
                    Dump the memory ;
                endif
            endif
        endif
    end
end
```

**Algorithm 6.3:** Implementation of on-line Hump-and-Dump algorithm with section check

In Figure 6.25, we use this unpacking strategy to locate the exact OEP of a UPX packed file, at 0x01012475. An unpacked file can be obtained by dumping the memory at this moment. Also note that the strategy is very efficient as only 162 instructions have been executed.



```
There is a big hump at 01020DB1
There is a big hump at 01020DAA
There is a big hump at 01020DAC
There is a big hump at 01020DAD
Jump to a new section!
 OEP is 01012475
Total eips in the structure: 162
```

```
UPX0:01001000 dd 400h dup(0E830FFFCh), 11Dh dup(5511D445h)
UPX0:01012474 db 0
UPX0:01012475 ; ---------------------------------------------
UPX0:01012475
UPX0:01012475 loc_1012475:                          ; CODE X
UPX0:01012475 push    70h
UPX0:01012477 push    10015E0h
UPX0:0101247C call    sub_10127C8
UPX0:01012481 xor     ebx, ebx
UPX0:01012483 push    ebx
UPX0:01012484 mov     edi, dword_1001000+20h
UPX0:0101248A call    edi ; kernel32_GetModuleHandleA
UPX0:0101248C cmp     word ptr [eax], 5A4Dh
UPX0:01012491 jnz     short loc_10124B2
UPX0:01012493 mov     ecx, [eax+3Ch]
UPX0:01012496 add     ecx, eax
```

**Figure 6.25:** Using the Hump-and-Dump IDA plugin, with section check, to find the OEP for a UPX packed file

## 6.5 Summary and Discussion

In this chapter, we have explored an execution pattern that is followed by most packers, i.e., the unpacking activity involves massive loops such as decompression and decryption while instructions around the OEP are almost always only executed once, and shown that this pattern can be successfully exploited to create a generic unpacking method. Rather than having a signature and separate unpacking implementation for every packer variant, the method, called “Hump-and-Dump” uses a single implementation to be able to unpack many different kinds of packed files, regardless of the packer which was used or the complexity of the unpacking method (except for virtual machine protection systems.) This is an important innovation needed to combat the ever increasing number of packer variants.

The “Hump-and-Dump” approach uses a simple EIP trace and an ordered address execution histogram. The method has been tested on the ten most widely used packers such as UPX and Morphine. Results show that all tested packers display a pattern in their ordered address execution histograms. The pattern, in which large unpacking humps followed by a flat section of OEP bytes, helps finding out when unpacking activity has finished and where the OEP area is.

Though the binary file dumped at a rough OEP area is normally sufficient for AV researchers to carry out an in-depth analysis, this research further explored a way to determine the exact OEP. To do that, experiments combining the “Hump-and-Dump” technique with the section information have been carried out. In this case, the method seeks an intersection jump after the Hump-and-Dump pattern in an ordered address execution histogram. Results show that this strategy precisely locates the exact OEP. A fully unpacked file can be obtained by dumping the memory at this moment.

“Hump-and-Dump” provides a generic solution to the main problem in the third stage of packer analysis, namely unpacking (please see Figure 1.1 again). However, in the arms race between malware writers and anti-malware researchers, it is understandable that malware/packer writers may be able to counter this unpacking method. The “Hump-and-Dump” approach is based on two observations that (a) even in a packed program, the OEP bytes are almost always only executed once, and (b) most packers unpack the original program to an area of memory which has not been previously executed. Though most existing packers have these features, it is unavoidable that malware/packer writers will be able to modify these two features of a packer. Besides, packer writers can implement a fake 'hump' in the unpacking routine to disturb our OEP detection. However, this will definitely slow down the unpacking process and therefore delay the malware execution.

Apart from the preliminary gains described here, especially the benefit of using an ordered address execution histogram for unpacking, the performance of the tracer, which was used to collect the EIPs, needs to be addressed in further research. There are many armoring techniques applied in the packer to hinder the packer analysis. How to bypass these armoring techniques in a tracer is a big question. Chapter 7 will address this issue.

# Chapter 7

# An Automatic Anti-anti-VMware Technique for Improvement of Unpacking Performance

The previous chapters have addressed problems in three packer analysis tasks, namely packer detection, classification and unpacking, and provided solutions for undertaking these tasks efficiently and effectively. We are aware that these tasks, especially the unpacking process and the execution of the unpacked file, need to be performed in a secure, contained environment. This is due to the fact that the unpacked original program might be malicious and will cause damage to the computer system. In this thesis, the VMware workstation (see Section 2.6) is used for all experiments.

The problem with VMware is that a large fraction of current generation malware employs various anti-VMware techniques for the purpose of resisting analysis. To make things worse, these anti-VMware techniques are applied not only in the payload itself, but also in the runtime packer that is used to disguise the malicious code.

In order to improve our unpacking performance, an automatic anti-anti-VMware technique has been developed and incorporated into the "Hump-and-Dump" unpacking strategy described in the previous chapter. With this anti-anti-VMware technique, judicious automated control of a debugger can successfully be used to slither around anti-VMware detections even in sophisticated packers, such as Themida.

## 7.1 Introduction

VMware Workstation virtualisation software is widely used by antivirus researchers for malware analysis. As described in Section 2.6, the advantages of virtualisation are significant. Malware authors are aware of this, and much modern malware attempts to detect the presence of VMware, and pursue some remedy if it is found. Remedies can include entering an infinite loop, crashing the application, shutting down the operating system, executing lots of decoy code, or simply exiting silently. This results in increased workload for AV researchers since it is hard to automate analysis under these conditions.

### 7.1.1 Anti-VMware Techniques

Though there are a lot of ways to detect virtual machines [35, 82], there is not a wide variety of methods employed by malware to detect VMware and the assembly code which describes the operation is quite typical. Some simple techniques for detecting VMware are searching for the VMware fingerprint in the register, MAC address, etc. Two common anti-VMware tricks that widely used by malware are called *Descriptor table* method and VMware *backdoor* method. This thesis focuses on these two tricks and their detailed description follows.

#### 7.1.1.1 Descriptor Table Method

The SIDT and SGDT instructions store the contents of the Interrupt Descriptor Table Register (IDTR) and Global Descriptor Table Register (GDTR) in a 6-byte memory location respectively while the SLDT instruction stores the content of the Local Descriptor

Table Register (LDTR) in a 16 or 32-bit general-purpose register or memory location. For each processor, there is only one set of IDTR, GDTR and LDTR registers. Therefore, when the virtual machine emulator is running, it must provide its own virtual set of these registers[96].

Based on this hardware virtualization issue, the Scoopy Doo [58] tool was developed to identify systems running under VMware. This tool uses SIDT, SGDT and SLDT operations to retrieve the values of the IDTR, GDTR and LDTR registers and then compare these actual values with predefined values which are related to some versions of VMware. Some predefined values the tool used are listed below.

```
IDT of system running inside of VMware version 3:   0xFFC6A370
IDT of system running inside of VMware version 4:   0xFFC17800
IDT of a native Windows XP and 2003 Server System:  0x8003F400
IDT of a native Windows 2000 Server System:         0x80036400

LDT of system running inside of VMware version 3:   0x3fa8
LDT of system running inside of VMware version 4:   0x4058
LDT of a native Windows XP and 2003 Server System:  0x0000
LDT of a native Windows 2000 Server System:         0x0000

GDT of system running inside of VMware version 3:   0xFFC05000
GDT of system running inside of VMware version 4:   0xFFC07000
GDT of a native Windows XP and 2003 Server System:  0x8003F000
GDT of a native Windows 2000 Server System:         0x80036000
```

Using the same basic idea, the RedPill [99] tool sets a threshold for the Interrupt Descriptor Table (IDT) to decrease the false positives. Using RedPill, a virtual machine emulator is assumed to be present if the value returned by (SIDT) exceeds the threshold.

However, according to the research performed by Quist and Smith [90], this approach is unreliable on multi-processor machines. They propose a different approach termed NoPill. The NoPill tool further extends Scoopy Doo and RedPill and uses the Local Descriptor Table (LDT) as the fingerprint for virtualisation. With NoPill, if the value of LDTR doesn't match a fixed value, a virtual machine is detected.

Other system instructions were explored for VMware detection. Quist [91] tested SMSW (store machine status word) on top of the NoPill to increase the detection accuracy. Omella [79] noticed that the retrieved value of TR (task register) by STR (store task register) from a native system and a virtual machine were different.

#### 7.1.1.2 VMware Backdoor I/O Port

The most reliable and therefore popular way to detect VMware is using the VMware backdoor [54]. The VMware backdoor is a communication channel used by VMware tools to communicate with the VMware running on the host machine. This channel offers a variety of guest-to-host and host-to-guest communication functions, including copy and paste operations. A backdoor function called "get VMware version" can be used to detect the presence of VMware. A typical disassembly of the relevant code is shown in Figure 7.1. A good example of this detection is a tool called Jerry developed by Klein [57]. Lallous [63] also provide a tool which can be used to detect VMware via VMware backdoor.

```
mov  eax,  564D5868h  ;‘VMXh’
mov  ebx,  0
mov  ecx,  0Ah
mov  edx,  5658h      ;‘VX’
in   eax,  dx
```

**Figure 7.1:** VMware backdoor code sample.

In Figure 7.1, the first line sets the EAX register to VMware's magic number 'VMXh'. The second line sets the command specific parameter to the EBX. This parameter can be any number except the magic number since the EBX will store the result later. The third line indicates that this function is to get VMware version. After passing the VMware specific port value 'VX' to the EDX, the 'IN' instruction is called.

The 'IN' instruction is a privileged instruction used by the `x86` processor to transfer data from an input port such as keyboard to the accumulator register. This instruction is not available for use by user written programs while in protected mode, unless the necessary privileges are enabled. Therefore, when a normal operating system executes this instruction, an exception named `‘‘EXCEPTION_PRIV_INSTRUCTION’’` will be caused. Then the exception handler which is the actual payload of the malware will be executed. However, if VMware is running, no exception is generated. Instead, the backdoor function is triggered by calling the 'IN' instruction. If the magic number ('VMXh') is returned to the EBX, then it is certain that the program is running inside VMware. The malware then either exits the program or runs as a normal program without executing the malicious code.

### 7.1.2 Multi-stage Anti-VMware

Though there aren't many anti-VMware methods, detection of these tricks is not straightforward for the AV researcher. That's because most, if not all, malware implements a variety of advanced techniques to obfuscate themselves executing these anti-VMware tricks. The issue therefore becomes when to look for such code during execution since the actual anti-VMware code may only be decrypted shortly before it is executed.

### 7.1.3 Related Work

Previous work has been carried out to disguise the VMware by altering VMware. O'Dea [77] stated that VMware provides an undocumented way to disable the "get VMware version" function. This can be done by setting below line to the VMware `.vmx` configuration file.

```
isolation.tools.getVersion.disable = ‘‘TRUE’’
```

Other `VMX` configuration options have been tested by Carpenter and his research team [14]. A list of options (see Figure 7.2) have been identified to control or eliminate behaviors that allow VMware detection.

```
isolation.tools.getPtrLocation.disable = ‘‘TRUE’’
isolation.tools.setPtrLocation.disable = ‘‘TRUE’’
isolation.tools.setVersion.disable = ‘‘TRUE’’
isolation.tools.getVersion.disable = ‘‘TRUE’’
monitor_control.disable_directexec = ‘‘TRUE’’
monitor_control.disable_chksimd = ‘‘TRUE’’
monitor_control.disable_ntreloc = ‘‘TRUE’’
monitor_control.disable_selfmod = ‘‘TRUE’’
monitor_control.disable_reloc = ‘‘TRUE’’
monitor_control.disable_btinout = ‘‘TRUE’’
monitor_control.disable_btmemspace = ‘‘TRUE’’
monitor_control.disable_btseg = ‘‘TRUE’’
```

**Figure 7.2:** VMware configuration options used to disguise VMware.

*Binary patching* is another way to disable the VMware detection. The best-known implementation of binary patching is Kortchinsky’s “Honey-VMware patch” [62] for his French HoneyNet Project. This project disables the I/O backdoor by modifying the magic number (‘VMXh’) in the VMware binary file. Another binary patch tool VMmutate was developed by Liston and Sloudis [66]. The VMmutate scans a VMware disk image and patches the magic value it found. In order to reduce the false positives, the tool also looks at the magic value’s context before altering it.

It is quite obvious that binary patching is not suitable for countering multi-stage packed malware. In addition, neither altering the configuration options nor binary patching is sufficient to hide VMware. If employed, certain VMware features, such as dragging and dropping files between guest and host, may also be disabled unexpectedly. Moreover, it is unknown whether there are side-effects along with the change of these undocumented options or the false detected magic number.

We propose a reliable anti-anti-VMware method without compromising VMware's functionality. This method dynamically locates the anti-VMware tricks and patches the VMware detection result, for both VMware backdoor detection and descriptor table method, instead of simple binary patching on the VMware binary file. It is applicable for a multi-stage packed malware and therefore can be incorporated into our 'Hump-and-Dump' unpacking plugin. The rest of this chapter will describe the implementation of the proposed tool, the experiments, and the result.

## 7.2 Implementation and Experiments

The tool described here is an IDA C++ plugin that allows the program to automatically bypass above two popular anti-VMware detections, namely descriptor table method and VMware backdoor method, whilst running in an IDA debugger.

### 7.2.1 Dynamical Patching

When the tool is running with the IDA debugger, it will scan the memory of the running malware application. If any of the two above-mentioned main types of the anti-VMware signature is found, the tool will set a hardware breakpoint on the main instruction such as 'IN' or 'SIDT'. Once this breakpoint is triggered during the analysis, the tool automatically patches the execution result in corresponding register or memory location with pre-defined value which belongs to the native machine. For example, after executing the VMware backdoor function, the magic value returned to 'EBX' will be changed to '0'. Therefore, one can easily fool the malware/packer to think it is running in a native machine. With this approach, there is no need to worry about what the following comparison result will be and which jump branch needs to be patched.

The question that needs to be addressed is when to scan the file to find out anti-VMware tricks. These tricks normally are encrypted since most malware is packed one or more times. Moreover, some packers may even contain anti-VMware tricks. Fortunately, these tricks must be decrypted before they can be executed. Therefore, the tool provides multiple scanning through the whole debugging process. It rescans the current memory section whenever one of below two events happens:

- *jump to a new block of code (also called 'Far jump')*, or
- *after an exception is handled.*

We define the first rescanning condition since most packers unpack the code into a new section. After unpacking, the malicious code and all anti-VMware tricks are decrypted. The execution is then transfered to the original entry point (OEP). The second rescanning condition is set because the application may have changed the execution flow and starts to execute the real code after handling the exception. Both 'Far jump' and exception handling are important moments for the program to stop the execution and start the scanning. The execution flow of the plugin can be seen in Algorithm 7.1.

### 7.2.2 Experiments and Results

#### 7.2.2.1 Anti-anti-VMware experiments and Results

Three types of experiments have been performed to evaluate the proposed anti-anti-VMware technique. All experiments were run in VMware 5.5.3 and the host operating system is Windows XP Professional 2002, service pack 2.

Firstly, experiments were run on simple VMware detection tools, such as Lallous's [63] VMware backdoor detection tool and NoPill tool [90] which implements the descriptor table method described previously.

Secondly, packed VMware detection tools were tested. For example, a UPX packed NoPill. Moreover, two samples of malware, win32/Moiling.A and win32/DlAnsein.A, were chosen from the VET database[1].

```
Data  : Running application
Result: Running application with anti-VMware tricks patched
begin
    Step 1:
    for each code line of this section do
        Search the running memory for JMPs and anti-VMware tricks ;
        if found then
            Set hardware breakpoint (BP) on main instruction ;
        endif
    endfor
    Step 2:
    Continue debugging;
    Step 3:
    if catch exception then
        Find the code section of the exception handler ;
        Goto Step 1;
    endif
    if BP is trigged then
        if anti-VMware tricks then
            Patch execution result;
        endif
        if JMPs then
            if Jump to another section then
                Goto Step 1;
            endif
        endif
        Goto Step 2;
    else
        Goto Step 2;
    endif
end
```

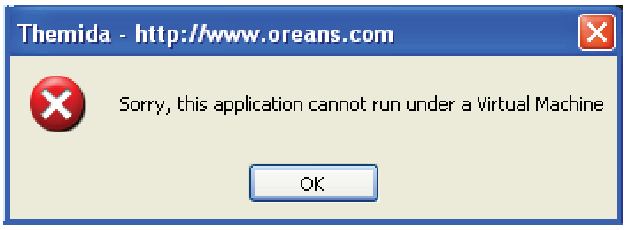
**Algorithm 7.1:** Execution flow of the anti-anti-VMware plugin.

Win32/Moiling.A is a Trojan that can be instructed to display messages and pop-ups as well as direct users to certain websites. The sample we used was packed with WinUpack 0.38 which is a multi-stage packer. Before it executes its payload, it uses the VMware backdoor to detect the VMware. If it gets the magic number which means it is

[1]This database is the property of CA

running inside a VMware, the program simply stops. Win32/DlAnsein.A is a Trojan that automatically downloads the win32/Ansein virus. The sample is packed with FSG2.0. After the unpacking, it checks the VMware presence via the SIDT instruction. After applying a decryption loop on the result of SIDT, the actual value is compared with a hard-coded threshold. If it exceeds the threshold, the VMware is detected. The virus then stops execution and exits.

Lastly, we tested several VMware-incompatible versions of the multi-stage packer, Themida. Themida is a powerful software protection system which employs its own unique SecureEngine technology [115]. SecureEngine provides a variety of advanced techniques to protect an application from being cracked. These techniques include anti-debugger, anti-monitor, anti-VMware/Virtual PC, etc. For the purpose of anti-VMware, it implements the VMware backdoor method to detect the presence of the VMware. Once the result is positive, Themida pops up a warning message *"Sorry, this application cannot run under a Virtual Machine"* (see Figure 7.3) and then exits.


Themida - http://www.oreans.com

Sorry, this application cannot run under a Virtual Machine

**Figure 7.3:** Themida running result in VMware

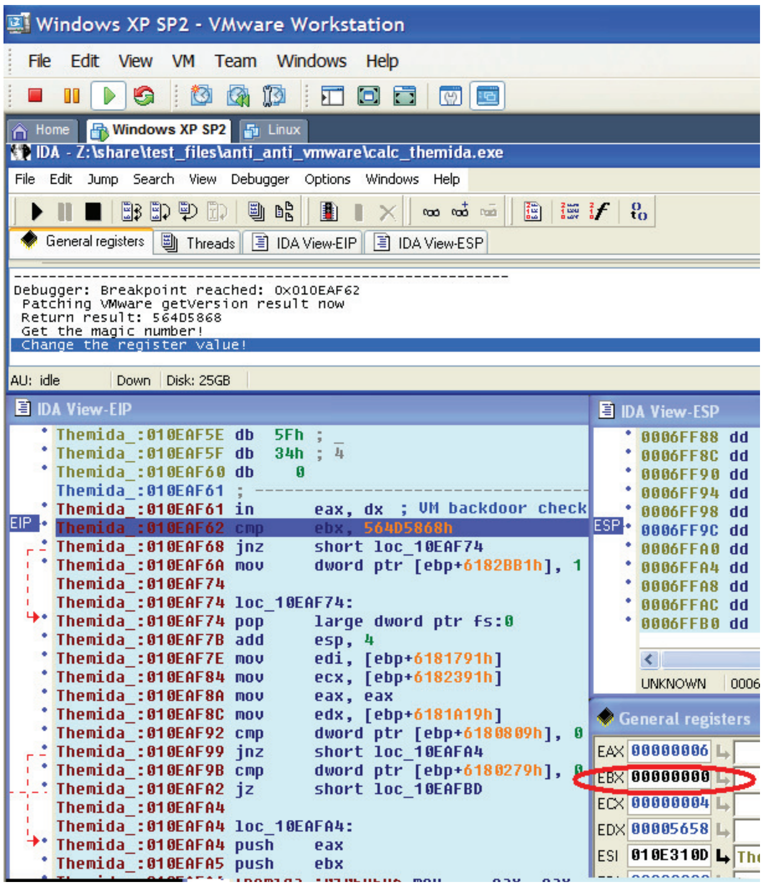
Like all other sophisticated packers, Themida implements polymorphic encryption layers as well. While encryption layers keep the code totally encrypted and only decrypt the code when it needs to be executed by the CPU, each polymorphic layer has a different algorithm and encryption key making it impossible to recognize where an encryption layer starts and finishes.

As shown in Table 7.1, using the anti-anti-VMware technique, our IDA plugin found and disabled VMware detections in all cases tested. For example, when we run VMware-incompatible Themida packed application in the VMware, the original return result of the VMware backdoor check should be "564D5868h". However, as shown in Figure 7.4, this result has been automatically patched with zero. Therefore, this Themida packed file can be smoothly executed under the VMware environment.

| **Experiment type** | **Testing sample** | **Used packer** | **Anti-VMware method** | **Patched?** |
|---|---|---|---|---|
| Simple anti-VMware tool | Lallous's tool | None | Backdoor | √ |
| | NoPill | None | Descriptor table | √ |
| Packed anti-VMware tool | NoPill | UPX 2.03w | Descriptor table | √ |
| | Win32/Molling.A | WinUpack 0.38 | Backdoor | √ |
| | Win32/DLAnsein.A | FSG 2.0 | Descriptor table | √ |
| Packer implements anti-VMware trick | calc.exe | Several VMware-incompatible versions of Themida | Backdoor | √ |

**Table 7.1:** Experimental results of anti-anti-VMware plugin

The tool is efficient as well. As described above, REFORM only rescans the current memory section when there is a FAR jump to a new block of code, or after an exception is handled. In the experiments described in this paper, REFORM requires two rescans on average, with an overall execution time of several seconds. Malware with more "layers" will cause the scanning time to increase, but not by much. Moreover, while the tool works in conjunction with a debugger, the overhead caused by REFORM relative to the debugger is small.


Windows XP SP2 - VMware Workstation
File Edit View VM Team Windows Help
Home
Windows XP SP2
Linux
IDA - Z:\share\test_files\anti_anti_vmware\calc_themida.exe
File Edit Jump Search View Debugger Options Windows Help
General registers
Threads
IDA View-EIP
IDA View-ESP
AU: idle
Down
Disk: 25GB
UNKNOWN
General registers

```
----------------------------------------------------------
Debugger: Breakpoint reached: 0x010EAF62
 Patching VMware getVersion result now
 Return result: 564D5868
 Get the magic number!
 Change the register value!
```

```
    Themida_:010EAF5E db   5Fh ; _
    Themida_:010EAF5F db   34h ; 4
    Themida_:010EAF60 db     0
    Themida_:010EAF61 ; ------------------------------
    Themida_:010EAF61 in      eax, dx  ; VM backdoor check
EIP Themida_:010EAF62 cmp     ebx, 564D5868h
    Themida_:010EAF68 jnz     short loc_10EAF74
    Themida_:010EAF6A mov     dword ptr [ebp+6182BB1h], 1
    Themida_:010EAF74
    Themida_:010EAF74 loc_10EAF74:
    Themida_:010EAF74 pop     large dword ptr fs:0
    Themida_:010EAF7B add     esp, 4
    Themida_:010EAF7E mov     edi, [ebp+6181791h]
    Themida_:010EAF84 mov     ecx, [ebp+6182391h]
    Themida_:010EAF8A mov     eax, eax
    Themida_:010EAF8C mov     edx, [ebp+6181A19h]
    Themida_:010EAF92 cmp     dword ptr [ebp+6180809h], 0
    Themida_:010EAF99 jnz     short loc_10EAFA4
    Themida_:010EAF9B cmp     dword ptr [ebp+6180279h], 0
    Themida_:010EAFA2 jz      short loc_10EAFBD
    Themida_:010EAFA4
    Themida_:010EAFA4 loc_10EAFA4:
    Themida_:010EAFA4 push    eax
    Themida_:010EAFA5 push    ebx
```

```
    0006FF88 dd
    0006FF8C dd
    0006FF90 dd
    0006FF94 dd
    0006FF98 dd
ESP 0006FF9C dd
    0006FFA0 dd
    0006FFA4 dd
    0006FFA8 dd
    0006FFAC dd
    0006FFB0 dd
```

```
EAX 00000006
EBX 00000000
ECX 00000004
EDX 00005658
ESI 010E310D
```

**Figure 7.4:** Application of anti-anti-VMware plugin on Themida

### 7.2.2.2 Improvement of Unpacking Performance

Anti-anti-VMware technique was implemented in the 'Hump-and-Dump' tracer described in Chapter 6 to improve the unpacking performance. Figure 7.5 is the experimental result of real Win32/DIAnsein.A sample unpacking. The Win32/DIAnsein.A sample is packed with FSG and implements VMware detection. It stops execution if it finds that the program is running inside a VMware. As shown, our improved version of the 'Hump-and-Dump' plugin can successfully bypass this VMware detection and trace all execution to identify the OEP.

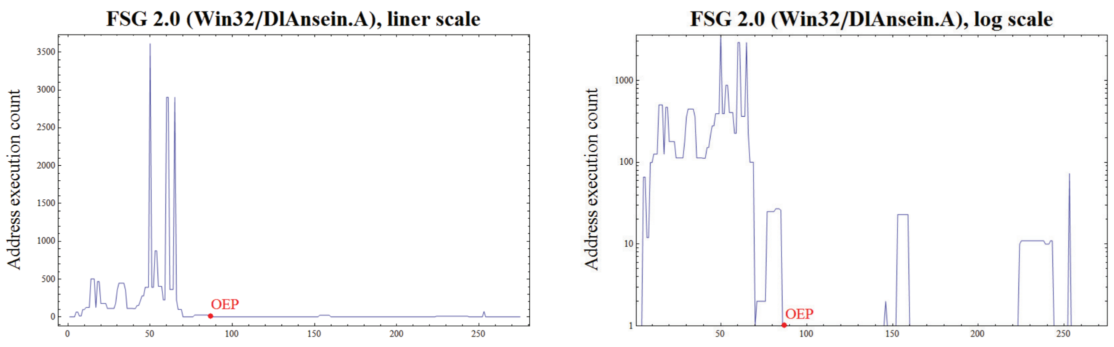


**Figure 7.5:** Win32/DlAnsein.A unpacking trace address histogram

## 7.3 Summary and Discussion

This chapter discussed two main types of anti-VMware tricks and presented an automatic anti-anti-VMware technique that thwarts the VMware detections. The benefit of this approach is twofold. It scans anti-VMware signatures at various stages of execution, which means it is applicable for a multi-stage packed malware or a multi-stage packer. Secondly, it patches the VMware detection result automatically and therefore achieves the anti-anti-VMware purpose. Thus, this technique provides a fast and reliable anti-anti-VMware approach.

Automated and scalable anti-malware tools will be essential to keeping up with the accelerating growth in malware. With increasing numbers of malware/packers using anti-VMware techniques, and anti-malware scanners relying on execution in virtual environments, an anti-anti-VMware method that can be automated, such as the dynamic scanning and patching technique described in this paper, will be an essential component of automated packer analysis and unpacking systems.

This anti-anti-VMware technique can be extended to apply to other virtual machine detection methods. Moreover, with the growth of anti-analysis tricks employed in the packers and malware, the dynamic scanning and patching technique can be scaled to resist these tricks as well. For example, some packers detect the presence of two popular monitors, Filemon and Regmon, and pursue some remedy if they are found. These techniques are called anti-filemon and anti-regmon tricks. In analogy with the work presented in this chapter, our ‘Hump-and-Dump’ unpacking tool can be upgraded to dynamically scan and patch these two tricks.

# Chapter 8

# Conclusions and Future Research

## 8.1 Conclusions

*This dissertation provides a complete solution to malware packer analysis.* Unlike previous attempts that mainly rely on human expertise and/or code simulation, the newly developed system, REFORM, applies techniques including reverse engineering, pattern recognition, information theory and statistical methods, and successfully solves three main packer analysis problems using novel techniques. The system is able to detect the packed code using a *local randomness test*, classify the packer family based on file's randomness profile and unpack the packed file via the *Hump-and-Dump* generic unpacking strategy. Furthermore, the system applies an automatic anti-anti-VMware technique to thwart the VMware detections implemented by malware, enabling the malicious code to be executed in a safe execution environment, VMware, without the malware altering its behavior. We argue that such an automatic packer analysis system is necessary to address the scalability problem faced by malware analysis. REFORM is an important step towards automated anti-malware systems.

### 8.1.1 Detecting the Packed Code Using Local Randomness Test

Randomness tests provide a fast way to estimate whether a file is packed or not. This is because the employment of compression and/or encryption in a packer brings the bytes closer to the appearance of a sequence of random bytes. However, most previous algorithms simply give an overall randomness value for the entire file or the whole section, such as the data section. Using these tests, the 'random' character of a small set of compressed data will be diluted by a large set of 'non-random' data, such as PE header and code.

In this thesis, we develop a novel fast randomness test, called *local randomness test*, that preserves local detail of a packed file. The test applies a modified version of Huffman encoding to represent bytes in the file and measures the amount of 'randomness' in different parts of a sample executable program using the corresponding encoding values. A plot of the randomness profile clearly shows areas of high randomness and low randomness across the file (see Figures 4.7-4.13 in Chapter 4). While the low, varying sections of the plot tend to be generated by code and header sections(in the case of a PE file), the high, flat sections often indicate encrypted or compressed data. Therefore, the plot can be a very effective aid to find packed data or even cryptographic keys in a sea of more structured data. Moreover, each packer has a distinctive randomness profile, allowing a visual assessment of packer classification by either an operator or an automatic system.

### 8.1.2 Packer Classification System Based on Randomness Profile

With the dramatically increased number of new packers and existing packer variants, it is essential to have a computer-based packer classification system. Instead of a traditional byte sequence signature based system that requires many human efforts, the new system should be able to identify packers automatically, with no AV researcher's input. Here, we present a *fast yet effective* packer classification system which applies pattern recognition techniques. In this approach, the low randomness profile of the packer is extracted and then passed to a statistical classifier.

Our work starts with investigating whether the randomness profile obtained from the local randomness test contains sufficient information for classifying packers. We refine the sliding window randomness test algorithm and conduct experiments to give guidance for optimal parameter settings. Then we evaluate the effectiveness of four pattern recognition algorithms for classifying packers using the randomness profile information. Experiments were carried out on a large set of real malware samples. Our work demonstrates that the randomness profile combined with strong pattern recognition algorithms can be used to produce a highly accurate packer classification system on real life data.

The system also reveals that the low randomness profile of a packed file, normally produced by the PE header and unpacking stub, contains the most important packer's information for classifying packers. Thus it is very useful in distinguishing between families of packers.

### 8.1.3 Generic Unpacking Using An Ordered Address Execution Histogram

Unpacking is a vital step to successful malware analysis and detection. Packers are not only used by malware to disguise itself from AV scanners, but also applied by legitimate software for protection against cracking. Therefore, once a packed sample is received, the AV scanner needs to firstly unpack the sample and then scan for malicious code. If it is a malware, AV researchers need to undertake further malware analysis.

As the variety of packing programs grows, their level of sophistication also increases. A generic unpacking technique that is capable of unpacking every type of packer without knowing which PE packer is used and what encryption/compression algorithm it utilizes, is essential to keeping up with the accelerating growth in packed malware.

This dissertation presents a new statistical approach for generic unpacking. The algorithm is based upon the dual observation that (a) even in a packed program, the OEP bytes are almost always only executed once, and (b) most packers unpack the original program to an area of memory which has not been previously executed. Given this, the technique relies upon creating a histogram of the addresses of executed instructions (EIP on x86) in the order of its last execution time. Decryption, decompression and copying appear as large spikes at the start of the histogram, followed by a flat section, of height one, which is usually the execution around OEP. The pattern, dubbed 'Hump-and-Dump', of the graph helps reveal when unpacking activity has finished and therefore determines the address at which to dump the memory for further analysis.

The thesis further demonstrates that the 'Hump-and-Dump' technique combined with the section information can precisely locate the exact OEP.

### 8.1.4 Automatic Anti-anti-VMware Technique

VMware is widely used by antivirus researchers for malware analysis in an isolated environment. However, a large amount of the current generation of malware employs various anti-VMware techniques in order to resist analysis. If malware detects the presence of VMware, it will act differently.

This dissertation categorizes the existing anti-VMware techniques and develops an anti-anti-VMware technique to automatically counter two popular anti-VMware detections. In this approach, the memory is scanned for anti-VMware signatures at three important stages of the execution, i.e., at the start of the execution, after jumping to a new section and after handling the exception. If anti-VMware signatures are found, a hardware breakpoint is set on the main instruction. Once this breakpoint is triggered during the analysis, the execution result will be automatically patched to a pre-defined value which belongs to the native machine. By intercepting the calls which malware make to detect whether they are running in a virtual machine, and altering the result, the malware is fooled into thinking it is running on hardware.

Experimental results on one of the most sophisticated packers, namely Themida, and real packed malware, show that this anti-anti-VMware technique is able to find and disable VMware detections, even in multi-packed files. Moreover, this 'dynamic scanning and patching' technique is scalable to resist some of other anti-analysis tricks as well.

## 8.2 Future Work

Following the very encouraging results described here, there are several extensions which can build on this work. We present a number of ideas as future research directions in malware/packer analysis.

### 8.2.1 Development of A Robust and Scalable System

Successful AV researchers have been kept confounded by counter-counter attacks by malware developers. New anti-analysis techniques continually evolve and are implemented quickly by malware writers in a range of packers and malware. This makes it hard for AV researchers to carry out daily analysis work. Moreover, these tricks also impede the effective and smooth operation of an automatic analysis system.

Fortunately, our 'dynamic scanning and patching' technique presented in Chapter 7 can be scaled to resist certain tricks. For example, some packers detect the presence of two popular monitors, Filemon and Regmon, and pursue some remedy if they are found. They are called anti-filemon and anti-regmon tricks. Similar to anti-anti-VMware, a scalable malware/packer analysis system can be upgraded to dynamically scan and patch for these two tricks.

Meanwhile, future work will focus on optimising the REFORM tool to resist various armoring techniques, such as various anti-unpacking tricks mentioned in Ferrie's paper [28, 29], applied by the packer and malware. A robust and scalable packer analysis system will be essential to keeping up with the accelerating growth in packed malware.

### 8.2.2 Investigating the Randomness Profile of Malware

Malware classification is another research topic in malware analysis [56, 61, 102, 107, 116]. Researchers have used some common features of malware, such as *trigram* [56, 116], function length [118], $n-$grams [61] and strings [102], to identify malware family. The randomness profile used for packer classification will be an interesting feature to explore for malware classification.

### 8.2.3 Identifying the Most Important Packer Features

It has been proved that the randomness profile is a strong packer feature for Packer Classification. However, it is still unknown whether there are a set of attributes more important than others in the extracted profile. We need make enhancements to the feature extraction algorithm to select most important features from the randomness profile.

Useful packer features other than the randomness profile, such as PE header information, string information, can also be explored and incorporated into the feature vector to improve the system's effectiveness.

Another direction is to develop new effective pruning methods. So far, the *Trunk* pruning, which retrieves low randomness values at two ends, achieves the best results. This is probably due to the fact that most existing packers store code at the beginning or the end of the file. This pattern is also displayed in the randomness scanning plot shown in Chapter 4. Once malware authors counter this approach and store code in other places, then our proposed *Ordered Smallest* pruning (see Section 5.2.2) might outperform *Trunk* pruning. New pruning methods that consider not only low randomness values but also certain sections can be developed. For example, we can only retain the lowest values in the code section.

### 8.2.4 Exploration of Different Statistical Classifiers

In this dissertation, we evaluated four statistical classifiers, namely Naive Bayes, Sequential Minimal Optimization, $k-$Nearest Neighbor and Best-first Tree. Several other classifiers can be added to this list, such as Random Forest, other Bayesian methods, Bagging, etc. Furthermore, we can apply various multi-classifier algorithms [17] or rank the output of different classifiers instead of choosing the best one among them.

### 8.2.5 Exploration of Unpacking Heuristics

Our proposed 'Hump and Dump' unpacking strategy identifies the OEP area in a packed file. For online computation, the two thresholds of the "hump" and "flat section", $T_{\text{hump}}$ and $T_{\text{flat}}$, need to be tuned with a greater variety of testing results. Due to fact that the hump size is related to the size of the original file (the size of the compressed/encrypted data), it maybe better to use adaptive thresholds, i.e., adaptively discover the best threshold for different files.

It is also promising to combine the 'Hump and Dump' unpacking strategy with other unpacking heuristics for better performance. Therefore, various stop conditions of emulation that described in Section 2.5.3.2 can be tested with the 'Hump and Dump' to seek the best unpacking performance.

# Bibliography

[1] David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, Jan 1991.

[2] Ai4r: Artificial intelligence for ruby. `http://ai4r.rubyforge.org/`.

[3] Ion Androutsopoulos, John Koutsias, Konstantinos V. Chandrinos, and Constantine D. Spyropoulos. An experimental comparison of naive bayesian and keyword-based anti-spam filtering with encrypted personal messages. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 160–167, 2000.

[4] Ion Androutsopoulos, Georgios Paliouras, Vangelis Karkaletsis, Georgios Sakkis, Constantine D. Spyropoulos, and Panagiotis Stamatopoulos. Learning to filter spam e-mail: A comparison of a naive bayesian and a memory-based approach. In *Proceedings of Workshop on Machine Learning and Textual Information Access, 4th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, pages 1–13, 2000.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauery, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, Oct 2003.

[6] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, Aug 2006.

[7] Fabrice Bellard. QEMU. `http://www.nongnu.org/qemu/`.

[8] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2002.

[9] Lutz Böhne. Pandora's bochs: Automatic unpacking of malware. Master's thesis, Laboratory for Dependable Distributed Systems, University of Mannheim, 2008.

[10] Tom Brosch and Maik Morgenstern. Runtime packers: The hidden problem? `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf`, 2006. Black Hat USA.

[11] Robert G. Brown. Dieharder. `http://stat.fsu.edu/pub/diehard/`, 2004.

[12] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998.

[13] Pedro Bustamante. Mal(ware)formation statistics. `http://research.pandasecurity.com/malwareformation-statistics/`, 2007.

[14] Matthew Carpenter, Tom Liston, and Ed Sloudis. Hiding virtualization from attackers and malware. *IEEE Security & Privacy*, 5(3):62–65, 2007.

[15] Ero Carrera. pefile. `http://code.google.com/p/pefile/`.

[16] Ero Carrera. Packer tracing. `http://blog.dkbza.org/2006/06/packer-tracing.html`, 2006.

[17] Yu Y. Chou and Linda G. Shapiro. A hierarchical multiple classifier learning algorithm. In *Proceedings of 15th International Conference on Pattern Recognition (ICPR'00)*, volume 2, pages 2152–2155, 2000.

[18] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. *Malware Normalization*. University of Wisconsin, Madison, Nov 2005.

[19] William W. Cohen. Learning rules that classify e-mail. In *Proceedings of the AAAI Spring Symposium on Machine Learning in Information Access*, pages 18–25, 1996.

[20] Gregory Conti, Erik Dean, Matthew Sinda, and Benjamin Sangster. Visual reverse engineering of binary and data files. *LNCS: Visualization for Computer Security*, 5210/2008:1–17, 2008.

[21] Compuware Corporation. SoftICE. `http://www.compuware.com/pressroom/news/2002/1342_ENG_HTML.htm`.

[22] Microsoft Corporation. WinDBG. `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`.

[23] Microsoft Corporation. Windows Virtual PC. `http://www.microsoft.com/windows/virtual-pc/default.aspx`.

[24] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory, 2nd Edition*. John Wiley & Sons, 2006.

[25] Paul Craig. Unpacking malware, trojans, and worms: Pe packers used in malicious software. `http://www.security-assessment.com/files/presentations/Ruxcon_2006_-_Unpacking_Virus,_Trojans_and_Worms.pdf`, 2006. Ruxcon.

[26] Tim Ebringer. Anti-emulation through time-lock puzzles. `http://www.datasecurity-event.com/uploads/timelock.pdf`, May 2008. 2nd International CARO Workshop.

[27] Peter Ferrie. Attacks on virtual machine emulators. In *AVAR 2006*, Auckland, NewZealand, Sep 2006.

[28] Peter Ferrie. Anti-unpacker tricks 2 part one. *Virus Bulletin*, pages 4–8, December 2008.

[29] Peter Ferrie. Anti-unpacker tricks current. `http://www.datasecurity-event.com/uploads/unpackers.pdf`, May 2008. 2nd International CARO Workshop.

[30] Peter Ferrie. Anti-unpacker tricks 2 part four. *Virus Bulletin*, pages 4–7, March 2009.

[31] Peter Ferrie. Anti-unpacker tricks 2 part three. *Virus Bulletin*, pages 4–9, February 2009.

[32] Peter Ferrie. Anti-unpacker tricks 2 part two. *Virus Bulletin*, pages 4–9, January 2009.

[33] Amir Fouda. Malware on a mission. In *Proceedings of 19th Virus Bulletin International Conference*, Sep 2009.

[34] William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, Englewood Cliffs, N.J, 1992.

[35] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot Topics in Operating Systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[36] Anant P. Godbole and Stavros G. Papastavridis. *Runs and Patterns in Probability: Selected Papers (Mathematics and Its Applications)*. Kluwer Academic Publishers, 1994.

[37] Lawrence A. Gordon, Martin P. Loeb, William Lucyshyn, and Robert Richardson. CSI/FBI computer crime and security survey, 2006. `http://i.cmpnet.com/gocsi/db_area/pdfs/fbi/FBI2006.pdf`.

[38] Aleksandr Gostev. Kaspersky security bulletin statistics 2008. http://www.viruslist.com/en/downloads/vlpdfs/kaspersky_security_bulletin_part_2_statistics_en.pdf, 2009.

[39] Aleksandr Gostev. Kaspersky security bulletin statistics 2009. http://www.viruslist.com/en/analysis?pubid=204792098, 2010.

[40] Tobias Graf. Generic unpacking - how to handle modified or unknown PE compression engines. In *Proceedings of 15th Virus Bulletin International Conference*, Oct 2005.

[41] Ilfak Guilfanov. Fast library identification and recognition technology. http://www.datarescue.com/idabase/flirt.htm.

[42] Ilfak Guilfanov. Parameter identificaiton and tracking technology. http://www.datarescue.com/idabase/pix/ida_pit_white.gif.

[43] Ilfak Guilfanov and Yury Haron. Universal unpacker. http://www.datarescue.com/idabase/unpack_pe/unpacking.pdf, 2005.

[44] Yoann Guillot and Alexandre Gazet. Semi-automatic binary protection tampering. *Journal in Computer Virology*, 5(2):119–149, May 2009.

[45] Fanglu Guo, Peter Ferrie, and Tzi cker Chiueh. A study of the packer problem and its solutions. In *Proceedings of 11th International Symposium On Recent Advances in Intrusion Detection (RAID)*, pages 98–115, Sep 2007.

[46] Peter Hellekalek. plab. http://random.mat.sbg.ac.at/tests/.

[47] Hex-Rays. The IDA Pro disassembler and debugger. http://www.hex-rays.com/idapro/.

[48] David A. Huffman. A method for the construction of minimum-redundancy codes. In *Institute of Radio Engineers*, pages 1098–1101, 1952.

[49] VMware Inc. VMware workstation. http://www.vmware.com/products/ws/.

[50] The WildList Organization International. WildList. http://www.wildlist.org/.

[51] Jibz, Qwerton, Snaker, and XineohP. PEiD. http://www.peid.info/.

[52] Oliver Johnson. *Information Theory and the Central Limit Theorem*. Imperial College Press, 2004.

[53] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of 5th ACM Workshop on Recurring Malcode (WORM 2007)*, pages 46–53, Nov 2007.

[54] Ken Kato. VM back. http://chitchat.at.infoseek.co.jp/vmware/backdoor.html.

[55] Maurice G. Kendall and Bernard Babington Smith. Randomness and random sampling numbers. *Journal of the Royal Statistical Society*, pages 147–16, 1938.

[56] Jeffrey O. Kephart, Gregory B. Sorkin, William C. Arnold, David M. Chess, Gerald J. Tesauro, and Steve R. White. Biologically inspired defenses against computer viruses. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 985–996, 1995.

[57] Tobias Klein. Jerry - a(nother) VMware fingerprint suite. http://www.trapkit.de/research/vmm/jerry/index.html, 2003.

[58] Tobias Klein. Scoopy Doo - VMware fingerprint suite. http://www.trapkit.de/research/vmm/scoopydoo/index.html, 2003.

[59] Donald E. Knuth. *The Art of Computer Programming, 3rd Edition*. Addison-Wesley Professional, 2004.

[60] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, pages 1137–1145, 1995.

[61] Jeremy Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2699–2720, 2006.

[62] Kostya Kortchinsky. Honeypots: Counter measures to VMware fingerprinting. `http://seclists.org/honeypots/2004/q1/0015.html`, 2004.

[63] Elias Aka Lallous. Detect if your program is running inside a virtual machine. `http://www.codeproject.com/system/VmDetect.asp`, Apr 2005.

[64] Boris Lau. DSD-Tracer implementation and experimentation. In *Proceedings of 17th Virus Bulletin International Conference*, pages 18–26, Sep 2007.

[65] Ming Li and Paul Vitanyi. *An introduction to Kolmogorov Complexity and its Applications: Preface to the First Edition*. Springer-Verlag, 1997.

[66] Tom Liston and Ed Sloudis. On the cutting edge: thwarting virtual machine detection. `handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf`, 2006.

[67] Guillaume Lovet. Dirty money on the wires: The business models of cyber criminals. In *Virus Bulletin 2006*, Montréal, Canada, Oct 2006.

[68] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, Mar-Apr 2007.

[69] George Marsaglia. The mardaglia random number CDROM including the diehard battery of tests of randomness. `http://stat.fsu.edu/pub/diehard/`, 1995.

[70] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: Fast, generic, and safe unpacking of malware. In *Proceedings of 23rd Annual Computer Security Applications Conference (ACSAC)*, Dec 2007.

[71] Joren McReynolds. Packer detection and generic unpacking techniques. http://securitylabs.websense.com/content/Blogs/2927.aspx, Feb 2008.

[72] Darek Mihocka and Stanislav Shwartsman. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure, June 2008. 1st Workshop on Architectural and Microarchitectural Support for Binary Translation.

[73] Machine learnig open source software. http://mloss.org/about/.

[74] Alistair Moffat and Andrew Turpin. *Compression and Coding Algorithm*. Kluwer Academic Publishers, 2002.

[75] Maik Morgenstern and Andreas Marx. Runtime packer testing experiences. www.datasecurity-event.com/uploads/runtimepacker.ppt, May 2008. 2nd International CARO Workshop.

[76] NIST. Nist statistical test suite. http://csrc.nist.gov/groups/ST/toolkit/rng/index.html.

[77] Hamish O'Dea. Trapping worms in a virtual net. In *Proceedings of 14th Virus Bulletin International Conference*, Fairmont Chicago, Illinois, USA., Sep 2004.

[78] Queensland University of Technology. Crypt-x. http://www.isi.qut.edu.au/resources/cryptx/.

[79] Alfredo Andres Omella. Methods for virtual machine detection. www.s21sec.com/descargas/vmware-eng.pdf, Jun 2006.

[80] Openrce. PE format. http://www.openrce.org/reference_library/files/reference/PE\%20Format.pdf.

[81] Athanasios Papoulis. *Probability and Statistics*. Prentice Hall, 1989.

[82] Tauhida Parveen, William H. Allen, Scott R. Tilley, Gerald A. Marin, and Richard Ford. Towards the detection of emulated environments via analysis of the stochastic nature of system calls. In *SEKE*, pages 802–807, 2008.

[83] Roberto Perdisci, Andres Lanzi, and Wenke Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters*, 29(14):1941–1946, 2008.

[84] Charles P. Pfleeger and Shari L. Pfleeger. *Security in Computing (3rd Edition)*. Prentice Hall PTR, 2002.

[85] Matt Pietrek. An in-depth look into the Win32 portable executable file format. `http://msdn.microsoft.com/msdnmag/issue/02/02/PE/print.asp`, 2002.

[86] PIN. `http://rogue.colorado.edu/Pin`.

[87] John C. Platt. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*. Microsoft Research, Apr 1998.

[88] PyDbg. `http://pedram.redhive.com/PyDbg/docs`.

[89] John Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[90] Danny Quist and Val Smith. Detecting the presence of virtual machines using the local data table. `http://www.offensivecomputing.net/dc14/vm.pdf`, 2005.

[91] Danny Quist and Val Smith. Further down the VM spiral. `http://www.offensivecomputing.net/dc14/furthur_down_the_vm_spiral.pdf`, Aug 2006.

[92] Danny Quist and Valsmith. Covert debugging: Circumventing software armoring. In *Black Hat Briefings and Training*, Las Vegas, USA, Aug 2007.

[93] Thomas W. Rauber. Tooldiag. `http://sites.google.com/site/tooldiag/`.

[94] Wolfram Research. Mathematica 7. `http://www.wolfram.com/products/mathematica/index.html`.

[95] Robert Richardson. CSI computer crime and security survey, 2008. `www.cse.msstate.edu/~cse6243/readings/CSIsurvey2008.pdf`.

[96] John Scott Robbin and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, Denver .CO, Aug 2000.

[97] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of 22rd Annual Computer Security Applications Conference (ACSAC)*, Dec 2006.

[98] Mark Russinovich. ProcDump v1.8. `http://technet.microsoft.com/en-us/sysinternals/dd996900.aspx`.

[99] Joanna Rutkowska. Red Pill...or how to detect VMM using one CPU instruction. `http://incisiblethings.org/papers/redpill.html`, 2004.

[100] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A bayesian approach to filtering junk e-mail. In *Papers from the AAAI Workshop, AAAI Technical Report WS-98-05*, pages 55–62, 1998.

[101] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Co., New York, 1983.

[102] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. Data mining methods for detection of new malicious executables. In *In Proceedings of the IEEE Symposium on Security and Privacy*, pages 38–49, 2001.

[103] Claude E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, pages 379–423, Oct 1948.

[104] Claude E. Shannon. Communication theory of secrecy system. *Bell Systems Technical Journal*, pages 656–715, Oct 1949.

[105] Haijian Shi. Best-first decision tree learning. Master's thesis, The University of Waikato, 2007.

[106] Alex Shipp. Unpacking strategies. In *Proceedings of 14th Virus Bulletin International Conference*, Sep 2004.

[107] Muazzam A. Siddiqui. Data mining methods for malware detection. Master's thesis, University of Central Florida, Orlando, 2008.

[108] VMProtect Software. VMProtect. `http://www.vmprotect.ru/`.

[109] Adrian E. Stepan. Defeating polymorphism: Beyond emulation. In *Proceedings of 15th Virus Bulletin International Conference*, Oct 2005.

[110] Joe Stewart. OllyBonE. `http://www.joestewart.org/ollybone/`.

[111] Li Sun, Tim Ebringer, and Serdar Boztaş. Hump and dump: Efficient generic unpacking using an ordered address execution histogram. `http://www.datasecurity-event.com/uploads/hump_dump.pdf`, May 2008. 2nd International CARO Workshop.

[112] Karl M. Syring. GNU utilities for Win32. `http://unxutils.sourceforge.net/`, 2004.

[113] Gabor Szappanos. Exepacker blacklisting. *Virus Bulletin*, pages 14–19, October 2007.

[114] Pang N. Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Pearson Education, Inc., 2006.

[115] Oreans Technologies. Themida. `http://www.oreans.com/ThemidaWhatsNew.php`.

[116] Gerald J. Tesauro, Jeffrey O. Kephart, and Gregory B. Sorkin. Neural networks for computer virus recognition. *IEEE Expert*, 11(4):5–6, 1996.

[117] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition (4th edition)*. Academic Press, 2008.

[118] Ronghua Tian, Lynn Margaret Batten, and Steve Versteeg. Function length as a tool for malware classification. In *3rd International Conference on Malicious and Unwanted Software (Malware)*, pages 79–86, Oct 2008.

[119] Christiaan van der Walt and Etienne Barnard. Data characteristics that determine classifier performance. In *Proceedings of 17th Annual Symposium of the Pattern Recognition Association of South Africa*, 2006.

[120] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, 1979.

[121] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques (2rd Edition)*. Morgan Kaufmann, San Francisco, 2005.

[122] Yun F. Wu and Jun H. Chen. Syser kernel debugger. `http://www.sysersoft.com/`.

[123] Oleh Yuschuk. OllyDBG. `http://www.ollydbg.de/`.

[124] Harry Zhang. The optimality of naive bayes. In *FLAIRS Conference*, 2004.

# Appendix A

# Glossary

## A.1 Glossary of AV Terms

**Assembly languages:** A family of low-level languages for programming computers, microprocessors, microcontrollers, and other (usually) integrated circuits. They implement a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture.

**Clean File:** A file that has been determined to be neither malicious, nor potentially unwanted.

**Cryptor:** A packer that may be used, legitimately, or illegitimately, to protect an application from being reverse-engineered, or otherwise analyzed. These tools use encryption in order to obfuscate the content of an application, often for the purposes of avoiding detection and hindering analysis.

**Instruction pointer (in 32-bit mode) (EIP):** The memory address which the processor will next attempt to execute.

**Emulation:** A method of creating a fake environment, and such techniques are deployed in various ways in computer security.

**Encryption:** Encryption is the method of transforming readable data into unreadable data for the purposes of secrecy. Once encrypted, such data cannot be interpreted (either by humans or machines) until it is decrypted. Encryption is performed using an encryption algorithm and a secret value called a ‘key’. Encrypted data generally cannot be decrypted without knowledge of the secret ‘key’ or substantial resources. Malware authors may use encryption in malware in order to obfuscate its code (make its code unreadable), thus hoping to hinder its detection and removal from the affected machine. A common and simple encryption technique used by malware is XORing, in which the Exclusive Or (XOR) computational operation is applied to each bit according to a given key. Malware authors may use cryptors in order to encrypt their code.

**Entry point (EP):** The address where the program starts execution.

**Exception:** Special condition that changes the normal flow of program execution. Exceptions are typically used to signal that something went wrong (e.g. a division by zero occurred or a required file was not found). Exceptions are raised or thrown (initiated) by either the hardware or the program itself by using a special command.

**Exception handling:** A programming language construct or computer hardware mechanism designed to handle the occurrence of exceptions. In general, an exception is handled (resolved) by saving the current state of execution in a predefined place and switching the execution to a specific subroutine known as an exception handler. Depending on the situation, the handler may later resume the execution at the original location using the saved information.

**Heuristics:** A tool or technique that enhances the ability to identify certain, and potentially common, code patterns. This is useful for making, for example, generic detections for a packer family.

**Image base:** The preferred address when the file is mapped into the memory.

**Import address table (IAT):** A table of external function pointers filled in by the windows loader as the DLLs are loaded.

**Interruption:** A signal through which a momentary pause in the activities of the microprocessor is brought about.

**In-the-wild:** Malware that is currently detected in active computers connected to the Internet, as compared to those confined to internal test networks, malware research laboratories, or malware sample lists.

**Little endian order:** Using this order, the least significant byte is stored at the first memory address allocated for the data. The remaining bytes are stored in the next consecutive memory positions.

**Malware:** Malicious software or potentially unwanted software installed without adequate user consent.

**Obfuscation:** A technique used in both malware, to disguise the content of a malicious program.

**Original entry point (OEP):** The EP of the original program that has been packed.

**Packer:** Packers are wrappers put around pieces of software to compress and/or encrypt their contents. They can be used by legitimate software to minimise download times and storage space or to protect copyrighted coding, but are commonly used in malware to disguise the contents of malicious files from malware scanners.

**Payload:** The malware's purpose other than propagation (in the case of viruses and worms); the actions conducted by a piece of malware for which it was created. This can include, but is not limited to, downloading files, changing system settings, displaying messages, logging keystrokes, and so on.

**Polymorphic:** A virus that can mutate its structure to avoid detection by anti-virus programs. It can mutate usually by changing a variable or variables in its code without changing its overall algorithm.

**Portable executable (PE):** A file format for executables, object code, and DLLs, used in 32-bit and 64-bit versions of Windows operating systems.

**Reverse engineering (RE):** A process of analysing a suspected malicious sample to discover what it does and what resources it uses. The process is a central part of malware analysis. Reverse engineering can be achieved through many processes of analysis, including observing the actions of a program as it runs, and using tools such as debuggers, disassemblers and decompilers to break down the actual code making up the file.

**Signature:** A set of packer characteristics that can be used to uniquely identify it using anti-virus products.

**Unpacking:** Removing packing from a file to see its true contents.

**Variant:** Individual piece of packer, sub-unit of a packer family. Most types of packers are subdivided into a number of families, or groups sharing many similarities, generally based on the same blocks of code and sharing similar behavior. Within a family, a variant signifies a single individual item that is uniquely different from other members of the same family.

**Virtual machine (VM):** A complete, fully functional operating system running inside some form of emulator. It can be used for a variety of purposes but are particularly useful for malware research where they act as complex sandboxes within which malware can be run safely and its behavior analysed.

**Virus:** A program that infects other files. The term 'computer virus' is often used as a general term for all types of malware, including Trojans and other non-replicating malicious code.

**WildList:** A monthly list compiled by the WildList Organization - a collaborative group of anti-malware experts - of the malware reported from the real world during the past month. As a reliable subset of current malware it is used as the basis for many product tests and certification systems.

## A.2 Glossary of Mathematics Terms

**Bernoulli random variable:** A random variable that takes on the value of one with probability $p$ and the value of zero with probability $1 - p$.

**Binary sequence:** A sequence of zeroes and ones.

**Binomial distribution:** A random variable is binomially distributed if there is an integer $n$ and a probability $p$ such that the random variable is the number of successes in $n$ Bernoulli experiments, where the probability of success in a single experiment is $p$. In a Bernoulli experiment, there are only two possible outcomes.

**Central limit theorem:** For a random sample of size n from a population with mean $\mu$ and variance $\sigma^2$, the distribution of the sample means is approximately normal with mean $\mu$ and variance $\sigma^2/n$ as the sample size increases.

**Class:** A collection of data samples defined by a human (designer or user) as sharing a specific property.

**Classifier:** An algorithm trained from a set of labeled data samples (each sample is associated with a class). When presented with an unseen example, the classifier returns a crisp decision (the estimated class). Internally, the classifiers are usually defined in two steps. The first is a statistical model producing a soft output (confidence that a sample belongs to a given class) and the second is a decision function converting the soft output into a crisp decision.

**Confusion matrix:** A matrix relating the class labels in a test set to the classifier decisions. It is usually a square matrix, with identical ordering of classes and decisions so that its diagonal represents correctly assigned samples and off-diagonal elements the errors.

**Cumulative distribution function (CDF) F(x):** A function giving the probability that the random variable $X$ is less than or equal to $x$, for every value $x$. That is, $F(x) = P(X \leq x)$.

**Discriminant:** A classifier separating two or more classes of objects. Discriminants split the feature space into open sub-spaces which means that any data sample arbitrarily far from the original training data (e.g. an outlier or example of a new class) is labeled into one of the trained classes. In order to reject examples unseen in training, a discriminant may be coupled with a detector.

**Entropy:** A measure of the disorder or randomness in a closed system. The entropy, or uncertainty, of a random variable $X$ with probabilities $p_1, \cdots, p_i, \cdots, p_n$ is defined to be $H(X) = -\sum_{i=1}^{n} p_i \ \log \ p_i$.

**Erfc:** The complementary error function $\boldsymbol{erfc}(z)$. It is related to the normal CDF.

**Feature:** A characteristic measured on all objects or phenomena to be processed by the pattern recognition system. Individual features are considered as dimensions of a feature space containing feature vectors (data samples). The statistical model one uses is crucially dependent on the choice of features.

**Histogram:** A histogram is a graphical display of tabulated frequencies, shown as bars. It shows what proportion of cases fall into each of several categories.

**Huffman encoding:** An entropy encoding algorithm used for lossless data compression. The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol.

**Level of significance ($\alpha$):** The probability of falsely rejecting the null hypothesis. It is the probability of concluding that a sequence is non-random when it is in fact random. Synonyms:Type I error, $\alpha$ error.

**Null hypothesis ($H_0$):** A statement about the assumed default condition/property of the observed sequence.

**P-value:** The probability (under the null hypothesis of randomness) that the tested sequence would be higher random than a perfect RNG produced sequence.

**Random binary sequence:** A sequence of bits for which the probability of each bit being a "0" or "1" is 1/2. The value of each bit is independent of any other bit in the sequence, i.e., each bit is unpredictable.

**Random number generator (RNG):** A mechanism that purports to generate truly random data.

**Run:** An uninterrupted sequence of like bits (i.e., either all zeroes or all ones).

**Statistically independent (events):** Two events are independent if the occurrence of one event does not affect the chances of the occurrence of the other event. The mathematical formulation of the independence of events A and B is that the probability of the occurrence of both A and B being equal to the product of the probabilities of A and B (i.e., $P(A \text{ and } B) = P(A)P(B)$), provided $P(A) \neq 0$ and $P(B) \neq 0$.

**Statistical test (of a hypothesis):** A function of the data (binary stream) which is computed and used to decide whether or not to reject the null hypothesis. A systematic statistical rule whose purpose is to generate a conclusion regarding whether the experimenter should accept or reject the null hypothesis $H_0$.

# Appendix B

# Abbreviations

## B.1 Abbreviations of AV Terms

**AV:** Anti-Virus

**DLL:** Dynamic-Link Library

**EIP:** Instruction Pointer (in 32-bit mode)

**EP:** Entry Point

**EPO:** Entry Point Obscuring

**GDTR:** Global Descriptor Table Register

**IAT:** Import Address Table

**IDTR:** Interrupt Descriptor Table Register

**LDTR:** Local Descriptor Table Register

**OEP:** Original Entry Point

**PE:** Portable Executable

**RE:** Reverse Engineering

**SEH:** Structured Exception Handling

**SGDT:** Store Global Descriptor Table

**SIDT:** Store Interrupt Descriptor Table

**SLDT:** Store Local Descriptor Table

**UPX:** the Ultimate Packer for eXecutables

**VM:** Virtual Machine

## B.2 Abbreviation of Mathematics Terms

**BFTree:** Best-First Decision Tree

**kNN:** k-Nearest Neighbor

**NB:** Naive Bayes

**RNG:** Random Number Generator

**SMO:** Sequential Minimal Optimization

**SVM:** Support Vector Machine