JAIST Repository
https://dspace.jaist.ac.jp/

Title	Original Entry Point detection of packed code based on graph similarity
Author(s)	Pham, Thanh Hung
Citation	
Issue Date	2023-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/18740
Rights	
Description	Supervisor: 小川 瑞史, 先端科学技術研究科, 修士(情報科学)

JAIST
JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Master's Thesis

Original Entry Point detection of packed code based on graph similarity

2010436 Pham Thanh Hung

Supervisor Prof. Mizuhito Ogawa
Examiners Associate Prof. Razvan Beuran
Prof. Minh Le Nguyen
Prof. Kazuhiro Ogata

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

September, 2023

Abstract

For many years, one of the great hazards to computers is malware which can cause a dramatic impact on their victim. Meanwhile, the number of new malware has increased exponentially each year, and most of them are obfuscated by packers for protecting malware from different anti-virus systems and many algorithms.

The Original Entry Point (OEP) of packed code is the place indicating the beginning point of the original payload during the execution of packed code. When OEP is detected, we can observe and investigate the original functionality. However, finding OEP automatically is still a challenging task because of the diversity of packers. In some previous works, OEP can be found in the specific region with some patterns (e.g., jump to the reserved region), but finding these patterns is still difficult for many packers. Therefore, a method to automatically detect OEP is essential and meaningful.

Based on our observation, most of packers will encrypt the original payload and combine it with an unpacking stub, which decrypts the packed payload during the execution of the packed code. After the last instruction in the unpacking stub is executed, the control flow will jump to the OEP. Hence, the unpacking stub can be regarded as a signature for each packer and we expect some pattern in each stub. To find such patterns, we observe CFG of various packed codes, which is generated by BE-PUM (Binary Emulation for PUshdown Model), a dynamic symbolic execution tool of Intel x86 binaries. We store the template of each unpacking stub by applying BE-PUM on example-packed codes. Since we know the original code, we can compare the original and the restored code by BE-PUM, and obtain the CFG of the unpacking stub as the difference.

This paper proposes a method for packer identification and OEP (Original Entry Point) detection based on the graph similarity on control flow graphs of packed codes. Packed code consists of an unpacking stub and a packed payload, which is recovered to the original after the unpacking stub execution. We start with the hypothesis that the CFG of the unpacking stub characterizes a packer. The CFGs of packed code are generated by a DSE (Dynamic Symbolic Execution) tool BE-PUM on x86-32/Windows, and when their original payloads and used packers are known, we can identify the unpacking stub in a packed code. First, for 771 samples packed by 12 packers from given payloads, we classify them by a clustering algorithm DBSCAN to confirm the hypothesis. We observe that when the allowance *eps* is enough small (e.g., *eps* = 0.02 in DBSCAN), (1) each class does not cross different packers (whereas some packer has multiple classes of CFGs of the unpacking stubs, e.g., WINUPACK has 2), and (2) the end instruction sequence (the prefix of the exit of an unpacking stub with the specified length) is the same in each class. Hence, we define the *template* of the class as the pair of the average of Weisfeiler-Lehman histogram vectors and the end sequence. Each template is computed packer-wise (i.e., clustering packed codes by the same packer) for the ease to cover a new packer. Next, for unknown packed code, BE-PUM incrementally generates the CFG. When the end sequence matches with the tail of a CFG fragment, we check the similarity

between its Weisfeiler-Lehman histogram vector and that in templates. Among them, the CFG fragment with the highest cosine similarity is regarded as the unpacking stub, which also detects the used packer and the OEP as the jump destination from the exit. Our experiment focuses on 12 packers, UPX, ASPACK, FSG, YODA'S Crypter, MEW, PACKMAN, PECOMPACT, PETITE, (WIN)UPACK, JDPACK, MPRESS, TELOCK, and the OEP of 688 among 700 non-malware packed samples (of which the original payload is also known) is correctly detected. Further, we apply the method to 1239 malware samples. Among them, 1089 samples are detected packed and 150 samples are packed by the 11 packers (except for TELOCK). We conclude that our method is highly effective as long as we have access to an executable of a target packer to compute its templates.

Keywords— Original Entry Point, x86 malware, packer, clustering algorithm, graph kernel, graph theory.

／# Acknowledgement

First and foremost, I would like to express my sincere gratitude and respect to my supervisor, Prof. Mizuhito Ogawa for his continuous support and guidance during my study at Japan Advanced Institute of Science and Technology. During this time, he has given me invaluable advice on how to think deeply and he has taught me how to write a technical document in a logical way and make it become a story. The process of undergoing training with him has helped me practice patience and meticulousness in my work. I believe that what I learned from him will help me a lot in the future.

I also appreciate Prof. Tu Minh Phuong and many teachers, and staff from PTIT because I have had the chance to study at JAIST because of their support.

I could not forget to mention the goodness of Mr. NGUYEN, Thai Son. He always told me that if I have any difficulty in my new life in Japan, I can tell him. He will support me as much as possible.

I would like to thank Giaohangtietkiem, particularly, Mr. Ma Trong Khoi, Mr. Dao Cong Anh, Mr. Pham The Hien, and my team Vision. When I expressed my decision to go to Japan, they supported me a lot. From my deep down, I am really grateful to them.

Thanks also go to Ms. KHONG, Phuong Thao from Le Quy Don University and Mr. NGUYEN, Thanh Tung from Giaohangtietkiem for their advice to me when I begin learning basic knowledge in the Security domain.

Especially, I would like to express my appreciation to my friends, Mr. TRAN, Cong Thanh, Mr. NGO, Tien Duc, Ms. NGUYEN, Thi Van Anh, Ms. NGUYEN, Thi Hai Yen, Ms. TRAN, Thi Anh Thu, Mr. NGUYEN, Vu Khac Hiep, and Mr. Pham Trung Tin. Thank you for your help not only in academic activities but also in daily life. I will remember those days when we studied together and ran together to keep our health and reduce my stress during my wonderful and tough time in JAIST.

Obviously, my big thanks go to my family, they always ask me about my health and my work during the year I am at JAIST. I always received their love and encouragement motivated me to study and keep moving forward.

Last but not least, my gratitude and love are to my wife NGUYEN, Thu Quynh. I cannot complete this work without her love, support, and faith.

Contents

List of Figures 4

List of Tables 0

1 Introduction 1

2 Preliminaries 6
2.1 Terminology and notation related to graph 6
2.2 Weisfeiler-Lehman Kernels 7
2.3 DBSCAN 10

3 X86 on Windows 14
3.1 X86 instruction set 14
3.2 Windows 17
3.3 API call in Windows 19

4 Malware and packer 23
4.1 Malware 23
4.2 Obfuscation techniques 24
4.3 Packer 27

5 Deobfuscation and BE-PUM 30
5.1 Deobfuscation and DSE 30
5.2 BE-PUM 32

6 The CFG of binary code 36
6.1 Problem Statement 36
6.2 The CFG of binary code 40
6.3 CFG of packed code 44

7 Control Flow Graph of unpacking stub 47
7.1 Reasons for avoiding retreating edges 47
7.2 The CFG of the unpacking stub characterizes a packer 49
7.3 Consistency of the end sequence 56

8 Packer identification and OEP detection **59**
8.1 Template matching for packer identification and OEP detection 59
8.2 Computing weisfeiler-lehman graph and its histogram vector 60
8.2.1 Computing weisfeiler-lehman graph 61
8.2.2 Computing weifeiler-lehman histogram vector 62
8.3 Template setup for each packer 66
8.3.1 Clustering procedure 66
8.3.2 Clustering Weisfeiler-Lehman histogram vector 68
8.3.3 Computing average Weisfeiler-Lehman histogram vector 68

9 Experiments **70**
9.1 Experimental environments 70
9.2 Packer identification 73
9.3 OEP detection 74
9.4 Packer identification and OEP detection on malware samples 77

10 Conclusion **78**
10.1 Discussion, Conclusion and Current limitation 78
10.1.1 Conclusion and current limitation 78
10.1.2 Discussion 79
10.2 Future works 81

List of Figures

2.1 WLK Iteration 7
2.2 WLK Subtree 10
2.3 DBSCAN 11
2.4 DBSCAN Illustration 12

3.1 Example of Conditional Jump 15
3.2 PE Structure 17
3.3 autologon in detectiteasy 19
3.4 User mode and kernel mode in Windows 20
3.5 An overview of OS service 20

4.1 Packing and Unpacking Process 28
4.2 Jump OEP in upx-autologon.exe in ollydbg 28

5.1 Symbolic Execution 31
5.2 On the fly manner 31
5.3 BE-PUM Architecture 33
5.4 Binary Emulator 34
5.5 GetDateFormat Flow Process 35

6.1 Packer Identification using Detect It Easy 37
6.2 AccessEnum non pack and packed 38
6.3 The last instruction 38
6.4 Unpacked AccessEnum.exe 38
6.5 Overview 39
6.6 a jump condition node in autologon.exe packed by UPX 41
6.7 CFG of high-level programming language 43
6.8 API name Capitalization 44
6.9 OEP node and end-of-unpacking node 46

7.1 CFG of Unpacking code and unpacked payload 48
7.2 Treating edges depends on DFS travel 49
7.3 CFG with Cycle 50
7.4 Reverse of a directed graph 52
7.5 Predecessor Subgraph Extraction 53

7.6 Reachable subgraph 54
7.7 End-of-unpacking graph 55
7.8 Different structure of end-of-unpacking graph in winupack 56

8.1 End-of-unpacking graph 59
8.2 Relabelling function 62
8.3 Frequency feature extraction 65
8.4 Standard end-of-unpacking clustering 67
8.5 DBSCAN clustering 68

9.1 OEP acquisition 71
9.2 GUI of GUnPacker 74

10.1 jump start in PACKMAN 80

List of Tables

7.1 End-of-unpacking sequence 57
7.2 The effect of eps on DBSCAN clustering 58

9.1 The number of packed codes for each packer 72
9.2 The results of the packer identification task 73
9.3 The results of the OEP detection task 75
9.4 The effect of end sequence in packer identification problem 76
9.5 The effect of end sequence in OEP detection problem 76
9.6 The number of malware samples can detected packer'name and OEP . . . 77

Chapter 1

Introduction

Motivation

Malware is software developed to do malicious actions on victim machines. To prevent malware, many antivirus software has been developed to protect the system. However, the problem of detecting and classifying malware has become harder because malware authors utilize packers to hide their malicious code. Actually, over 80% malware is obfuscated by packers for protecting it from anti-virus systems.

Packer is a software that often with obfuscation techniques packs the payload of software. The original aims of packers are to reduce the size of the files and evade reverse engineering to protect the licensed software from crackers. However, because of the ability to evade reverse engineering, malware authors have utilised packers to hide their malicious behaviors. The action of using packers to encrypt a binary is called the packing process, and the encrypted binary is called a packed code, which consists of an unpacking stub, this stub will decrypt the payload to the original file. After executing the unpacking stub, the control flow will be transferred to the original entry point (OEP) where the original program begins. At this moment, the functionality of the original program will be recovered totally.

To handle a packed code, it is necessary to perform the unpacking process to recover the original payload, which can be analysed for many different purposes. In the case of malware, the unpacking process can reveal their malicious intention and many analysis can be conducted further. Therefore, the problem of how to unpack a packed code has been received a lot of attention in both the industry and academia. Currently, there are many techniques for unpacking a packed code, but they can be classified into three major approaches which are manual unpacking, static unpacking, and generic unpacking for unpacking a packed file. Manual unpacking executes the packed program by using a native debugger (e.g., Ollydbg, SoftIce) to analyze the encryption of packer layer and decompression method, and manually recover the original files. Manual unpacking requires a huge effort and is time-consuming, while static unpacking is dedicated routines to decrypt executables packed by a specific packer without actually executing the malicious program, but this approach can be bypassed by unforeseen or custom packers. Generic

unpacking which uses programs to execute or emulate unknown packed executables until they are fully decrypted in memory becomes a potential approach. It can be defined as a method do not require specific knowledge about the packer and automatically perform unpacking. Nonetheless, regardless of the employed methodology, the detection of the OEP consistently remains a crucial necessity. However, finding OEP automatically is still a challenging task because of the diversity of packers. Therefore, a method to systematically detect the OEP of a packed code and easily extend it for different packers will be very useful since it saves a lot of human effort. Currently, our method follows the generic unpacking direction.

This paper proposes a method for packer identification and OEP (Original Entry Point) detection based on the graph similarity on control flow graphs of packed codes. Packed code consists of an unpacking stub and a packed payload, which is recovered to the original after the unpacking stub execution. We start with the hypothesis that the CFG of the unpacking stub characterizes a packer. Note that when their original payloads and used packers are known, we can identify the unpacking stub in a packed code. Fig 7.7 shows preliminary observation on FSG, MEW, UPX, and WINUPACK, respectively. They may have different patterns of CFGs of the unpacking stubs (for instance, WINUPACK has 2), but their similarity is high regardless of the payloads.

First, for 771 samples packed by 12 packers from given payloads, we classify them by a clustering algorithm DBSCAN to confirm the hypothesis. We observe that when the allowance *eps* is enough small (e.g., $eps = 0.02$ in DBSCAN), (1) each class does not cross different packers (whereas some packer has multiple classes of CFGs of the unpacking stubs, e.g., WINUPACK has 2), and (2) the end instruction sequence (the prefix of the exit of an unpacking stub with the specified length) is the same in each class. Hence, we define the *template* of the class as the pair of the average of Weisfeiler-Lehman histogram vectors and the end sequence. The length of a Weisfeiler-Lehman histogram vector rapidly increases if we consider a larger diameter of neighborhoods. For instance, if the number of instructions appearing in a CFG is m (often $m > 20$), the sum of 1-neighbors to k-neighbors (the vector length) grows $C_{m,k}$. Therefore, we choose $k = 2$.

Next, the template of each packer is prepared, which is computed packer-wise (i.e., clustering packed codes by the same packer) for the ease to cover a new packer (with relaxed $eps = 0.05$ for the optimal clustering when the used packer is known). We prepare the templates for 12 packers, UPX, ASPACK, FSG, YODA'S Crypter, MEW, PACKMAN, PECOMPACT, PETITE, (WIN)UPACK, JDPACK, MPRESS, and TELOCK.

Finally, for unknown packed code, BE-PUM incrementally generates the CFG. When the end sequence matches with the tail of a CFG fragment, we check the similarity between its Weisfeiler-Lehman histogram vector and that in prepared templates. Among them, the CFG fragment with the highest cosine similarity is regarded as the unpacking stub, which also detects the used packer and the OEP as the jump destination from the exit. To perform these two steps above, we need to prepare the template of the similarity and the end sequence of the instructions of the unpacking stub for each packer. Therefore, the limitation of our method will occur only with an unknown packer that we have no access to it.

Since the process of searching a graph is required in our method, we need an approach to perform the comparison between graphs. However, the direct comparison between graphs is not easy. Indeed, there are several approaches to measuring the similarity between graphs such as Graph Edit Distance (GED) and Graph Neural Networks (GNNs). However, the former can sometimes be working well in practice, but the problem of graph edit distance computation generally is NP-hard [1] and is even hard to approximate [2]. Meanwhile, the latter is a powerful approach with the advantage of deep learning, but this approach needs a huge dataset for training. Therefore, we apply a graph kernel method which transforms effectively graphs into feature vectors in a higher-dimensional space, making it easier for similarity comparison and applying machine learning algorithms. Among 700 non-malware packed dataset (of which the used packer and the original payload are labeled), our method correctly identifies the used packers for 689 samples, and the OEP for 688 samples. For the used packer, `VirusTotal` (which is the databese collected from various resources) identifies 699 samples beyond our result, but the OEP detection result is distinguished from others, e.g., `GunUnpacker` and `QuickUnpack` find 501 and 275, respectively. Further, we apply the method to 1239 malware samples. Among them, 1089 samples are detected packed and 150 samples are packed by the 11 packers (except for TELOCK).

Our main contributions consist of:

1. We proposed a generalized method to extract the templates of the unpacking stub for each packer by applying a clustering algorithm (as long as we have access to its excutable to pack).

2. We introduce the end instruction sequence of the unpacking stub. This sequence can be used as symbolic evidence indicating the exit of the unpacking stub.

We expect that precise OEP detection will enable us to understand techniques for infection and malicious actions, which will be in the original payload of malware.

Thesis structure

This thesis is composed of 10 chapters. Chapter 1 is the introduction, the next chapters are summarized as follows:

- **Chapter 2** presents some fundamental terminology and notation used in our work. In addition, weisfeiler-lehman graph kernels and a clustering algorithm are described as well.

- **Chapter 3** briefly provide some information of X86 on Windows. In addition, we explain how Windows handle API calls from a user process.

- **Chapter 4** firstly introduce malware and some malicious techniques. Second, we explain some typical obfuscation techniques. Third, we introduce packer which is a program often used to protect malware from anti-virus systems.

- **Chapter 5** presents Dynamic Symbolic Execution which is an approach to overcome obfuscation techniques. In addition, a tool called BE-PUM that can generate a precise control flow graph of a binary file is also introduced.
- **Chapter 6** presents the control flow graph of a binary, especially for a packed code.
- **Chapter 7** discusses several reasons why we need to remove retreating edges. This chapter also illustrates how an acyclic backbone is extracted from a graph. Next, we present the method used to generate a predecessor graph from an acyclic backbone. Finally, we introduce the sequence of the end of the unpacking stub.
- **Chapter 8** explains how a frequency vector is converted from a graph. On the other hand, the process of computing the average frequency vector for each packer is presented as well. Lastly, we will provide the utilization of average frequency vectors for both problems packer identification and OEP detection of a packed code.
- **Chapter 9** firstly describe our experimental environment. Next, we show the result of our experiment 700 non-malware samples of 12 packers and 1239 malware samples.
- **Chapter 10** discuss the failed case in our method and then summarizes the main contributions of the thesis and its current limitation. After that, some future works are also mentioned to suggest some directions to improve our method.

Related works on the OEP detection

Most of the OEP detection is based on a dynamic analysis. If fully executed, the original payload must be somewhere on the memory image. It would be too large to explore and mostly Instead of the full execution, some tries to interrupt around the detected OEP. For instance, OllyBonE[1] and Renovo [3] try to stop when the control jumps to an allocated region, where they expect that the decrypted payload is stored. OllyBonE includes a Windows kernel driver for the page protection of a specified region. Its OEP detection starts to choose the memory area and set the exception break-on-execute. It then waits for the unpacking stub to complete. Then, an exception occurs and the OEP is identified if the control flow moves to the address inside the protected area. This plugin, however, frequently fails when packers use anti-debugging techniques using API `IsDebuggerPresent@kernel32.dll`.

Meanwhile, Renovo is built on the top of an emulation environment, TEMU[2], and a shadow copy of the memory space of the targeted file is stored by Renovo. This method monitors runtime updates on memory. Then, the OEP is detected by extracting the recently generated code and data.

There are also some statistical analysis methods to detect OEP in [4], [5]. In the work [4], they will execute the packed code and keep running until meet one of the instructions

[1] http://www.joestewart.org/ollybone

[2] http://bitblaze.cs.berkeley.edu/temu.html

such as `JMP, JCC, CALL, or RET`. Next, entropy analysis will be conducted by measuring memory spaces. They assume that the end of the unpacking process happens when the entropy no longer changes. Meanwhile, the work [5] will collect OEP candidates where a page fault occurs, and then they will use two other OEP detection approaches to confirm whether a candidate is OEP or not. If both approaches decide a candidate is OEP, they conclude that OEP has been found. Here, the first approach is based on entropy, and the remainder is based on the number of API-call instructions present in the memory.

On the other hand, we can mention some works that detect OEP based on observing dynamic behavior, such as Polyunpack [6] and Omniunpack [7].

Another approach[8], [9] for the OEP detection prepares a candiate list of the OEP. `PinDemonium` [8] dumps the memory at an OEP candidate by using Scylla[10], and tries to reconstruct the library function table. If it works, the OEP is detected. However, the number of OEP candidates is often quite large. [9] tries to reduce the number of OEP candidates by observing `WrittenAndExecuted` addresses and the branch instructions. They are further refined by the command-line parameters of the main function.

The SE handler installation routine is then located by tracking the calls made by the system startup function. Particularly, the address of this routine is the last write instruction on fs:[0]. Finally, it considers the OEP as the WrittenAndExecuted address that is nearest to the SE handler.

However, most of the work focuses on monitoring the behaviour of packed code so it can be cheated by the use of obfuscation techniques, and sometimes we can just have a list of OEP candidates. Lastly, after we have the OEP, we still need to do more steps to retrieve the original program as an executable file.

In the work [11], the authors developed a hardware-assisted tool, API-Xray, to reconstruct import tables to achieve the ultimate goal of Windows malware unpacking. Their main method, API Micro Execution, investigates every possible API callsite and runs them without knowing the values of the API argument. In parallel, they use Intel Branch Trace Store and NX bit hardware tracing to determine API names and finally construct import tables. On the other hand, the work [12] introduces an all-in-one unpacking system that includes many different phases in Malware analysis such as detection, unpacking, and verification.

Regarding the graph similarity, [13], [14] apply for malware detection and malware analysis. After obtaining CFGs of packed codes by symbolic execution (BE-PUM and Angr, repectively), the former uses CNN on graphs and the latter uses (1-dimensional Weisfeiler-Lehman kernel, similar to us). However, they do not distinguish the unpacking stubs and the payloads, and the whole CFGs of packed codes are classified. Hence, they do not try on the packer identification and the OEP detection.

Chapter 2

Preliminaries

This chapter presents fundamental knowledge about the predecessor graph, Weisfeiler-Lehman kernel and the DBSCAN clustering algorithm. These notions and algorithms will be applied in this research.

2.1 Terminology and notation related to graph

We denote the contatenation of two vector v_1 and v_2 by $v_1 \oplus v_2$. For a finite directed graph $G = (V, E)$ with $E \subseteq V \times V$, let

$$\mathit{ancestors}(v) = \{u \mid (u, v) \in E\} \qquad \mathit{successors}(v) = \{u \mid (v, u) \in E\}.$$

The *indegree* $\deg^-(v)$ and *outdegree* $\deg^+(v)$ of $v \in V$ is $|\mathit{ancestors}(v)|$ and $|\mathit{successors}(v)|$, respectively. $v \in V$ is a *source node* (resp. *sink node*) if $\deg^-(v) = 0$ (resp. $\deg^+(v) = 0$). We also sometimes denote $u \to v$ if $(u, v) \in E$. A directed graph G is *acyclic* if there are no $v \in V$ with a cycle $v \to^+ v$.

We assume a DFS on G, which introduces the order among children nodes. We say,

- Foward edges: from ancestor to proper descendant.
- Cross edges: from right to left.
- Retreating edges: from descendant to ancestor (not necessarily proper).
- Back edges: Retreating edges with its tail dominate its head. Given two nodes u and v in a directed graph, node u dominates node v when every path from the root of the DFS tree to v has to contain node u.

Definition 2.1.1. Let $G = (V, E)$ be a directed acyclic graph. For $u \in V$, the *predecessor graph* from u is a graph $Pre_u^G = (V_u, E_u)$ with

$$V_u = \{v \in V \mid v \xrightarrow{*} u\} \qquad E_u = E \cap (V_u \times V_u) \tag{2.1}$$

If G is clear from the context, we may omit it as Pre_u.

The label of $G = (V, E)$ is given by a labeling function $l_G : V \to \Sigma$ where $\Sigma = \{\sigma_k \mid 1 \leq k \leq |\Sigma|\}$ is the set of labels. When $l_G(u) = \sigma$, σ is the label of u in G.

2.2 Weisfeiler-Lehman Kernels

The Weisfeiler-Lehman Test of Isomorphism

Graph isomorphism refers to the concept of determining whether the structures of two graphs are identical or "isomorphic". Alternatively, the target of this task is to determine whether one graph can transform into other graphs by relabeling their nodes.
In order to deal with this problem, The 1-dimensional Weisfeiler-Lehman test has been introduced. This algorithm performs with multiple iterations. The key strategy of the algorithm is to relabel nodes in two graphs by compressing the current node labels with the sorted set of node labels of neighbouring nodes. This process will be terminated with 2 conditions:

- The label sets of the two graphs are different.
- The number of interactions reaches a default value of h.

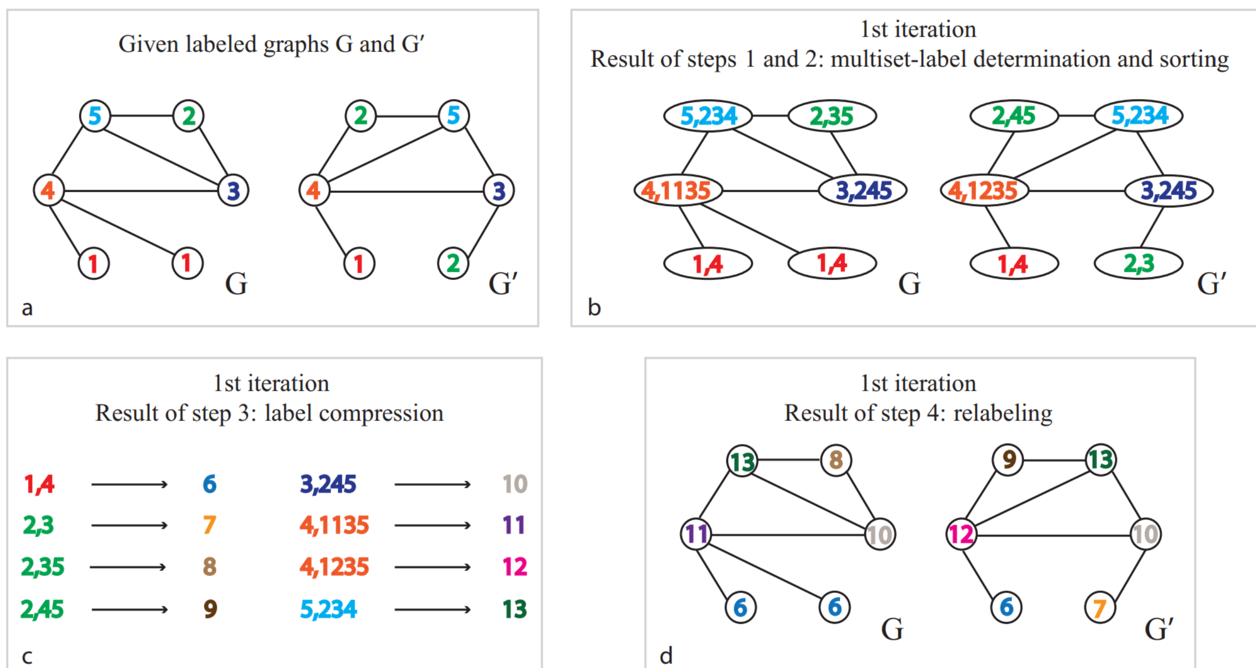

Figure 2.1: An example of one iteration of Weisfeiler-Lehman algorithm [15]

Figure 2.1 illustrates one iteration of the algorithm. For example, the node has label 5 in graph G has its neighbour are 2, 3 and 4. Therefore, in the first iterations, labels 2, 3, and 4 will be compressed to 234, and then 5 will be concatenated with 234 as a prefix to obtain the new label which is 5,234. This operation will be applied to other nodes in the same way. Next, the new labels will be compressed into a number. This step can be considered as a function f. Here is the way f work. First, there will be a counter variable

for f that records the number of unique strings before the current iteration. When a string has been compressed in previous iterations, f will map this string into a value in the counter. In contrast, if the string is new, and it has not been recorded before, the counter will be increased by 1, and f assigns this new value to the new string. In addition, the purpose of sorting the set of neighbours is to ensure all identical strings are mapped into the same number.

In this section, we refer definitions to [15].

Procedure 2.2.1. One iteration of the 1-dim. Weisfeiler-Lehman test of graph isomorphism: Given two graphs G and G', the label function $\ell : V \rightarrow \Sigma$ is a function that assigns labels from an alphabet Σ to nodes in the graph. l_i is the label function of the graph after iteration i of the relabelling procedure. This procedure has four steps:

- **Step 1**: Multiset-label determination:
  - $M_0(v) := l_0(v) = \ell(v)$.
  - For $i > 0$, each node v will be assigned by a multiset-label $M_i(v)$ in G and G'. $M_i(v) = \{l_{i-1}(u) \mid u \in N(v)\}$.
- **Step 2**: $M_i(v)$ sorting:
  - The set $M_i(v)$ will be arranged in ascending order and then they will be concatenated into a string $s_i(v)$.
  - $s_i(v) = l_{i-1}(v) + s_i(v)$.
- **Step 3**: Label compression
  - For all v from G and G', $s_i(v)$ sorted in ascending order.
  - Assign each string $s_i(v)$ to a new label using a function $f : \Sigma^* \rightarrow \Sigma$ such that $f(s_i(v)) = f(s_i(w))$ if and only if $s_i(v) = s_i(w)$.
- **Step 4**: Relabeling.
  - Compute $l_i(v)$ by assign $l_i(v)$ equal to $f(s_i(v))$ for all nodes in G and G'.

Note that, the above definition used the same node labelling functions $\ell, l_0, ..., l_h$ for both G and G'. Intuitively, the idea of the relabelling procedure is similar to the N-gram technique in NLP.

The Weisfeiler-Lehman Kernel

From the previous section, we have introduced an approach to checking the Graph isomorphism. However, when the structure of two graphs is not identical, how much they are similar. In order to answer this question, this section will introduce the kernel. Intuitively, kernels can be understood as functions measuring the similarity of pairs of graphs.

Definition 2.2.1. Define the Weisfeiler-Lehman graph at height i of the graph $G = (V, E, \ell) = (V, E, l_0)$ as the graph $G_i = (V, E, l_i)$. The sequence of Weisfeiler-Lehman graphs can be defined as:

$$\{G_0, G_1, \ldots, G_h\} = \{(V, E, l_0), (V, E, l_1), \ldots, (V, E, l_h)\}$$

Where $G_0 = G$ and $l_0 = \ell$ represent the Weisfeiler-Lehman sequence up to height h of G. So, we have G_0 as the original graph, and $G_1 = r(G_0)$ indicates that G_1 is the result of applying the Weisfeiler-Lehman algorithm on G_0. This is similar to other iterations.

Definition 2.2.2. Let k be any kernel for graphs, that we will call the base kernel. Then the WeisfeilerLehman kernel with h iterations with the base kernel k is defined as

$$k_{\mathrm{WL}}^{(h)}(G, G') = k(G_0, G'_0) + k(G_1, G'_1) + \ldots + k(G_h, G'_h),$$

Where:

- h is the number of Weisfeiler-Lehman iterations.
- $\{G_0, \ldots, G_h\}$ and $\{G'_0, \ldots, G'_h\}$ are the Weisfeiler-Lehman sequences of G and G' respectively.

Now, we will describe an instance of the Weisfeiler-Lehman Kernel which is the Weisfeiler-Lehman Subtree Kernel.

Definition 2.2.3. Let G and G' be graphs. Define $\Sigma_i \subseteq \Sigma$ as the set of letters that occur as node labels at least once in G or G' at the end of the i-th iteration of the Weisfeiler-Lehman algorithm. Let Σ_0 be the set of original node labels of G and G'. Assume all Σ_i are pairwise disjoint. Without loss of generality, assume that every $\Sigma_i = \{\sigma_{i,1}, \ldots, \sigma_{i,|\Sigma_i|}\}$ is ordered. Define a map $c_i : \{G, G'\} \times \Sigma_i \to \mathbb{N}$ such that $c_i(G, \sigma_{i,j})$ is the number of occurrences of the letter $\sigma_{i,j}$ in the graph G.

The Weisfeiler-Lehman subtree kernel on two graphs G and G' with h iterations is defined as:

$$k_{\mathrm{WLsubtree}}^{(h)}(G, G') = \left\langle \phi_{\mathrm{WLsubtree}}^{(h)}(G), \phi_{\mathrm{WLsubtree}}^{(h)}(G') \right\rangle,$$

Where,

$$\phi_{\mathrm{WLsubtree}}^{(h)}(G) = (c_0, (G, \sigma_{0,1}), \ldots, c_0, (G, \sigma_{0,|\Sigma_0|}), \ldots, c_h(G, \sigma_{h,1}), \ldots, c_h(G, \sigma_{h,|\Sigma_h|})),$$

And,

$$\phi_{\mathrm{WLsubtree}}^{(h)}(G') = (c_0(G', \sigma_{0,1}), \ldots, c_0(G', \sigma_{0,|\Sigma_0|}), \ldots, c_h(G', \sigma_{h,1}), \ldots, c_h(G', \sigma_{h,|\Sigma_h|})),$$

In short words, the Weisfeiler-Lehman subtree kernel counts common original and compressed labels in two graphs.

Example 2.2.1:
Let's consider the figure 2.2 to see how feature vectors of two graphs are generated.

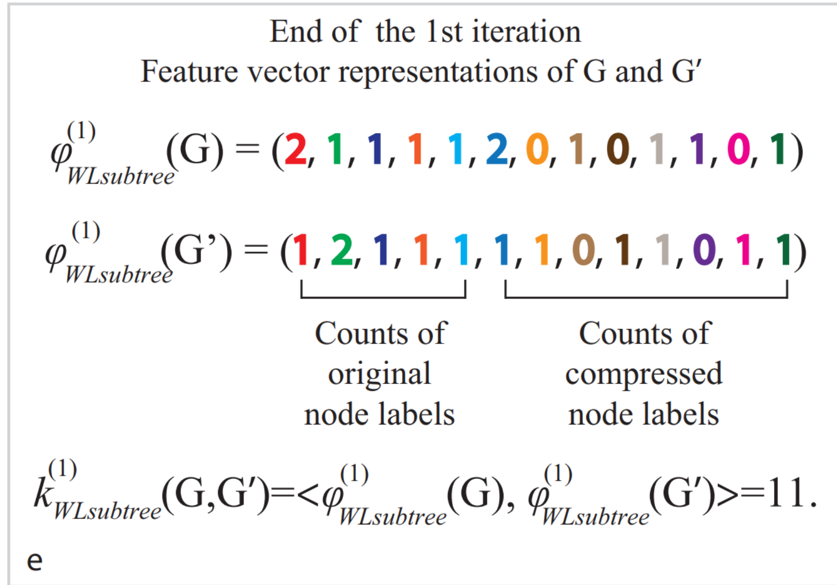

Figure 2.2: Feature vectors of two graph G and G' obtained by Weisfeiler-Lehman subtree kernel [15]

With the Weisfeiler-Lehman algorithm, now we can convert a graph into a feature vector. This plays an important role in this research because we have to deal with many graphs and compare their similarity.

2.3 DBSCAN

Machine learning

Machine learning (a branch of Artificial Intelligence) are algorithms that enable machines to learn from data. Next, it can predict new unseen data in the future. Machine learning algorithms can be categorized into three kinds of algorithm:

1. Supervised learning: an algorithm that predicts the output (outcome) of new data (new input) based on previously known (input, outcome) pairs. This data pair is also known as (data, label), i.e. (data, label). Supervised learning is the most popular group of Machine Learning algorithms.

2. Unsupervised learning: Unsupervised learning is a machine learning technique in which models are not supervised using a training dataset. In other words, our data don't have labels. An unsupervised learning algorithm will do some tasks from the

structure of the data. Problems in unsupervised learning can be divided into two groups: Clustering and Association.

3. Reinforcement learning: Algorithms that help a system automatically determine behavior based on circumstances to achieve the most benefit (maximizing the performance). Currently, Reinforcement learning is mainly applied to Game Theory, algorithms need to determine the next move to achieve the highest score.

In our work, we use a clustering algorithm belonging to an unsupervised learning algorithm to find the standard frequency vectors of each packer. The algorithm we used in this research is DBSCAN algorithm.

Density-based spatial clustering of applications with noise (DBSCAN)

DBSCAN is an unsupervised algorithm to handle clustering problems. This algorithm relies on a density-based notion of clusters which is designed to discover clusters of arbitrary shape. The key idea is that for each point of a cluster, there are a minimum number of points within a given radius, i.e. the density in the neighbourhood has to exceed some threshold. For example, the Figure 2.4 gives us some intuition about the cluster and noice.

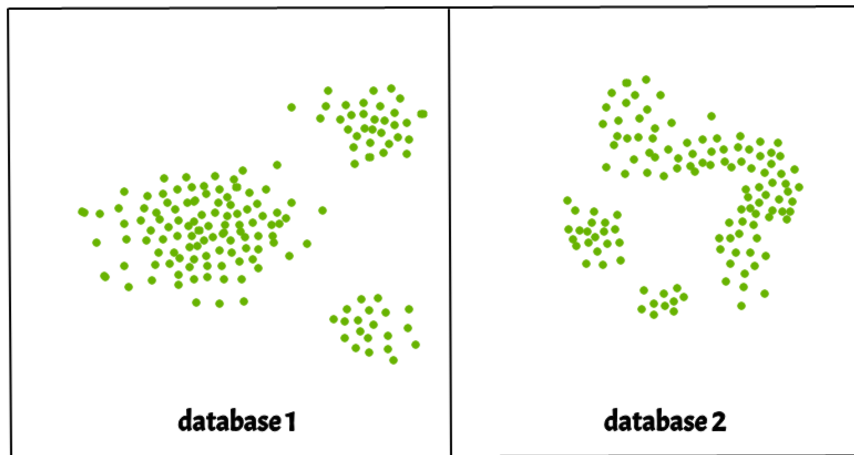

Figure 2.3: An sample set of points [16]

Clusters in the figure above are easily detected by the sense of a typical density of points which is considerably higher than outside of the cluster. In contrast, the density within the noise area is really low.

Before we go further about how DBSCAN work, there is some definition of the type of data points in this algorithm.

Definition 2.3.1. Let's consider a set of data points in some space to be clustered. We assume that ϵ is the radius of a neighbourhood with respect to some point. The data points in the DBSCAN algorithm can be defined below:

- A point p is considered as a core point if there are a minimum $minPts$ points within distance ϵ of the point (including p).
- A point q is said that it can be directly reached from p when point q is within distance ϵ from core point p. Note that, the word "reachable" is only used when the points are reachable from core points.
- If there is a path $p_1, ..., p_n$ with $p_1 = p$ and $p_n = q$, and each $p_i + 1$ is directly reachable from p_i, then a point q is reachable from p. With the possible exception of q, this implies that the starting point and every other point along the path must be core points.
- When we can find a point o and both p and q are reachable from o, two points p and q are density-connected.
- Outliers or noise points are all points not reachable from any other point.

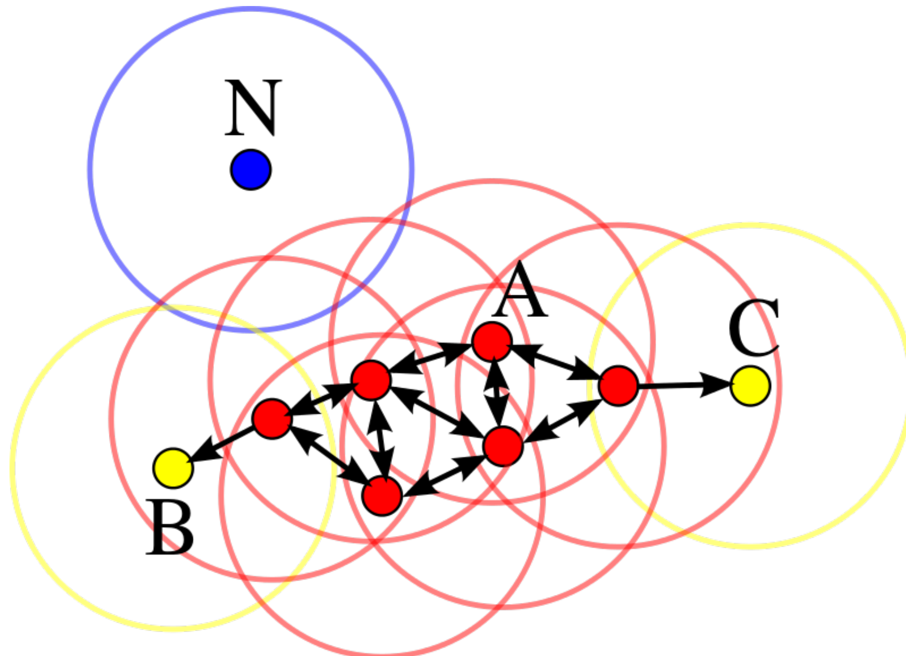

Figure 2.4: An sample of different kind of points [17]

In the figure above, the value of minPts is 4. The red points illustrate the core points because the number of points in their neighbourhood within the ϵ radius is no less than 4. In addition, these core points also form a cluster when they directly are reachable to each other. Meanwhile, B and C are not core points but they are reachable from A. As

a result, B and C also belong in the cluster. Finally, N is a noise because it's not a core point and it cannot be reached from other points.

Definition 2.3.2. DBSCAN algorithm. This clustering algorithm contains 4 steps:

- **Step 1**: Searching for core points by counting all the neighbour points within eps or visited with more than MinPts neighbours.
- **Step 2**: Creating a new cluster when a core point is not assigned to any cluster.
- **Step 3**: Recursively, travelling to all its density-connected points and merging them to the same cluster as the core point.
- **Step 4**: Travelling through the rest of unvisited points in the dataset, and identifying a noise if a point is not assigned to any cluster.

There are three main reasons we choose to use DBSCAN in this work.

1. DBSCAN does not require defining the number of clusters in the data. This is an important point in our research because there are no reasons to know a packer have a fixed number of graph structure in the CFG of unpacking stub.
2. DBSCAN can detect noise and this algorithm is robust with outliers.
3. The number of parameter of DBSCAN is small (2 parameters), these parameters is mostly insensitive to the ordering of the points in the database.

Chapter 3

X86 on Windows

This chapter presents some fundamental knowledge about X86 architecture on Windows and introduces how Windows OS handle API calls.

3.1 X86 instruction set

Because x86 is a popular architecture in computers, there are many malware developed to work in this architecture. However, there are also many anti-virus programs supported in this structure to prevent malware. To overcome many methods to detect malware in computers, malware in x86 has utilised many obfuscation techniques on it to hide its behaviour. Therefore, the research related to malware in x86 is really important.

Now, we need to go through some basic concepts of computers that are necessary to analyze malware in x86.

Processor

CPU stands for Central Processing Unit is an electronic circuit that executes computer program instructions by performing mathematical, logical, and comparative calculations. and basic input/output operations from predefined code in a computer. Nowadays, CPUs can support a much bigger memory space, and this allows them to store billions of values. As a result, processors can perform more complex operations. In addition, CPUs'power also increased by registers that are fast and small memory storage units inside them. Furthermore, processors can support many instruction types that can make processors change the control flow of execution based on certain conditions.

Register

Although processors can access a huge memory space that is provided by RAM devices, the processing time to access data in RAM is quite long. To speed up this process, processors are armed with small and fast internal memory storage units called registers. They can store the immediate values that are needed when processors perform calculations

and data transfer.
Register have a vary names, sizes and functions depending on the architecture. However, there are still several widely used types.

- **General-purpose register**: To temporarily store arguments and results for multiple arithmetics, and data transfer operations, the use of these registers is needed.
- **Stack and frame pointers**: They point to the top and particular fixed points of the stack.
- **Instruction pointer**: This register is used to point to the next instruction that will be executed by processors.

Memory

CPUs can process more information and perform more complicated operations such as displaying graphical interfaces in 3D and virtual reality thanks to the ability of memory when it can rapidly manage lots of values, text, images and video.

Instructions

Instructions are machine codes that CPUs can understand and execute. These machine codes are represented in the form of bytes.
Overall, Instructions, regardless of the architecture, can be divided into certain groups.

- **Data manipulation**: This include arithmetic and bitwise operations.
- **Data transfer**: These instructions in this group can move data involving register, memory and immediate values.
- **Control flow**: This kind of instruction can change the order execution of other instructions.

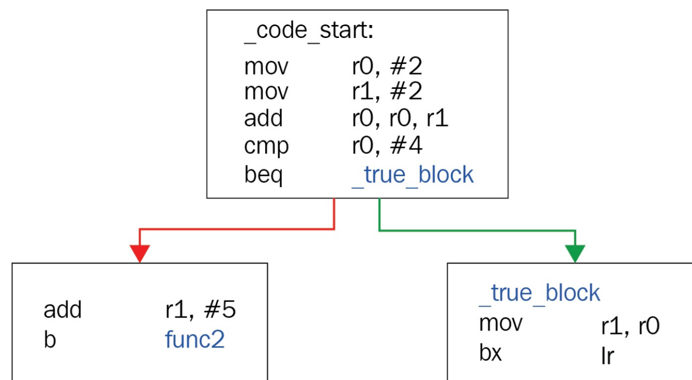

Figure 3.1: An example of conditional jump [18]

Now we will move on to know about x86 and malware.

x86 architecture

Intel 86 is the most popular architecture used in computers. This is the reason why most malware samples support this architecture. In this section, we will focus on the instructions of this architecture that we see many times during the process of analyzing an x86 file.
The common structure of instruction in x86 is *opcode, dest, src*. In this structure, *dest, src* can be referred to as **operands**, and the number of operands in instruction can be 0, 1, 2, and 3 depending on what kind of instruction. Particularly, the detail about opcode, dest and src will be described below:

- **opcode**: The name of the instruction that indicates what operation was performed. Some instructions such as nop, pushed, popa do not have operands.
- **dest**: The destination, or the place where the result of the operation will be saved. For example, *add eax, acx* instruction mean *eax = (eax + ecx).* Dest can be a register or a place in memory.
- **src**: The source or another value in the calculations. It could be a register, a place in memory or an immediate value, but it will not be used to save the results.

To understand more about x86, we will dive deeper into various instruction sets in this architecture.

- **Data manipulation instructions**: This set contains the most common arithmetic instructions such as *add/sub, inc/dec, mul, dev*, or instructions represent logical/bitwise operations like *or, and, xor, not.* Finally, bitwise shifts and rotations (*shl/shr, rol/ror*) are also included.
- **Data transfer instructions**: As the name of this instruction set, the instructions in this group can move the data. For example, *mov* instruction can copy a value from *src* to *dest*. For other instructions related to stack, we can mention *push/pop, pushad/popad, etc*. Lastly, string manipulation instructions are *lodsb/lodsw/lodsd/lodsq*. These instructions load a byte, 2 bytes, 4 bytes, or 8 bytes from *esi* address into *eax*, and there are many other instructions in this group.
- **Control flow instructions**: These kinds of instructions can change the value of *eip* register, so the next executed instruction may not be the next ones sequentially. Some common instructions can be mentioned like *jmp, call, ret/retn.* These instructions do not need a condition to execute. On the other hand, there are instructions such as *jnz/jz/ja/jb, loop* that need a condition for their execution. In this case, some form of comparison like *cmp, test* needs to be used.

In the next section, the PE Header structure will be introduced. Gaining an understanding of this structure and the information it contains can provide many benefits during the analysis of an x86 file.

3.2 Windows

Virtual memory

In modern OSs, they can create an isolated virtual memory for each process. In addition, applications are only designed to access their virtual memory. Here, they can read and write code and data and execute instructions. When an application wants to access a value stored in memory, it needs its virtual address.

PE structure

The PE header is a standard structure that all executable window files have to comply with this structure. It contains much valuable information such as the supported system, various metadata, and the memory layouts of sections containing code and data. These pieces of information help the system load and execute a file correctly.

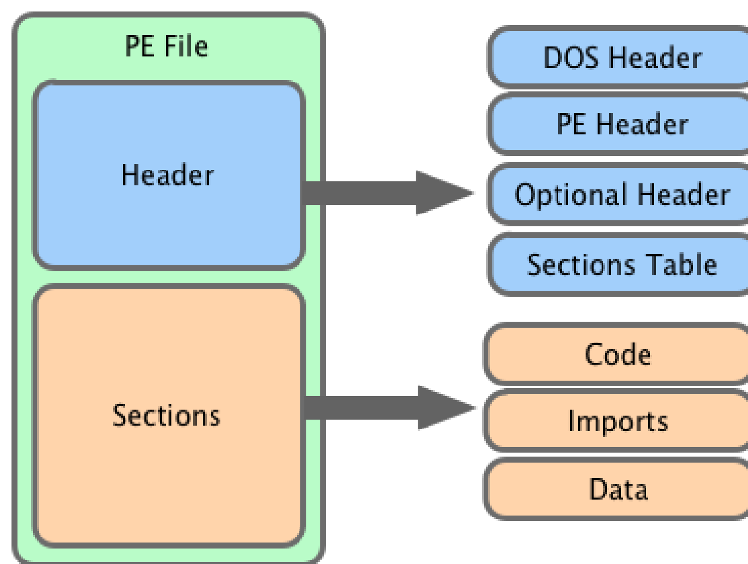

Figure 3.2: Example of PE structure [19]

PE Header

Some of the important values provided by the PE header are:

1. Machine: processor type is represented in this field.
2. NumberOfSections: The number of sections that come after the header such as the code section, data section, or resource section.
3. TimeDateStamp: The exact date and time that this program was compiled.
4. Characteristics: The type of executable file and specifies whether it is a program or a dynamic link library.

Optinal Header

Similar to the previous sections, here are some valuable information in this header:

1. Magic: The identity of the platform supported.
2. AddressOfEntryPoint: A very important field for our analysis. This value is the starting point of program execution (i.e., the first instruction will be executed in the program) relative to its starting address (its base).
3. ImageBase: The address where the program was designed to be loaded into virtual memory.
4. SectionAlignment: The size of each section and all header sizes should be aligned to this value when loaded into memory.
5. MajorSubsystemVersion: The minimum Windows version to run the application on such as Windows XP or Windows 7.
6. SizeOfImage: The size of the whole application in memory.
7. SizeOfHeaders: The size of all headers.
8. Subsystem: This field shows that this file could be a Windows UI application, a console application, or a driver, or that it could even run on other Windows subsystem.

The entry point of a normal program

Definition 3.2.1. The entry point of a program is the place where the execution of a program begins. This point is represented as a pair of $\langle address, instruction \rangle$.

Note that, the entry point of a program can be obtained from the header of the program. We can use many tools such as PEiD, detectiteasy and the VirusTotal website to obtain this point.

Example 3.2.1:
Let us consider the case of Autologon.exe.
The figure below shows the entry point of Autologon.exe read by detectiteasy tool in the green box is *00403980*. In addition, the assembly code of Autologon.exe also can see in this figure. The first address and instruction *00403980: call 0x408490* that appeared in the yellow box indicated that this is the entry point of Autologon.exe.

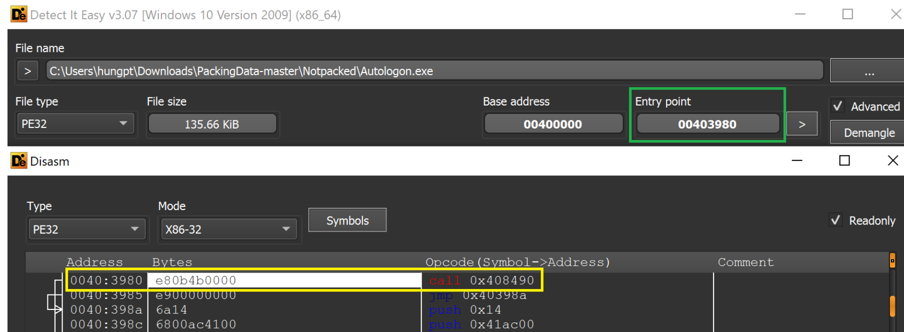

Figure 3.3: Autologon.exe disassembled by detectiteasy

3.3 API call in Windows

User mode and kernel mode

Because the Windows operation system shares the software resources of the computer system and the hardware with its users, the operation system needs to ensure that the incorrect software or malicious software can have a negative effect on other programs and the operation system itself. To achieve this target, the execution of the operating-system and user-defined codes must be separated. A processor in a computer running Windows has two different modes: user mode and kernel mode. The processor switches between the two modes depending on what type of code is running on the processor. Applications run in user mode, and core operating system components run in kernel mode. While many drivers run in kernel mode, some drivers may run in user mode. Windows starts a process for the user-mode application when you launch it. The procedure gives the application a private handle table and virtual address space. Data belonging to another program cannot be changed since each application has its own private virtual address space. Each application runs independently, and if one crashes, the crash only affects that particular application. The operating system and other apps are unaffected by the incident. A user-mode application's virtual address space is constrained in addition to being private. Virtual addresses that are set aside for the operating system cannot be accessed by a process running in user mode. By restricting a user-mode application's virtual address area, the application is prevented from changing and potentially harming. In contrast to user mode, A single virtual address space is shared by all code run in kernel mode. A kernel-mode driver isn't separate from other drivers or the operating system as a whole because of this. Data belonging to the operating system or another driver may be compromised if a kernel-mode driver unintentionally writes to the incorrect virtual address. The entire operating system fails if a kernel-mode driver malfunctions.

This diagram illustrates communication between user-mode and kernel-mode components.

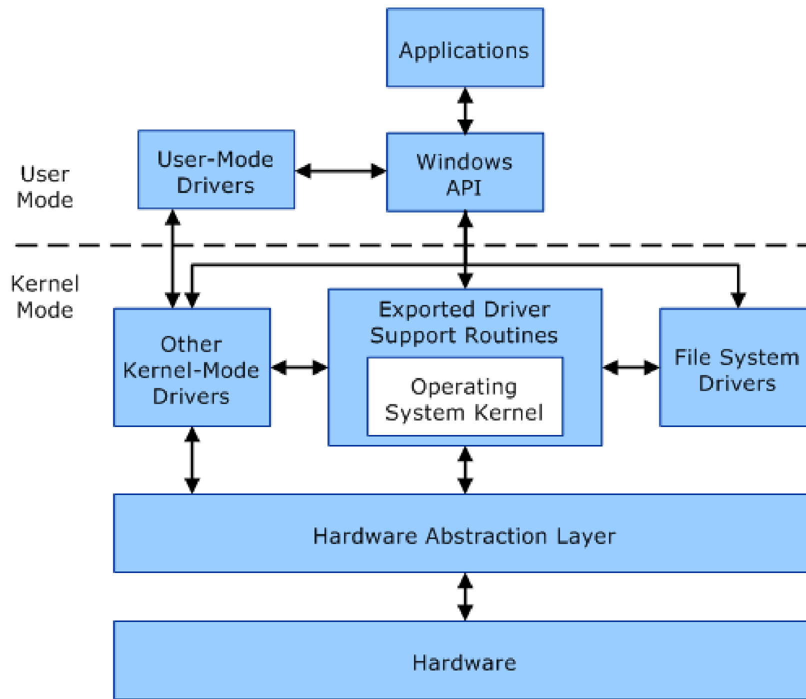

Figure 3.4: User mode and kernel mode in Windows [20]

Operating-System Services

Operating systems provide the environment for the execution of a program, and they also supply many specific services to programs and their users. Although these services can vary from one operating system to another operating systems, there are still many common classes between them.

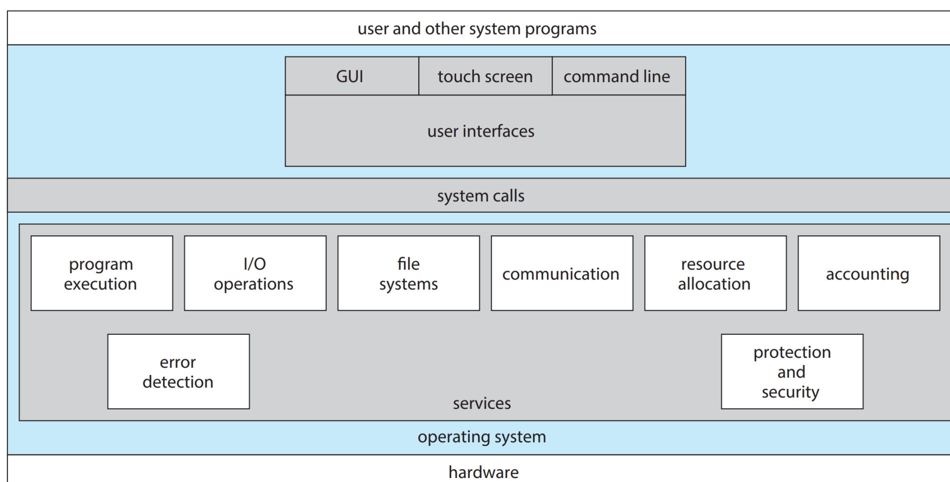

Figure 3.5: An overview of OS service[21]

Figure 3.5 show an overview of the operating system service. The functions provided by

these services are helpful to the users. From the figure, functions from operating system service can be classified into several major groups which are program execution, I/O operations, file systems, communication, resource allocation, accounting, error detection, and protection and security.

System call

System calls provide an interface to the services made available by an operating system. Generally, these calls are implemented as C/C++ functions. However, assembly language instructions also can be used to implement specific low-level tasks where hardware must be accessed directly. In fact, one simple program can make heavy use of the operating system. For example, to copy a file in Windows to another location, there are many functions needed by the operating system. User interfaces are required to help users can choose a file from the screen of Windows to copy, or reading the input file and write the output file related to File-system manipulation. Fortunately, most programmers do not need to know detail to this level. Instead, application developers build their applications based on an application programming interface (API). Particularly, a set of functions will be specified by API including the parameters and the returned values for each function. In the Windows system, the set of these APIs is called Windows API, and the functions that create an API basically invoke the actual system calls on behalf of the developer. For example, the Windows function `CreateProcess()` that is used to create a new process actually invokes the system call `NTCreateProcess()` in the Windows kernel.

In Windows, DLLs are used the most frequently by Windows applications, making them the most obvious and significant use. With the use of DLL:

- At build time, library functions are not connected. Instead, they are linked either at runtime (explicit linking) or at the moment the application loads. As a result, the program image can be considerably smaller because the library routines are not included.
- Shared libraries can be made with DLLs. Only one copy of a DLL library, which is shared by several apps, is loaded into memory. Although each process has a unique copy of the global variables in the DLL, all programs map the DLL code to their process address space.
- All programs that use the library can use the new version unaltered by just providing a new version of the DLL to support new versions or alternative implementations.
- The library will run the same processes as the calling program.

The entire Windows API is supported by a DLL that invokes the Windows kernel for additional services. Windows processes can share DLL code, but the code, when called, runs as part of the calling process and thread. The library will therefore be able to utilize the resources of the calling process, such as file handles, as well as the stack of the calling thread. Therefore, thread-safe DLLs should be created.

In Windows, system calls can be classified into 6 major categories. Examples for each category will be listed below:

1. Process control: CreateProcess(), ExitProcess(), and WaitForSingleObject().
2. File management: CreateFile(), ReadFile(), WriteFile(), and CloseHandle().
3. Device management: SetConsoleMode(), ReadConsole(), and WriteConsole().
4. Information maintenance: GetCurrentProcessID(), SetTimer(), and Sleep().
5. Communications: CreatePipe(), CreateFileMapping(), and MapViewOfFile().
6. Protection: SetFileSecurity(), InitlializeSecurityDescriptor(), SetSecurityDescriptorGroup().

Chapter 4

Malware and packer

This chapter presents fundamental knowledge about malware and packer.

4.1 Malware

Malware is software trying to thief information from computer users or having malicious actions on their machines. These actions can result in negative effects on victims. In general, there are three steps applied in malware techniques:

1. Obfuscation: The aim of this step is to complicate the control flow to avoid the detection of signature-based methods. Another purpose is to hide malicious behaviors in virtual environment emulation.

2. Infection: Malware attaches computers through Windows security holes.

3. Malicious actions: perform malicious behaviors such as taking control illegally, destroying data, and information theft.

Here, Taking control illegally refers to a situation where an attacker gains unauthorized access to a personal computer or network using malware. Once the attacker gains control, they can perform various illegal actions, such as:

- Theft of sensitive information: Malware can be designed to steal personal data, financial information, login credentials, or intellectual property.

- Botnet creation: Malware can turn infected computers into part of a botnet, a network that is remote-controlled by a command server. Whoever controls the botnet can make those zombie computers do bad stuff.

- Distributed Denial of Service (DDoS) attacks: Malware-infected systems can be used to flood websites or networks with traffic, causing them to become unavailable to legitimate users.

- Crypto hacking: Malware can hack a victim's computer to mine cryptocurrencies without their consent, utilizing the computer's resources and slowing down its performance.

Meanwhile, destroying data refers to the malicious activity carried out by certain types of malware that are specifically designed to delete, corrupt, or render data inaccessible to a target system. This destructive intent sets these types of malware apart from other categories that may focus on stealing data, spying on users, or using the infected system for other nefarious purposes. several common ways of destroying data can be listed below:

- Data Deletion: Malware simply delete files or directories including critical system files, user documents, applications, and more on the infected system.
- File Corruption: Malware can alter the content of files, making them unusable or causing errors when attempting to access them.
- Disk Wiping: making the data irrecoverable by wiping the entire contents of a hard drive or storage media. Certain types of malware, such as "wiper" or "data erasing" malware, are designed to perform this action.
- Overwriting: As the name of this malicious action, random characters can be overwritten on data. As a result, it effectively destroys the original content.
- Encryption and Ransomware: This kind of malicious technique doesn't typically destroy data. Instead of that, it encrypts files, making them inaccessible until the victim satisfied the requirement of malware authors for the decryption key. In this case, if the victim doesn't pay, the data effectively remains locked and may be considered lost.
- Master Boot Record (MBR) attacks: Malware can attack the MBR of a computer, rendering it unable to boot up, effectively causing data loss.
- Bricking: Bricking refers to the practice or act of rendering an electronic computing device — often a smartphone becomes useless or inoperable.

So, obfuscation is the first step for each malware technique. Therefore, the ability to deal with obfuscation techniques has an important impact on malware analysis.

4.2 Obfuscation techniques

The difficulty of analyzing packed malware comes from the use of the obfuscation technique by the packer. When a program is obfuscated, its assembly code will not be obtained precisely in a disassembler. Therefore, these packed codes can evade firewall and antivirus (AV) scanners. Currently, we concentrate on 14 typical obfuscation techniques, and these techniques can be classified into 6 groups:

1. **Entry/code placing obfuscation** (Code layout): overlapping functions, overlapping blocks, and code chunking.
2. **Self-modification code** (Dynamic code): overwriting and packing/unpacking.
3. **Instruction obfuscation**: Indirect jump.
4. **Anti-tracing**: SEH (structural exception handler) and 2API (the use of special APIs, LoadLibrary and GetProcAddress in kernel32.dll).
5. **Arithmetic operation**: Obfuscated constants and checksumming.
6. **Anti-tampering**: Timing check, anti-debugging, anti-rewriting, and hardware breakpoints. Anti-rewriting consists of stolen bytes and checksumming.

Entry/code placing obfuscation

Because the length of the x86 instruction sets can vary, the overlapping fragments of a binary sequence may happen in many ways:

- Overlapping instructions
- Overlapping function
- Code chunking

To clearly understand the concept of these techniques, let's consider the example below:

Example 4.2.1:

Let's assume we have a binary sequence `b8 eb 07 b9 eb` and `0f 90 eb`. These sequences can be read as `mov eax, ebb907eb` and `seto b1`. However, if we look at the fragment `eb 0f`, this fragment can be read as `jmp 45402c`. So, there is an overlap between instructions in this case. As a result, this case is an example of overlapping instruction.

Depending on different situations, we have overlapping functions and overlapping blocks. When the overlapping instructions happen among functions, we call it overlapping functions. On the other hand, the overlapping occurs between blocks, we have overlapping blocks. Especially, if we can divide a code into many fragments and these fragments are connected by jump instruction, we call this obfuscation technique code chunking.

Self-modification code

The obfuscation technique *overwriting* in this group tries to modify an instruction to another instruction. While the purpose of *packing/unpacking* techniques is to encrypt/decrypt a code.

Instruction obfuscation

We have *Indirect Jump* technique in this group. In the name of this technique, the word `jump` means a call, a jump, or a return, meanwhile *indirect* wants to express that the target destination is not easy to know. Indeed, these target in the directed jump is stored in a memory address, a register, or a stack frame. Furthermore, when the return destination stored in the stack is changed, this case is called an indirect return. For example, `54034FB CALL DWORD PTR DS:[ESI+503C]` illustrates an indirect call in UPX.

From the observation above, we can see that indirect jump is really a challenge when we want to analyze a binary code. Even if, we use symbolic execution which is a powerful technique to explore all possible paths of execution of a binary file.

Anti-tracing

- SEH: Structured Exception Handler is an exception handler written in a user process. The aim of this handler is to prepare for an exception and post-processes when these situations happen. However, this handler can be used with the *trap flag AL*. The trap flag AL can be changed to true and can cause a single-step exception, as in the code.

- 2API: This technique imply the use of special APIs, `LoadLibrary` and `GetProcAddress` in `kernel32.dll`.

Arithmetic operation

- Obfuscated Constants: replacing a constant with arithmetic operations. However, the result of these operations is the same original constant value.

- Checksumming: a typical of this technique is CRC checksumming.

Anti-tampering

- Anti-Debugging: As the name of this technique, it detects the presence of a debug mode by some instruction such as `CALL kernel32.IsDebuggerPresent`

- Stolen bytes: This technique will call `VirtualAlloc` and allocates a buffer, and then the unpacked code will be copied to this area.

- Timing Check: This technique is used to detect timing anomaly compared to the native Windows environment.

- Hardware breakpoint: Jump destination will be stored in debug registers such as `DR0, DR1, DR2, and DR3`.

Some obfuscation techniques in this group such as Anti-debugging and timing checks can make Dynamic analysis approaches using a debugger like OllyDbg failed to detect

malware. Especially, malware based on trigger-based behavior and VM-awareness. For example, malware can detect a virtual environment, and then it will do nothing. Another example is malware has trigger-based behavior, this malware only activates its malicious actions at a specific time such as April Fools' Day. Therefore, it has to be really lucky to detect this malware by dynamic analysis.

4.3 Packer

To evade firewall and antivirus scanners, malware authors utilize packers for protecting their malware from anti-virus systems. Originally, the first aim of packers is to reduce the size of binary files, and the second aim is to evade reverse engineering and protect the licensed software from crackers by encrypting a program into a packed code. However, the ability of packers now is also used to protect malware.

Definition 4.3.1. Packer is a program that employs obfuscation techniques to pack the payload of the original software A and transfers it into another packed code B. Packer can be defined as a function with domain X as a set of original software A and range Y as a set of packed code B.

$$Packer : X \rightarrow Y \quad \text{and} \quad Packer(A) = B \tag{4.1}$$

Definition 4.3.2. Packed code B can be defined as a tuple:

$$B = \langle H, U, P \rangle \tag{4.2}$$

Where:

- H is the portable executable header.
- U is the unpacking stub.
- P is the packed payload.

Definition 4.3.3. The Original Entry Point (OEP) is the place where the execution of the original payload begins during the execution of the packed code.

To understand more about OEP, let's consider Figure 4.1 which expresses both the packing process and the unpacking process.

- Packing process: a process that transforms an original program into a packed code.
- Unpacking process: a process that transforms a packed code into an unpacked program.

In the original program, the entry point will be pointed to the beginning of the original payload. When this file is packed by the packer, the original payload will be encrypted to the packed payload, and this packed payload is combined with an unpacking stub to create

a packed code whose entry point will be the beginning of the unpacking stub. In contrast, the unpacking process will start when the packed code is executed. In this process, the unpacking stub will be processed first. Then, this stub decrypts the packed payload to the unpacked payload and transfers the control to the starting point of this payload. The starting point of the unpacked payload is the original entry point. This explains why packed code can bypass statistic analysis or disassemble because these approaches can only see the unpacking code and the payload have been encrypted. As a result, the malicious behaviors of malware cannot be detected.

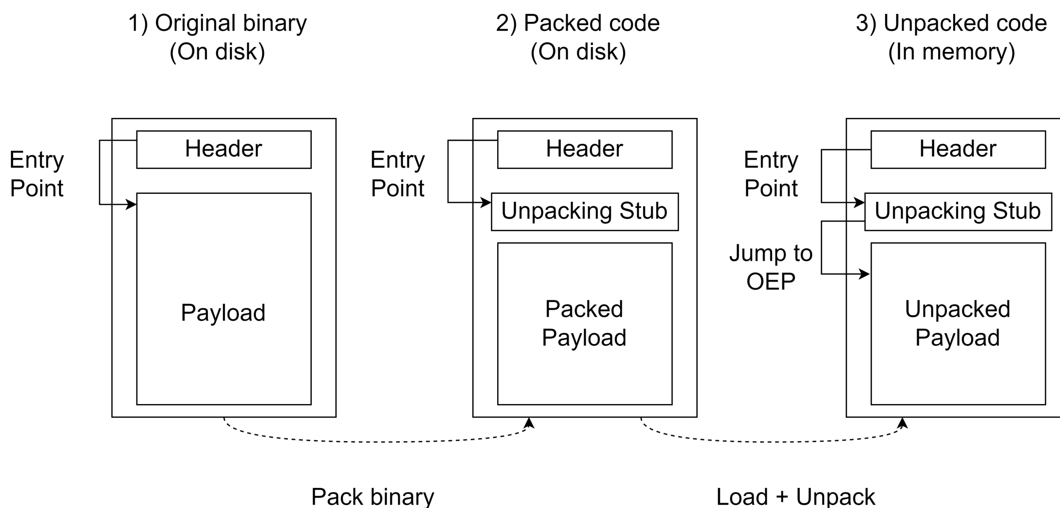

Figure 4.1: Packing and Unpacking Process

Example 4.3.1:

Let's consider Autologon.exe packed by UPX. For convenience, this packed code will be named `upx_Autologon.exe`. When we disassemble this file by Ollydbg, the entry point is ⟨`002F54B0, PUSHAS`⟩. However, when we continue to execute, we can reach a jump instruction ⟨`002F5664`, `JUMP upx_Auto.002D`**`3980`**⟩, This is the moment the control flow transferred to the original entry point.

```
002F5655  . 58            POP EAX
002F5656  . 61            POPAD
002F5657  . 8D4424 80     LEA EAX,DWORD PTR SS:[ESP-80]
002F565B  > 6A 00         PUSH 0
002F565D  . 39C4          CMP ESP,EAX
002F565F  .^75 FA         JNZ SHORT upx_Auto.002F565B
002F5661  . 83EC 80       SUB ESP,-80
002F5664  .-E9 17E3FDFF   JMP upx_Auto.002D3980
```

Figure 4.2: Jump to OEP in upx_Autologon.exe

It is clear that the development of unpacking techniques has become vitally important, and one of the problems that need to be solved during this development is OEP detection. In addition, the ability to deal with the obfuscation technique can have a significant impact on this problem. And, Dynamic symbolic execution has emerged as a good solution to deal

with obfuscation techniques. In addition, BE-PUM is a tool that applies this technique to generate a precise control flow graph. From that, it can handle the typical obfuscation techniques mentioned above.

Chapter 5

Deobfuscation and BE-PUM

5.1 Deobfuscation and DSE

In this section, obfuscation techniques can be handled by DSE below:

- Anto-debugging, Timeming Check handled by symbolic execution.
- Indirect jump handled by dynamic symbolic execution
- Self-modification code handled by the design of a node in the CFG generated by symbolic execution.

Symbolic Execution [22] is a commonly employed technique in software testing, where a program is executed symbolically instead of using specific input values. This approach allows for exploring all possible scenarios that may happen during execution. In this technique, symbolic values like α, and β will be the input. As a result, when a conditional branch appears, the constraints will be expressed by path condition *pc*. By utilizing this approach, all possible paths of a program can be explored by Symbolic Execution, and all possible outcomes can be tested.

The figure above describes how symbolic execution work to explore all possible scenarios of a program. From the top of the figure, we assume that the state of the program is n_0 with its path condition φ_0. In addition, the condition to make the program change from n_0 state to n_1 state is c_1. Therefore, when the state of the program is n_1, the corresponding path condition will be $\varphi_1 = \varphi_0 \wedge c_1$. In the same way, we have the path condition for the state n_2, n_3, and n_4. Then, an SMT Solver (e.g, $Z3$) will be used to check the satisfiability of these path conditions. With this property, symbolic execution can handle techniques such as Anti-Debugging, Timing Check because all possibilities of execution will be explored although the malicious behaviour is only performed with some conditions, especially for trigger-based behaviour e.g., attacks triggered by specific date and time and vm-aware in malware.

On the other hand, exploring all possible scenarios of a program is not easy with the use of *indirect jump technique*. To decide the next destination in Symbolic Execution, Static Symbolic Execution (SSE) and Dynamic Symbolic Execution (DSE) are two options for that purpose.

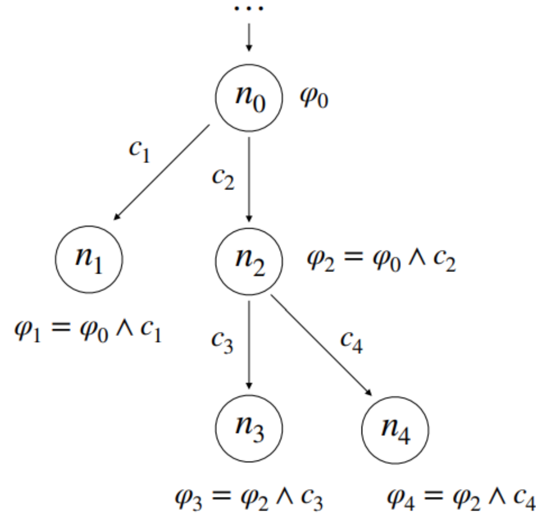

Figure 5.1: An illustration of symbolic execution [23]

- **Static symbolic execution.** The next destination candidates are statically detected. Then, each destination is checked by using an SMT solver to decide the feasibility of each path condition corresponding to these destinations. However, theorem provers cannot easily check the satisfiability of the candidates in some specific contexts such as complex constraints or indirect jumps in binary code.

- **Dynamic symbolic execution (or concolic testing).** To overcome these difficult situations mentioned above, static symbolic execution will combine with concrete execution, then dynamic symbolic execution be introduced. In DSE, the feasibility of the next destination will be checked by using a satisfiable instance from the precondition. This requires a binary emulator.

For example, we assume that there is a jump instruction *jump eax* at state n_2 in Figure 5.2, and the value of *eax* is expressed by a symbolic value. In this case, DSE will choose a concrete value satisfying path condition φ_2 as an instance of eax. Then, the CFG will be extended based on the determined target.

This way of extending CFG is named an on-the-fly manner.

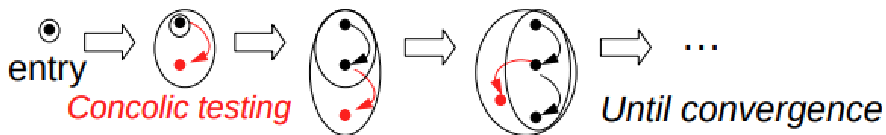

Figure 5.2: Control flow graph constructed by using DSE [24]

The figure above describes a control flow graph constructed in an on-the-fly manner. For each step, concolic testing will be used to extend the CFG while the state of the binary program and the environment of the binary emulator (flags, registers, stack, and memory) are also updated. This procedure will be finished when it faces an unsupported instruction or the end of the program is reached. In addition, to handle self-modification techniques, the node of a CFG will be represented by a pair of an address and its corresponding instruction.

To summarize, the obfuscation techniques used in trigger-based behavior, vm-aware, and dead-code insertion can be handled by symbolic execution. However, only the symbolic execution technique cannot overcome indirect jump, and Dynamic Symbolic Execution has emerged as an effective solution to handle this obfuscation technique.

5.2 BE-PUM

BE-PUM (Binary Emulation for PUshdown Model) [25] is a binary code analyzer concentrating on malware on Intel x86/Win32 architecture. The power of dealing with obfuscation techniques of BE-PUM is derived from the Dynamic Symbolic Execution technique (DSE) when this technique is used to explore the control flow graph of a binary file.

Architecture of BE-PUM

Overall, the architecture of BE-PUM can be illustrated by three main components which are a CFG storage, a binary emulator, and a symbolic execution (Figure 5.3). In addition, BE-PUM applies JackStab 0.8.3 [26] as the disassembler, and Z3 4.3 [27] as the theorem prover to perform test instances in the DSE process.

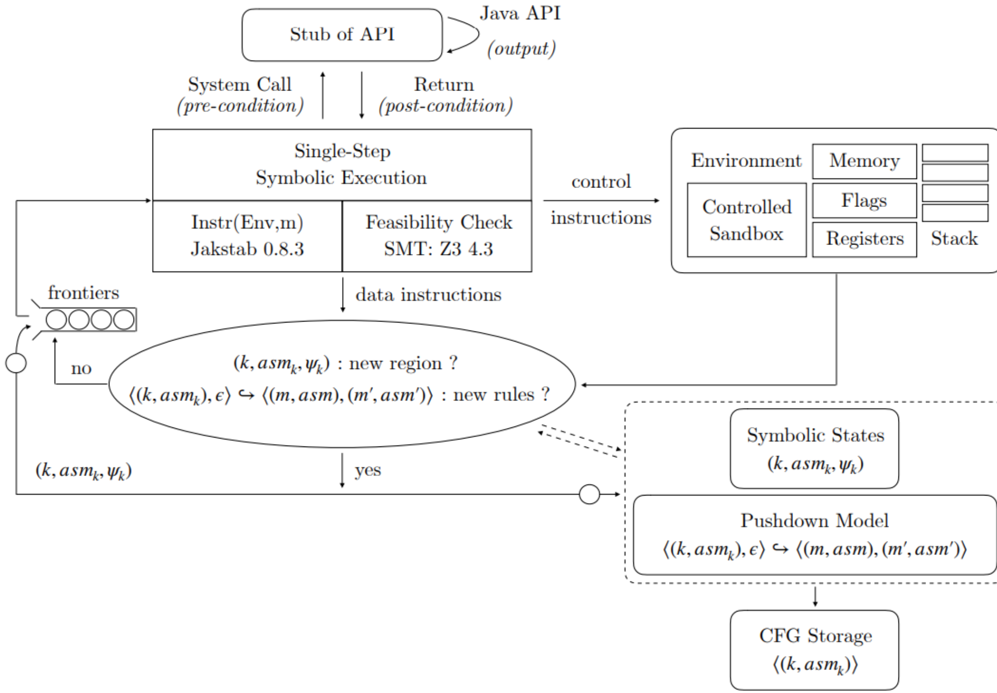

Figure 5.3: The architecture of BE-PUM [24]

First, a symbolic state at the end of an explored execution path will be selected from the frontiers as in the left-hand side of the figure. Next, BE-PUM tries to extend one step by Single-Step Symbolic Execution. At this step, If the instruction is a data instruction (i.e., only the environment is updated and the next location is statically decided), BE-PUM will disassemble the next instruction. In contrast, if the instruction is a control instruction (e.g., conditional instruction jumps) then the concolic testing is applied to decide the next location. In addition, the design of BE-PUM restricts the binary emulator that is required by concolic testing in a user process and handles APIs by stubs [25]. The advantage of this choice is giving flexibility in symbolic execution. Particularly, trigger-based behavior can be handled due to this choice. For example, the malware only executed on New Year's Day represents the property of trigger-based behavior. When we use OllDbg or Intel/Pin to analyze this malware, the malicious behavior may be failed to detect because these tools simulate the entire system. Therefore, if the system time is not New Year's Day, malware will not do anything. In contrast, BE-PUM can return the value of API in both symbolic value and concrete value. Therefore, the condition for exposing intended malicious activities on New Year's Day can be checked. After conducting concolic testing, a new CFG node or a new CFG edge will be created. In both cases, all of them are stored in CFG storage. In addition, a configuration is added to the frontiers. This procedure is terminated when either the exploration has converged or comes to unknown instructions, system calls or addresses.

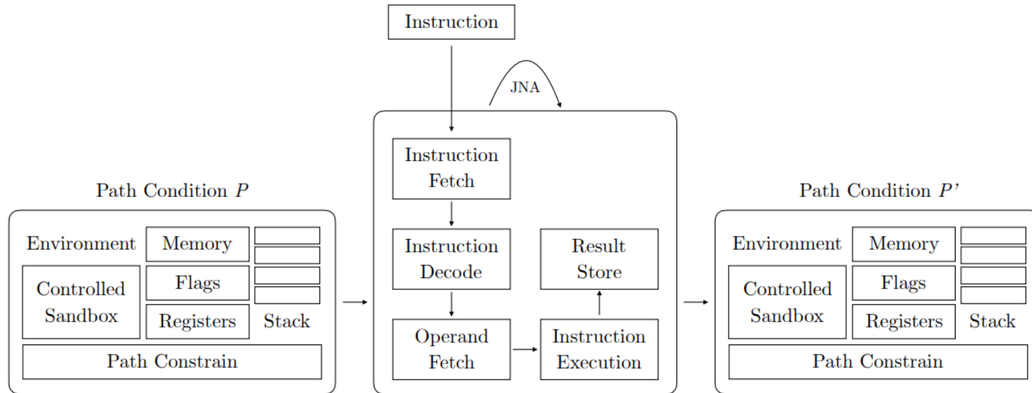

Figure 5.4: One-step concolic testing in BE-PUM [24]

On the other hand, Figure 5.4 shows the execution of a single step of concolic testing in BE-PUM. There are two kinds of functions of the binary emulator. The first one is to interpret an x86 instruction, and the second is to spawn a Windows API stub. Here, the Windows API sub is used to handle system API calls.

Windows API Stubs in BE-PUM

In BE-PUM, each Windows API call is considered a single instruction, and it updates the environment based on the technical document from MSDN.

With the use of Java Native Access (JNA)[1], Windows API stubs in BE-PUM are proxy objects capable of invoking native API functions and updating the simulation environment after the API call. It makes flexibility in symbolic execution and avoids the cost of manual APIs approximation. Currently, the work [28] has proposed a method to generate Windows API Stubs automatically for BE-PUM.

The flow process of API stub consists of 5 stages:

1. First, The number of values are popped from the stack of a simulation environment to the variables in the API Stub program based on the number of parameters.
2. Second, when a parameter is a pointer, the API stub program copies the referenced memory area's values to variables.
3. Third, JNA transfers the variables from the API stub to the native stack in the actual environment as input parameters before calling the native API function.
4. Next, JNA return to Java stack and the results are converted to proper variables in the API stub program.
5. Finally, based on the API definition, the API stub copies the value of variables to the appropriate EAX register and Memory objects in the simulation environment.

[1]https://github.com/java-native-access/jna

The five stages above can be illustrated in figure 5.5 below:

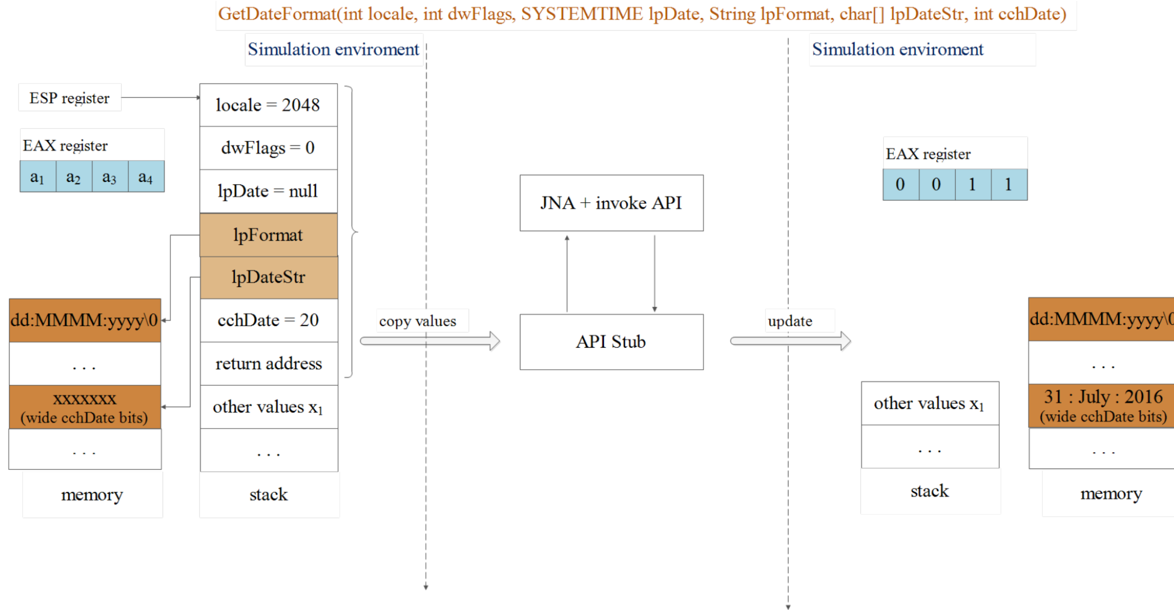

Figure 5.5: GetDateFormat Flow Process [28]

Chapter 6

The CFG of binary code

6.1 Problem Statement

In this work, OEP detection and Packer identification are two problems we focus to solve. These problems can be stated below.

Packer Identification Problem

Packer identification is a problem referred to the task of identifying the name of the packer that has been used to pack a binary file.

Input: Give a packed code that we do not know how this file is packed.
Output: The name of the packer used in the input file.

Example 6.1.1:
Figure 6.1 shows the result of packer identification of `AccessEnum.exe` packed by Packman packer. In the green rectangle in the figure, the tool has detected this file is packed by Packman version 1.0.

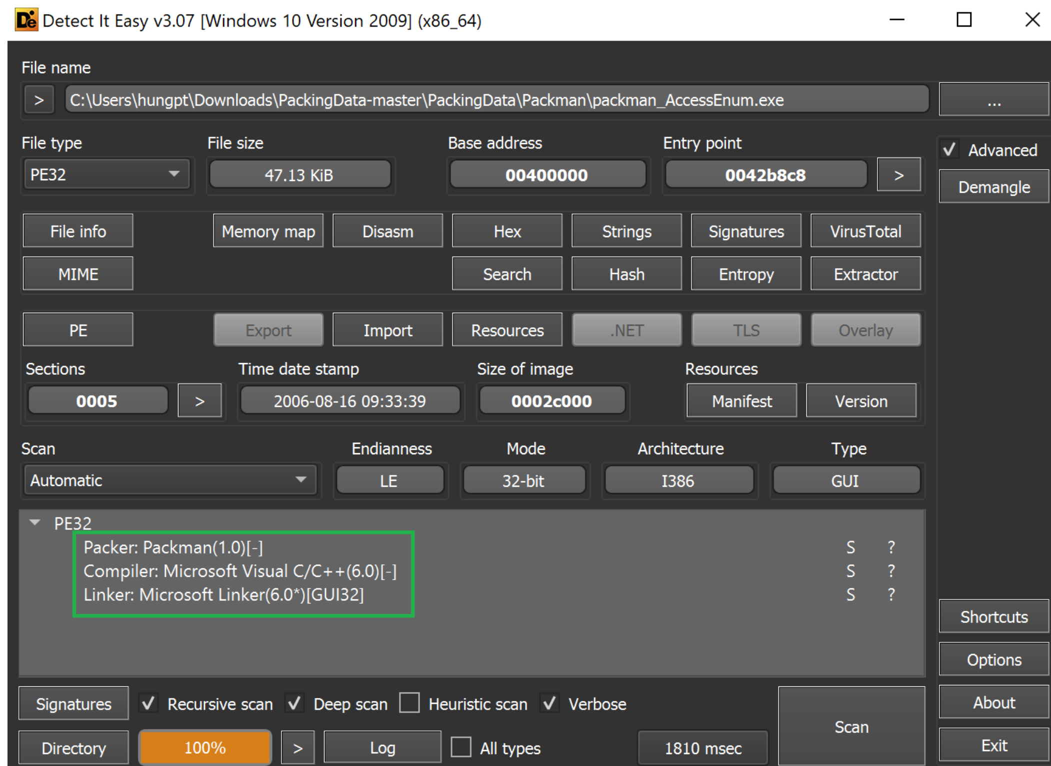

Figure 6.1: Packer identification using Detect It Easy tools

OEP Detection Problem

Original Entry Point detection refers to the challenge of detecting the beginning point of the original file during the execution of packed code

Input: Give a packed code that contains the encrypted original program.
Output: The beginning point of the original program.

Example 6.1.2:

Figure 6.2 illustrates the difference between the assembly code of `AccessEnum.exe` and its corresponding packed code by UPX packer (`upx_AccessEnum.exe`). It is clear that the original code has been hidden in the `upx_AccessEnum.exe`. The assembly code of the original file begins at `00407A98: PUSH EBP`, while the beginning point of `upx_AccessEnum.exe` is `0042B6E0: PUSHAD`.

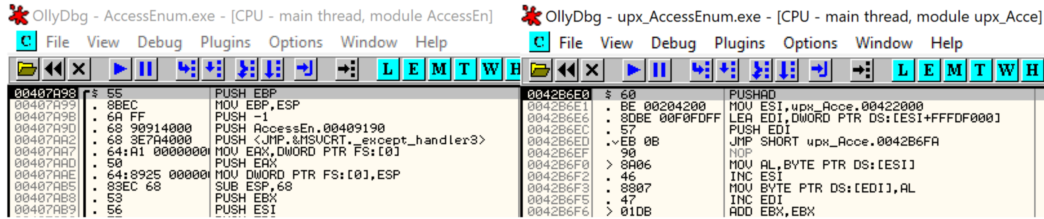

Figure 6.2: Original AccessEnum.exe and Packed AccessEnum.exe

When we continue to execute instructions in `upx_AccessEnum.exe` step by step, we will reach an instruction `0042B86C: JMP upx_Acce.00407A98` as in figure 6.3.

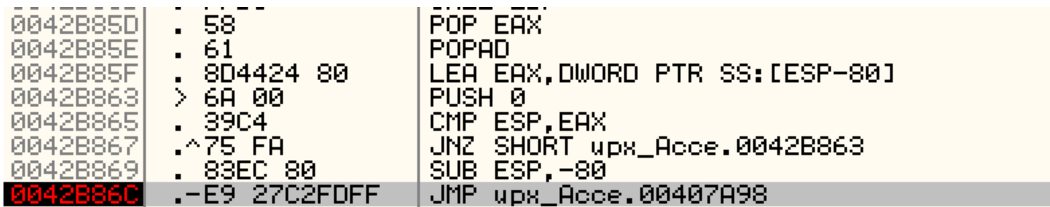

Figure 6.3: The end of the unpacking stub

This is the end instruction for the unpacking stub, and the control flow will be transferred to the original entry point after this instruction is executed. Now, the original program has been decrypted in figure 6.4, and this code in `upx_AccessEnum.exe` is identical to the original code on the left-hand side of figure 6.2.

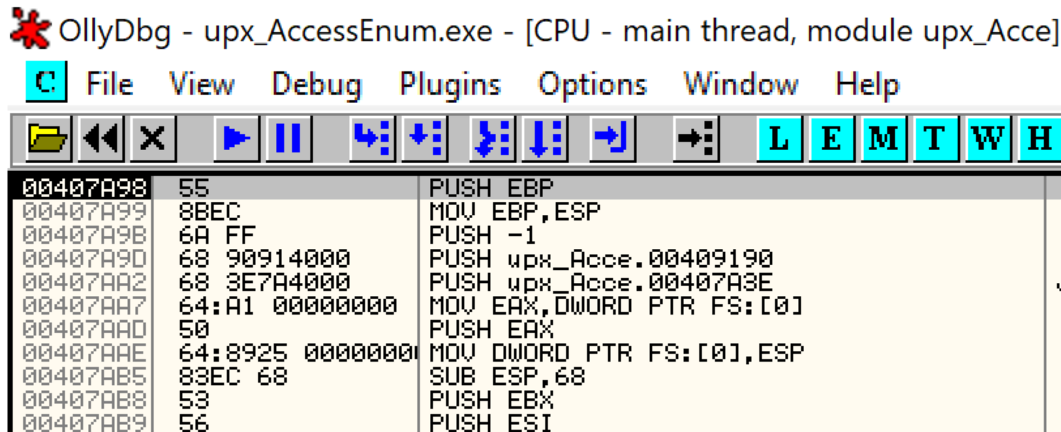

Figure 6.4: Unpacked AccessEnum.exe

Therefore, when we apply our method to `upx_AccessEnum.exe`, our expected output is `00407A98: PUSH EBP`.

Solution Overview

The process of OEP detection based on graph similarity can be divided into two major parts.

- Part 1: Average frequency vector generation. This part involves the process of obtaining average frequency vectors that represent the "signature" graphs of the unpacking stub for each packer. In this part, we have the original programs of packed codes.
- Part 2: Average frequency vector searching. This part involves the process of searching average frequency vectors from Part 1 in the CFG of new packed codes. From the result of the search process, we can obtain the OEP and the name of the packer. In contrast to Part 1, we do not have the original program of packed codes. This part represents the action we take when we encounter a new packed code with no information about the name of the packer and its original entry point.

The overview of our method will be illustrated in the figure below:

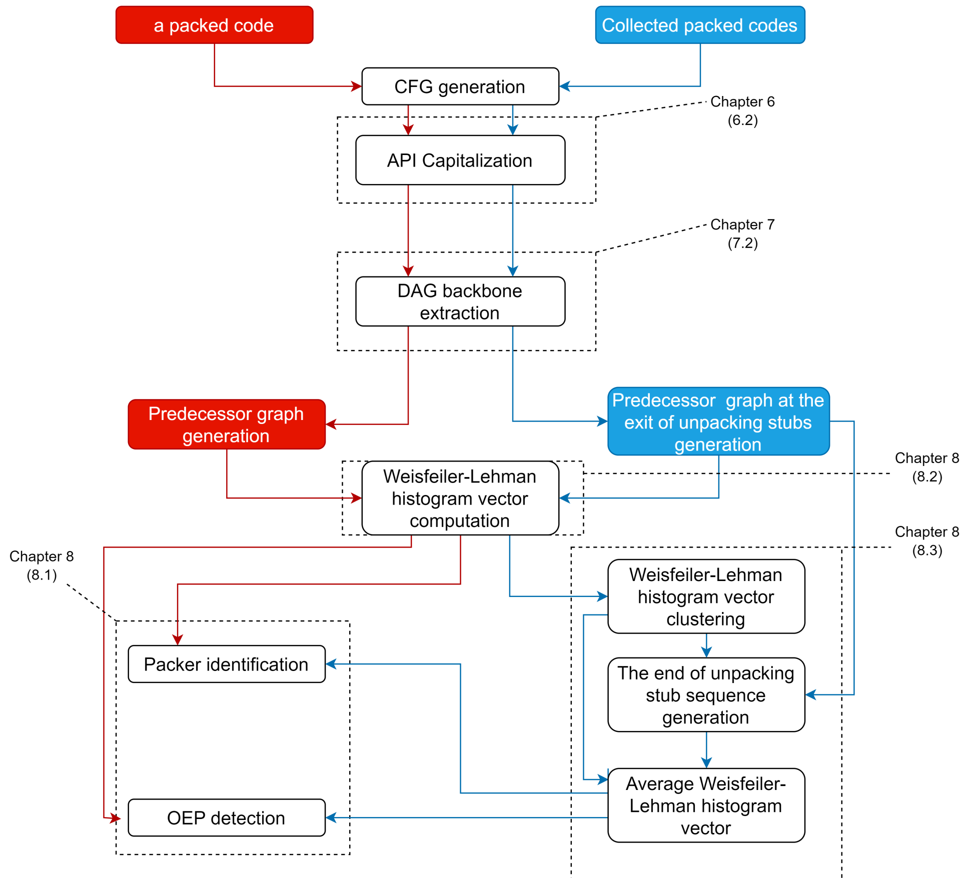

Figure 6.5: Overview of OEP detection based on graph similarity

The figure above illustrates how packer identification and OEP detection are performed in our work. The right-hand side of the figure with blue colour is related to Part 1: Average frequency vector generation. Meanwhile, the left-hand side with red colour belongs to Part 2: Average frequency vector searching. In both parts, the packed code needs the same graph refinement process which is API capitalization and acyclic backbone extraction. In addition, all graphs are required to convert into frequency vectors for the convenience of graph similarity computation. After the graph refinement process, graphs in Part 1 will be clustered into groups to find the average frequency vectors and the sequence of the end of the unpacking stub. Then, they will be used in Part 2 to find the graph of the unpacking stub in a packed code. After finishing the search process, the OEP and the packer's name will be obtained.

6.2 The CFG of binary code

To analyse malware, we need to have an abstract model that represents the packed code without actually running the binary. After obtaining an abstract model, many further analyses can be conducted based on the generated model. A popular model used in this approach is Control Flow Graph. It is clear that the precision of the CFG greatly influences the analysis in the next step, especially with the presence of obfuscation techniques. Fortunately, we know that obfuscation techniques can be handled by Dynama Symbolic Execution, and this technique can be used to generate a precise control flow graph in Chapter 5. Therefore, the first step in our method is generating a precise control flow graph using BE-PUM.

Definition 6.2.1. A Control Flow Graph (CFG) of a binary code P is a directed graph $CFG_P = (E_P, V_P)$ that represents the execution process of P. In this graph, each node represents a pair $\langle Add, Inst \rangle$ which are the address and its corresponding instruction. Or, the node just has a single value when it represents an API. Meanwhile, each edge can be expressed as a tuple $\langle u, v, l \rangle$ where:

- u is the tail node.
- v is the head node.
- l is the label of the edge. Three values of l are *True*, *False*, and *NIL*. If P represents a conditional jump, L can have a value *True* or *False*. Otherwise, l is *NIL*.

Example 6.2.1:

Figure 6.6 represents a part of the CFG of autologon.exe packed by UPX. In this CFG, the node in the green rectangle of the figure is a pair of address `0x004254d1` and instruction `jb 0x004254c0`. This node represents a conditional jump `jb`, so we can see that the edge connects this node with its child ⟨`0x004254c0`, `movb` (%`esi`), %`al`⟩ has label `True`.

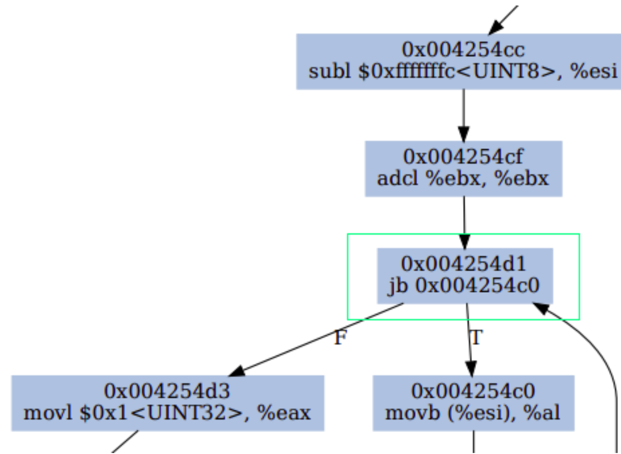

Figure 6.6: a node representing a jump condition in the CFG of autologon.exe packed by UPX

Definition 6.2.2. Given a control flow graph $G = (V_G, E_G)$, the label of a node $v \in V_G$ can be defined below:

- If v present a $\langle address\ a, instruction\ i\rangle$ then the label of v is the opcode of instruction i.
- If v present an API then the label of v is the name of the API.

Example 6.2.2:
Let's consider a node ⟨`0x00419650`, `movb` (%`edi`), %`al`⟩. The label of this node is `movb`. Meanwhile, if a node represents API `LOADLIBRARYA@KERNEL32.DLL`, its label is `LOADLIBRARYA@KERNEL32.DLL`.

Unlike the CFG of high-level programming language which is a directed graph where each edge represents the flow of control between basic blocks and each node represents a basic block, the CFG of binary code can be distinguished by the following characteristics:

- Representation:
  - Binary code CFG: The graph is constructed directly from the binary code, and identifies instructions and the control flow between them.
  - High-level programming language CFG: The graph is generated from the source code, and identifies functions, control structures, and the flow of control within the code.
- Level of abstraction:
  - Binary code CFG: The abstraction is the low level. It directly represents the execution of instructions.

  - High-level programming language CFG: High-level abstraction is the property of this kind of CFG. It focuses on the structure of programs such as loops, and conditionals.

- Complexity:
  - Binary code CFG: The construction of this kind of CFG is complex. It need to disassemble a binary file and identify instructions. It is hard to understand the logic and structure of the binary.
  - High-level programming language CFG: This kind of CFG is less complex than the CFG of binary code because the source code has a clear structure, and therefore the information about functions, control structures, and the flow of control is easy to obtain.

- Ease of Understanding:
  - Binary code CFG: understanding the CFG of binary code can be challenging because of its low-level property. It requires readers to have knowledge of assembly language and familiarity with machine instructions.
  - High-level programming CFG: Because of the characteristic of high-level abstraction, the CFG of high-level programming language can be more friendly to humans.

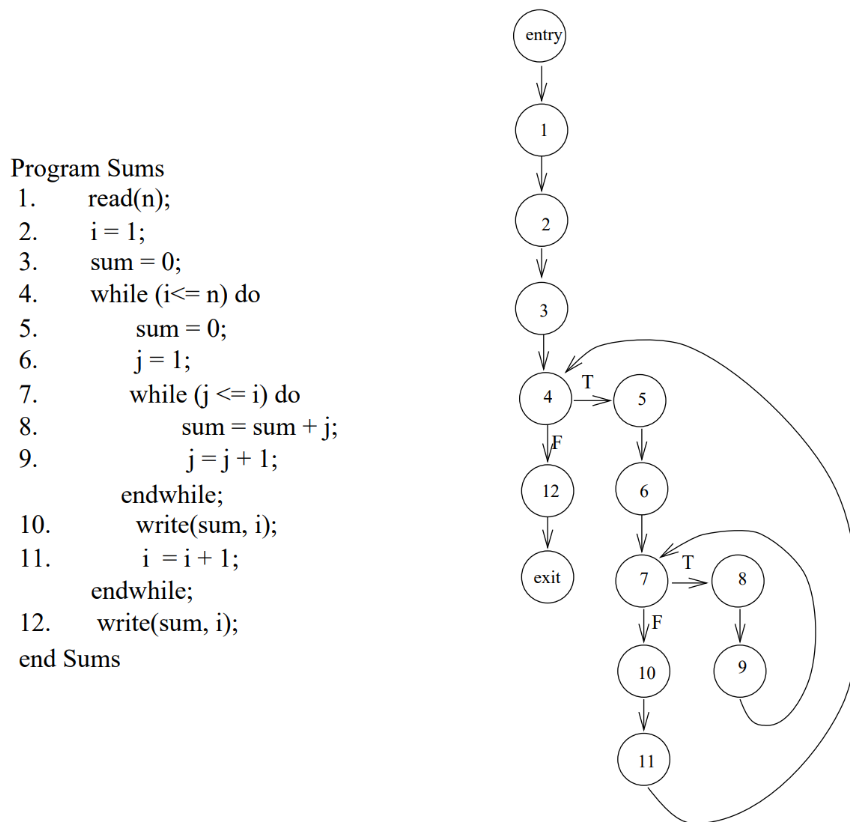

Figure 6.7: An example of control flow graph of high-level programming language [29]

Figure above illustrates the CFG of the *Sums* program that is on the left-hand side of the figure. In this illustration, a statement is considered a basic block, and the node number of the CFG corresponds to a statement number. In addition, nodes representing a transfer control will have two labelled edges that go out from it, and other edges are unlabelled. For example, node 4 in the CFG corresponds to statement 4 in the code. Because this statement is a while loop, node 4 in the CFG will have two branches True and False expressing whether the condition in the while loop is satisfied or not.

Furthermore, one of our observations in the CFG of binary files is one API can be represented by many nodes due to the case-sensitive. However, Windows does not distinguish Upper/Lower characters in API names and needs to unify by capitalization.

Procedure 6.2.1. API name capitalization process: Let the set of API names we need to capitalize are $\{a_1, a_2, a_3, ..., a_n\}$, and:

- accessors(u) is the set of accessors of node u
- successors(u) is the set of successors of node u

Assume that these API names are capitalized into a new API name a, then we have:

$$accessor(a) = \bigcup_i accessor(a_i) \quad \text{and} \quad successor(a) = \bigcup_i successor(a_i) \tag{6.1}$$

Former API names $\{a_1, a_2, a_3, ..., a_n\}$ and their edges will be removed from the graph.

Example 6.2.3:
Figure 6.8 illustrate the process of API name capitalization. On the left-hand side is the original CFG of packed code. In this case, the name of the API is case-sensitive. In addition, the green node represents the API `VirtualFree@KERNEL32.dll`. This node has accessor nodes $n2$ and $n3$, and successors $n4$. Meanwhile, nodes $n1$ is the accessor node of the blue node and its successor nodes are $n5$ and $n6$. After API name capitalization, the original CFG becomes the CFG on the right-hand side with a red node representing both API names in the original CFG. Now, this new node has $n1$, $n2$, and $n3$ as its accessor nodes, and $n4$, $n5$ and $n6$ are its successor nodes in this new CFG.

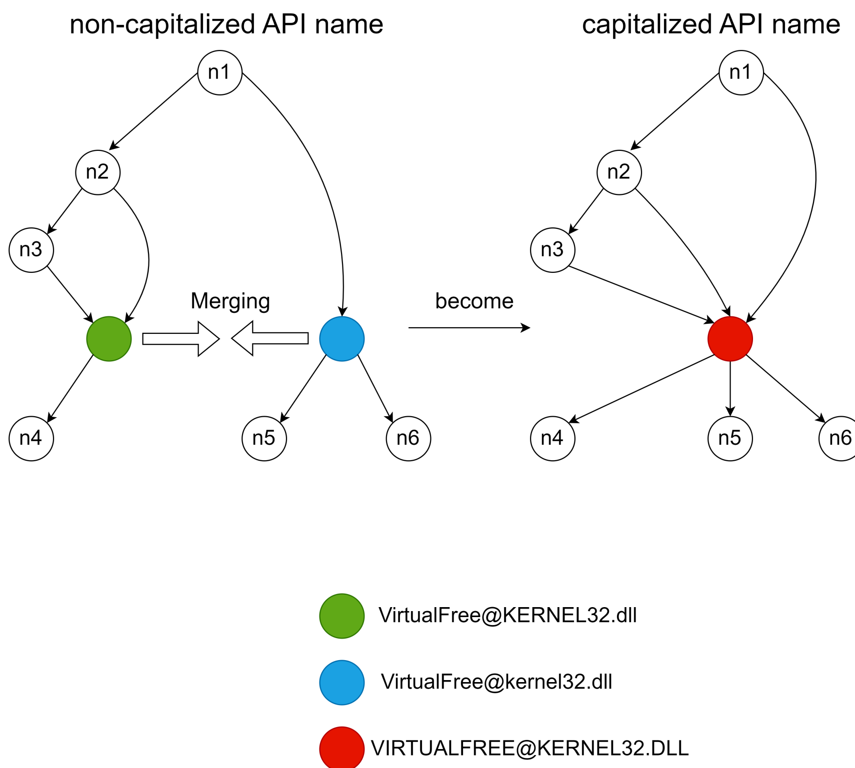

Figure 6.8: An example of API name capitalization

6.3 CFG of packed code

It is important to remind that the CFG of a packed code can be changed during the execution of unpacking stubs. Therefore, we want to clarify that the control flow graph

of the packed code mentioned in this work implies the CFG of the packed code when the unpacking stub has finished. As a result, we can find the CFG of the unpacked payload in the CFG of packed code at this time. In addition, there will be a node in the CFG of packed code that represents its OEP.

Definition 6.3.1. In the CFG of packed code, the **original entry point node (OEP node)** is a node in the graph such that the subgraph with the OEP as a source node is the control flow graph of unpacked code.

Therefore, The OEP detection problem now can define as the task of finding the OEP node in the CFG of packed code.

Definition 6.3.2. In the CFG of packed code, the **end-of-unpacking nodes** are the parents of the OEP node.

From our observation in CFG of packed codes, the OEP node has only one parent.

Example 6.3.1:
Figure 6.9 shows an example of an OEP node and end-of-unpacking node in the CFG of Bginfor.exe packed by FSG packer. The OEP node here is the $\langle 0x005b5d42, call\ 0x5c4706\rangle$, while its parent $\langle 0x008050e1, je\ 0x005b5d42\rangle$ is an end-of-unpacking node.

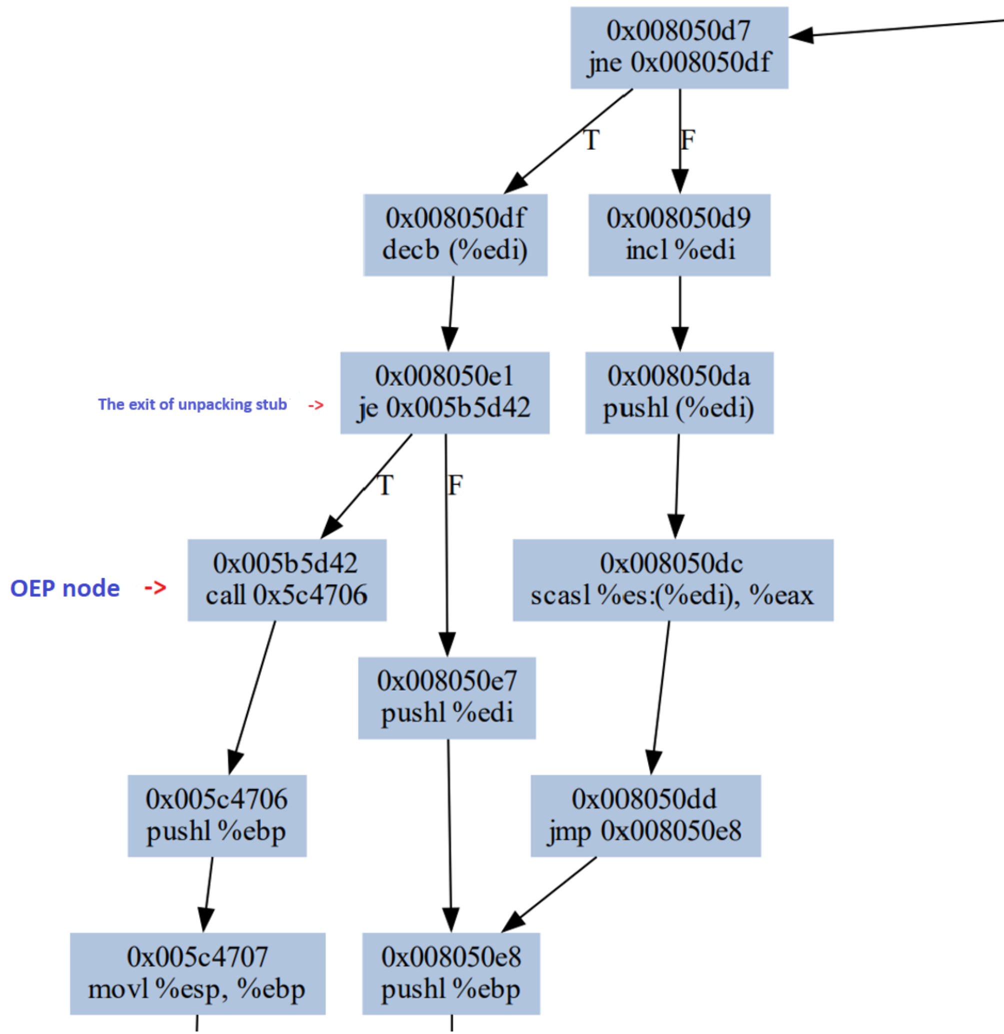

Figure 6.9: An example of OEP node and end-of-unpacking node

Chapter 7

Control Flow Graph of unpacking stub

7.1 Reasons for avoiding retreating edges

To solve our problems, we begin with the hypothesis that a similar class of CFGs of unpacking stubs does not cross different packers. If this hypothesis works, our problems can be solved based on detecting the unpacking stub because the unpacking stub can be regarded as a signature for each packer. Also, the OEP of a packed code can be detected because the control flow will be transferred to OEP when the execution of the unpacking stub is finished. To verify this hypothesis, we have tried to observe the CFG of unpacking stubs of several packed codes by the same packer. Therefore, we have to obtain the CFG of the unpacking stubs. To do that, we need to know the original program of a packed code, then we will know where is the exit of unpacking stubs because the difference between the original program and the packed code can be regarded as the unpacking stub. Next, the CFG of unpacking stubs will be the predecessor graph at the exit of the unpacking stub. However, we also observed that there are edges whose tails belong to the CFG of the payload and their heads belong to the CFG of the unpacking stub in some cases.

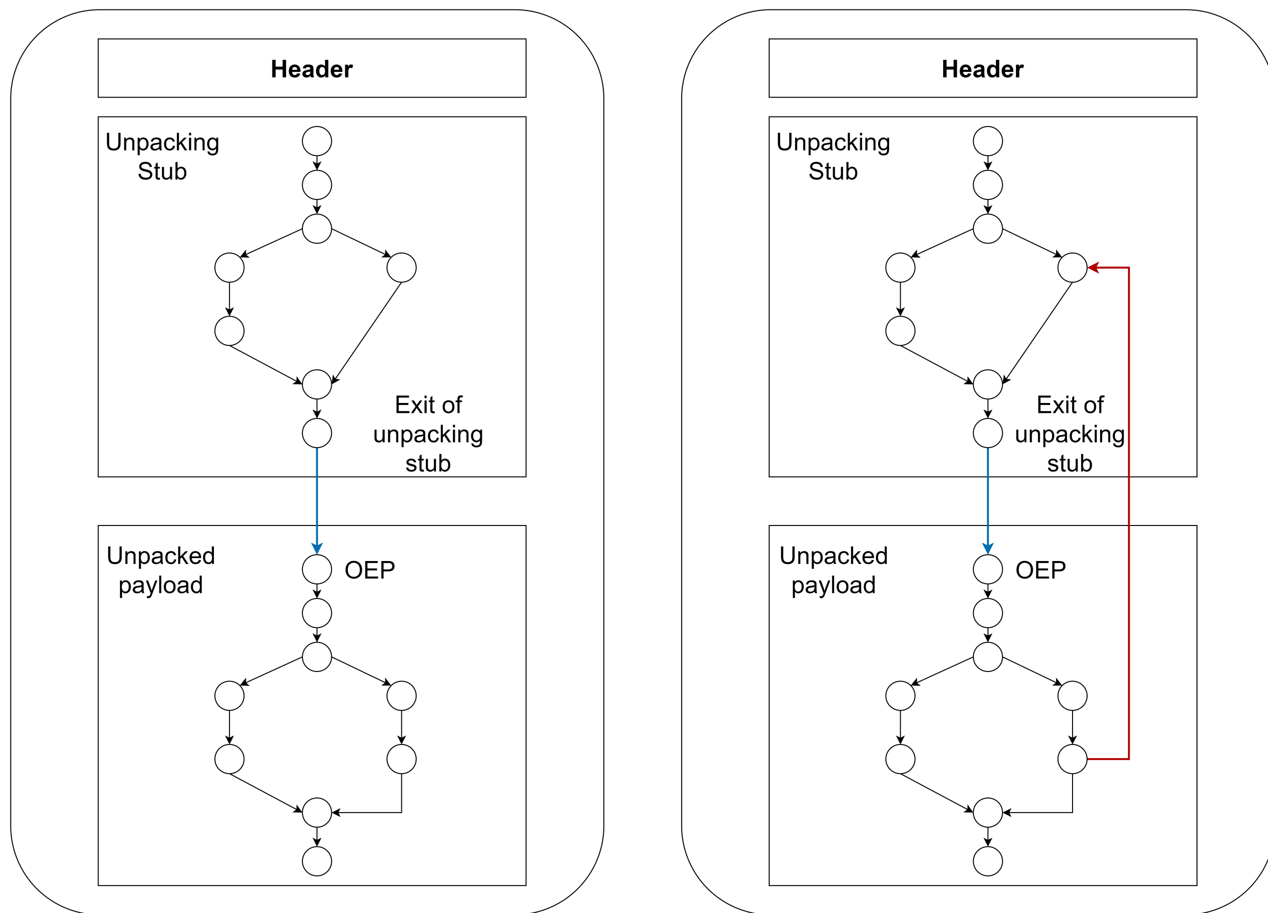

Figure 7.1: An illustration of back-edge from unpacked payload to unpacking code

For example, in the CFG of whois.exe packed by yodaC, node ⟨`0x004089f5`, `call %esi`⟩ go to node API `GetProcAddress@KERNEL32.dll` that has been used in unpacking stub. This situation is illustrated on the right-hand side of Figure 7.1. From our observation, when the unpacking stub and the unpacked payload call the same API, it is likely a cycle will be created. From our point of view, this design for illustrating API call as a node in the CFG of a program may be not a good choice because API does not belong to packed code. Then, when we represent API as a node of CFG of a program may be not suitable.

To mitigate these cases, we decided to approximate the CFG of packed code by removing all retreating edges in this CFG to break the cycle before generating the predecessor graph. However, the set of retreating edges in a graph depends on the order of nodes traveled in the DFS procedure.

Example 7.1.1:

Figure 7.2 illustrate that different DFS order can lead to a different set of retreating edge. On the left-hand side of this figure, the DFS travel order is $n1 \rightarrow n2 \rightarrow n3$, because $n2$ already is in the travel path, so the edge connect from $n3$ to $n2$ is a retreating edge. In contrast, the travel order on the right-hand side of the figure starts from $n1$ to $n3$, and

then from $n3$ to $n2$. Next, $n2$ go back to $n3$. In this case, the edge from $n2$ to $n3$ is a retreating edge.

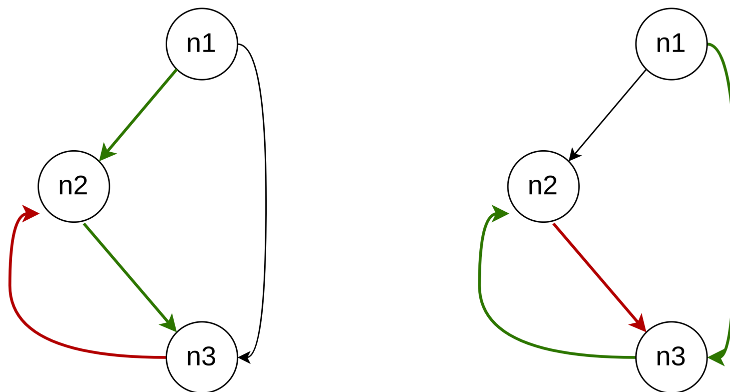

Figure 7.2: An example of retreating edges on different travel order of DFS

Fortunately, all most normal program has a control flow graph that is reducible [30]. This implies that retreating edges in this graph are also back-edges and these edges are unique. Although this research doesn't mention the CFG generated from a binary code, we expect this property also holds for binary code as well. In addition, we also introduce a strategy for our DFS procedure to remove retreating edges that connect unpacked payload and unpacking stub that happens because of calling the same API. The details of our DFS procedure will be described in Section 7.2. Now, after removing retreating edges, the new graph will become a directed acyclic graph (DAG). Therefore, we can easily obtain the predecessor graph at the exit of the unpacking stub.

7.2 The CFG of the unpacking stub characterizes a packer

Acyclic backbone extraction

After API name capitalization, we need to remove retreating edges to break cycles in this CFG and a DAG will be extracted. From a DAG, we can separate this graph from a node u into two graphs more easily.

Now, to remove retreating edges, we can use depth-first search (DFS) to detect these edges. In particular, we will traverse the CFG from the entry node of it, then we keep a stack where we push a node into the top of the stack when this node is reached by DFS and pop it when the algorithm backtracks. If the algorithm ever hit a node that's already on the stack, then we found a retreating edge indicating that there is a cycle. Next, we will just remove this edge from the graph to break the cycle.

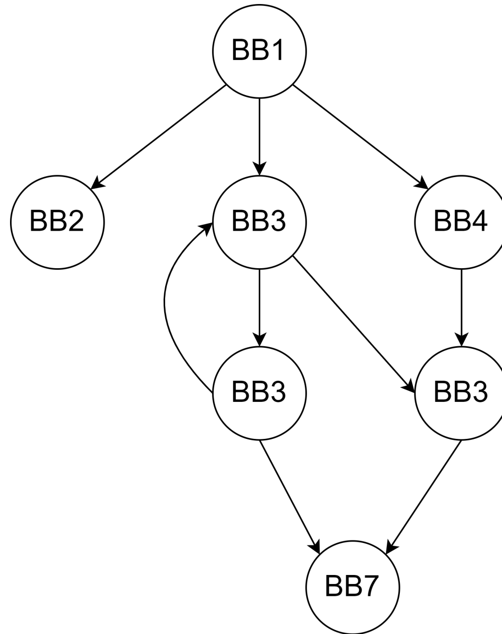

Figure 7.3: An example of CFG with back-edges

For example, let's assume we are traversing the CFG in Figure 7.3. The DFS starts at the entry point $BB1$. First, the DFS explores the leftmost branch of BB1 but quickly backtracks as it hits a dead end. It then enters the middle branch, leading from BB_1 to BB_3, and continues its traverse through BB_5. After that, it hit the BB_3 again because the node BB_3 currently is in the stack. Therefore, there is a back-edge from BB_5 to BB_3. Let's observe the stack during this procedure. The example below shows how the DFS state evolves and how this algorithm detects retreating edges in the CFG. In step 5, we can see that BB3 already have been in the stack, so the algorithm detects a retreating edge.

```
The state of stack during the process of DFS
0: [BB1]
1: [BB1, BB2]
2: [BB1]
3: [BB1, BB3]
4: [BB1, BB3, BB5]
5: [BB1, BB3, BB5, BB3]
```

Although we can expect these retreating edges to be unique because most of the CFG of a normal program to have retreating edges are back edges in [30], this paper still is an empirical study and cannot ensure this holds for all the cases. Therefore, we also introduce a strategy for travelling order of DFS with two purposes:

1. The set of retreating edges will be unique with this strategy when the DFS procedure is performed.

2. With this strategy, retreating edges from the unpacked payload to the unpacking stub that happened when they call the same API will be removed.

Because we want to remove treating edges $u \to v$ with v representing an API, our strategy will try to make API nodes visited as soon as possible.

Definition 7.2.1. DFS order travelling strategy:
Let's assume DFS is performing, and node u is the current visited node in the travelling path of DFS.
Let $M = sorted\{v_1, v_2, v_3, ..., v_n\}$ be the sorted set by descending priority of successor nodes of u that are not visited. The node that has maximum priority will be the next visited node.
Let's consider node v_i and node v_j in M. Their priority can be defined below:

1. If v_i represent a API and v_j represent a $\langle address, instruction \rangle$, then priority($v_i$) > priority($v_j$).
2. If both v_i and v_j represent a API, priority(v_i) > priority(v_j) if
   - indegree(v_i) > indegree(v_j)
   - indegree(v_i) = indegree(v_j) and alphabet(v_i) > alphabet (v_j).
3. If both v_i and v_j represent a $\langle address, instruction \rangle$, priority($v_i$) > priority($v_j$) if address in v_i > address in v_j.

Example 7.2.1:
Let's consider the set
$M = \{\langle$`0x004c4993, adcb %dl, %dl`$\rangle$, `LoadLibraryA@kernel32.dll`, $\langle$`0x004c49e8, jb 0x004c49fb`$\rangle\}$.
After sorting, the descending order is:
$\{$`LoadLibraryA@kernel32.dll`, $\langle$`0x004c49e8, jb 0x004c49fb`$\rangle$, $\langle$`0x004c4993, adcb %dl, %dl`$\rangle\}$.
Therefore, the node `LoadLibraryA@kernel32.dll` will be selected to visit first.

Predecessor graph generation

Before diving into the way how a predecessor graph is generated, we want to introduce the definition of a reverse graph.

Definition 7.2.2. The reverse of a directed graph $G = (V, E)$ is a direct graph $G^R = (V^R, E^R)$ satisfying:

$$E^R = \{(v, u) : (u, v) \subseteq V\} \quad \text{and} \quad V^R = V \tag{7.1}$$

Example 7.2.2:
Figure 7.4 shows a directed graph (left-hand side graph) and its reverse (right-hand side graph). On the left-hand side, we have node C going to node A. In contrast, node A goes to node C on the right-hand side. Similar to other edges, the direction of each edge in the original graph has been reversed on the right-hand side.

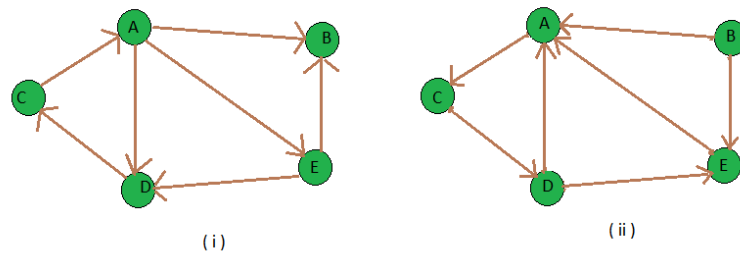

Figure 7.4: An example of reverse of a directed graph

Now, from the definition of the predecessor graph Pre_u^K from node u in DAG K, we can see that when we reverse the direction of each edge in K, all of the nodes can reach u in K will be nodes reached by u in the revered graph. Therefore, with the purpose of easy implementation, we will generate Pre_u^K via the reverse graph K^R.

Now, we will describe how we obtain a predecessor graph of a node U in the DAG of the CFG of a packed code.

The procedure of predecessor graph extraction can be expressed with three steps in Figure 7.5.

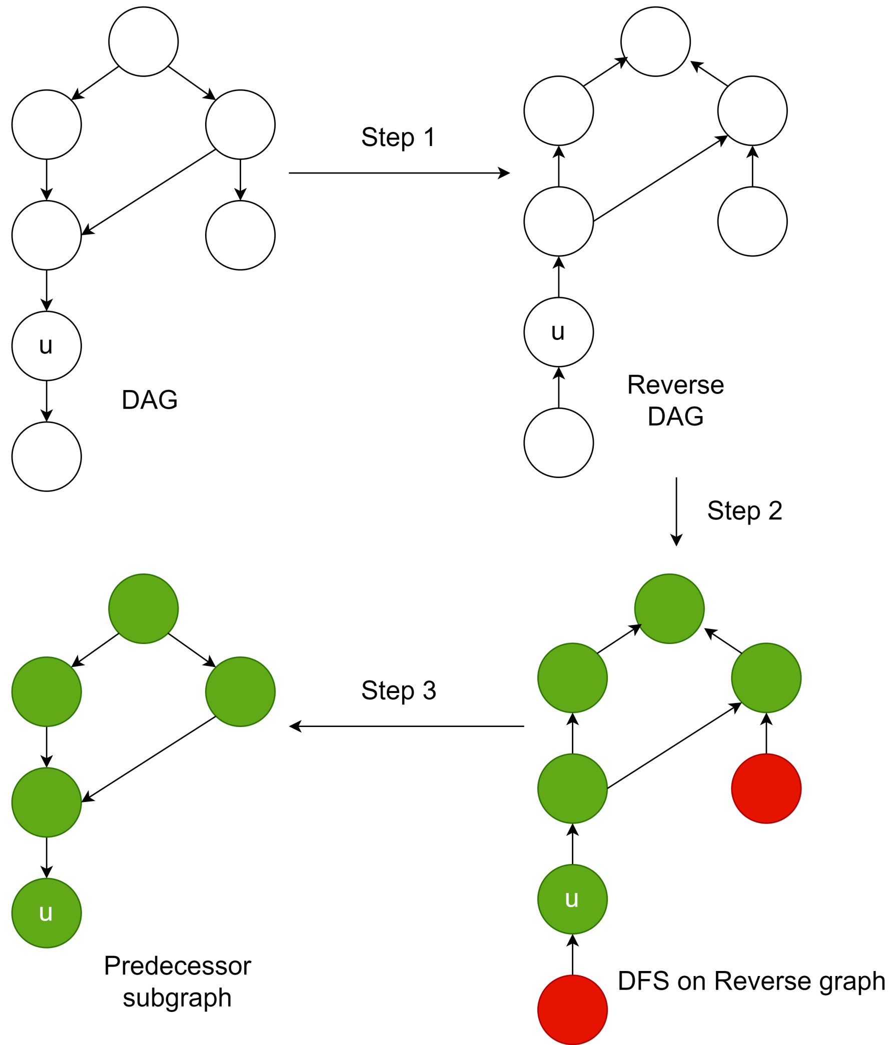

Figure 7.5: Predecessor graph extraction

Definition 7.2.3. Let's assume we have a directed acyclic graph $K = (V_K, E_K)$, and a given node $u \in V_K$. The predecessor graph extraction from node u includes 3 steps:

- **Step 1**: From K obtain its reverse graph K^R.
- **Step 2**: Perform DFS procedure from node u in K^R.
  Let $I = \{v : v \in V_K, u \xrightarrow{*} v\}$.
- **Step 3**: Extract a predecessor graph from node u with its vertex set is I, and its edges set is $\{$(u→v) : (u,v) $\in E_K, u, v \in$I$\}$.

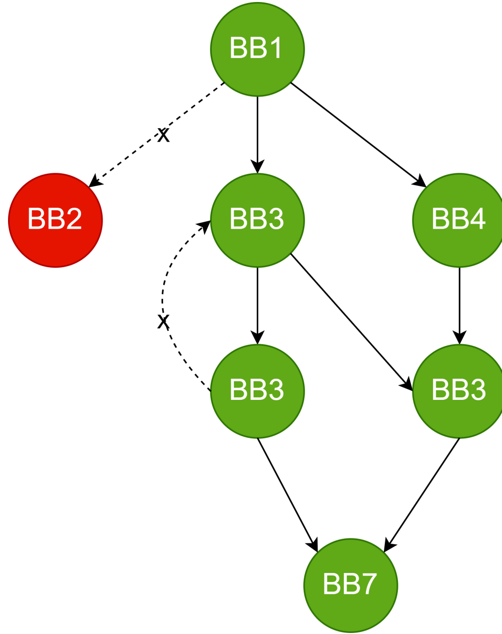

Figure 7.6: An example of predecessor graph

Now, after performing retreating edges removal and predecessor graph extraction. The example in Figure 7.3 will become the predecessor graph in Figure 7.6. This graph containing $BB1$, $BB3$, $BB4$, $BB5$, $BB6$, and $BB7$. These nodes are labelled by green colour, meanwhile, red node $BB2$ indicates that this node does not belong to the predecessor graph. In addition, the dashed edge is removed because it is the retreating edge.

From our observation, these predecessor unpacking graphs in packed codes by the same packer are similar.

Example 7.2.3:

Figure 7.7 shows the predecessor unpacking graph of packed codes from 3 different packers which are FSG, MEW, and UPX. These graphs from the same packer are similar. We have computed the similarity between the predecessor unpacking graph in packed codes by the same packer in the figure above, and the score between them is more than 0.95.

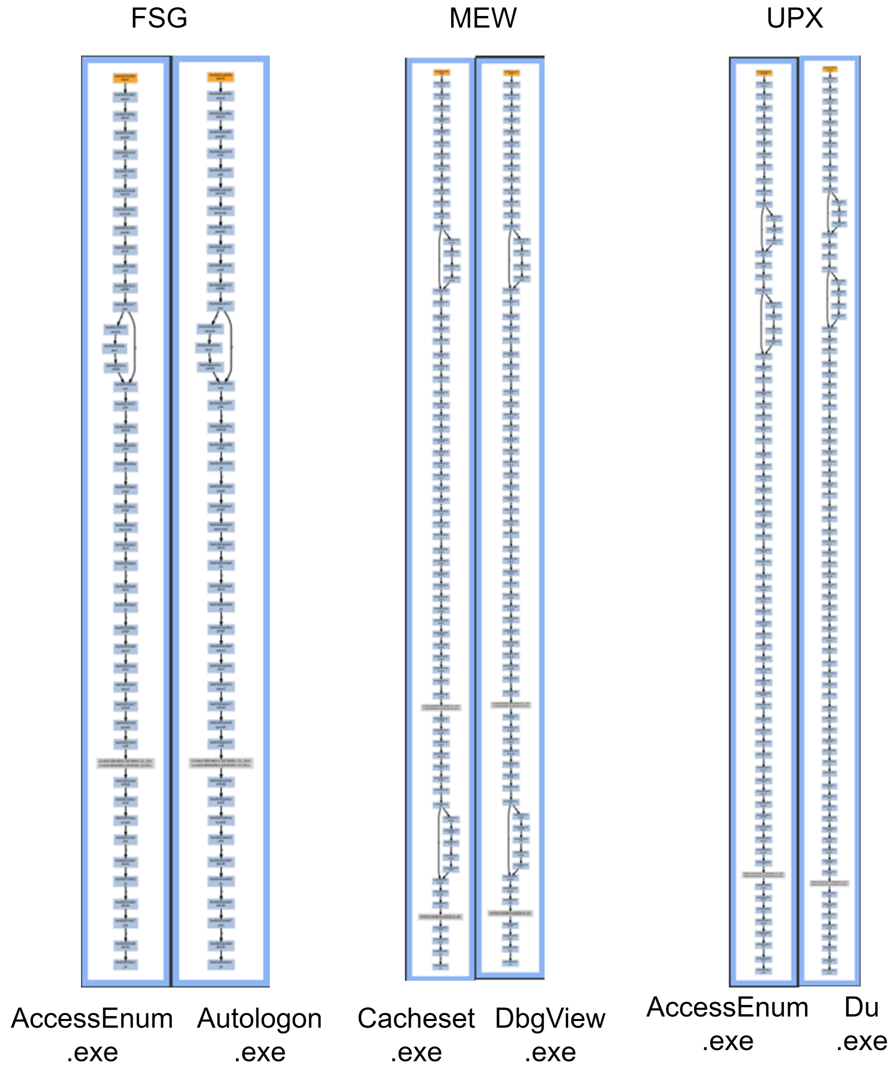

Figure 7.7: The similarly between end-of-unpacking graph in the same packer

On the other hand, we also observed that a packer can have several structures of the predecessor unpacking graph. For example, the figure below shows 2 kinds of structures of predecessor unpacking graphs in packed codes by WinUpack.

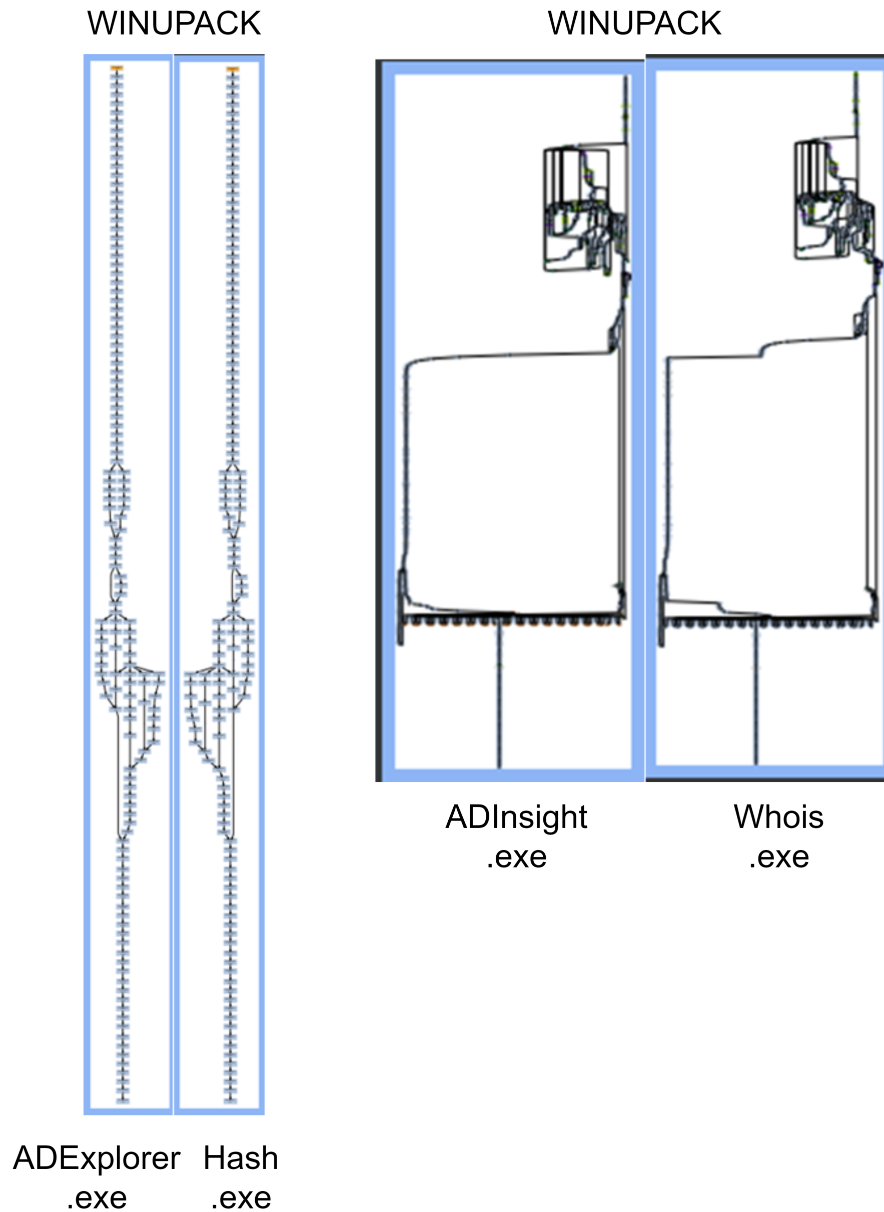

Figure 7.8: An example of winupack has 2 kinds of end-of-unpacking graph

It is clear that the predecessor graph with the end-of-unpacking node as a sink node can be a property to detect the end-of-unpacking stub. Then, the OEP can be detected based on this end-of-unpacking node.

7.3 Consistency of the end sequence

Although these predecessor graphs at the exit of unpacking stub by the same packer are similar, they still are not identical because of many obfuscation techniques applied by the

packer. However, we found that the end sequences of the last k instruction before the control flow transferred to OEP in packed code are consistent. Currently, we observe with $k = 5$. The definition of end sequences will be described below:

Definition 7.3.1. Given a predecessor graph $K = (V_K,\ E_K)$ with node $u \in V_K$ is the sink node. The end sequence of K is seq_K can be defined below:

$$seq_K = \langle label(u), label(v_1), label(v_2), ..., label(v_q) \rangle \tag{7.2}$$

Here, $label(u)$ is an function such that:

$$label(u) = \begin{cases} \text{the opcode of instruction } u & \text{if } u \text{ is a node representing an instruction} \\ \text{API name} & \text{if } u \text{ represents an API.} \end{cases}$$

And $v_i \in V_K$, v_i is the parent of v_{i-1} if $i > 1$ and $i \leq q$, the indegree of v_{i-1} equal to 1, v_1 is the single parent of u.

Definition 7.3.2. Given a predecessor unpacking graph $K' = (V_{K'},\ E_{K'})$. The end sequence $seq_{K'}$ of this graph will be called the end-of-unpacking sequence.

Example 7.3.1:

Let's consider table 7.1. This table shows several end-of-unpacking sequences for UPX, FSG, MEW packers. For instance, the end-of-unpacking sequence of UPX is *jmp, subl, jne, cmpl, pushl*.

Packer	the sequence of the end of unpacking stub
UPX	jmp, subl, jne, cmpl, pushl
FSG	je, decb, jne, decb, je
MEW	ret, jne, testl, stosl, GETPROCADDRESS-KERNEL32-DLL

Table 7.1: End-of-unpacking sequence of UPX, FSG and MEW

The end sequence of unpacking stubs has an important role in our method because weisfeiler-lehman kernels is a statistical method and it just approximates a graph into a vector. So, sometimes, the noise in a graph can make this method detect the unpacking stubs incorrectly. However, an end sequence is a property of the unpacking stub for each packer. Therefore, it can reduce the number of unpacking stub candidates and confirm again whether a node in a graph is an exit of an unpacking stub or not.

Besides empirical examples from the previous section, we also confirm our hypothesis by applying a clustering algorithm. If the CFG of unpacking stubs characterizes a packer, we believe that after the clustering procedure, (1) each class does not cross different packers and (2) the end sequence is the same in each class. Particularly, we use DBSCAN [31] algorithm for the task of clustering on all 771 non-malware packed codes from 12 packers. In addition, we run DBSCAN with fixed parameters $min_sample = 2$ and $metric = cosine$. Next, we will experiment with different values of the eps parameter.

eps	0.1	0.09	0.08	0.07	0.06	0.05	0.04	0.03	0.02
number of class	11	11	12	14	16	16	18	18	21
Hypothesis (1)	False	False	False	False	False	False	True	True	True
Hypothesis (2)	False	False	False	False	False	False	False	False	True

Table 7.2: The effect of eps on DBSCAN clustering

Table above shows the effect of the eps parameter on clustering mixed-packed code from different packers. In this experiment, $eps = 0.04$ is enough to make hypothesis (1) hold, but there are some classes where the end sequence is not consistent. Finally, both hypotheses hold with eps = 0.02. Overall, when the value of eps becomes smaller, the number of classes (templates) will increase, but the end sequence will become more consistent. So, with the allowance *eps* is enough small, our hypothesis is correct, and this shows that the CFG of unpacking stubs can be regarded as a signature for each packer.

Chapter 8

Packer identification and OEP detection

8.1 Template matching for packer identification and OEP detection

Based on our observation from previous sections, our approach consists of two phases. In phase 1, template setup is performed. In this phase, we will obtain the template for unpacking stubs of each packer. This template is the pair of the average of Weisfeiler-Lehman histogram vectors and the end sequence. In phase 2, we have an unknown packed code, and templates prepared in phase 1 will be used for template matching to find the unpacking stub. From this unpacking stub, we can detect packer identification and OEP. The template matching process consists of 4 steps below:

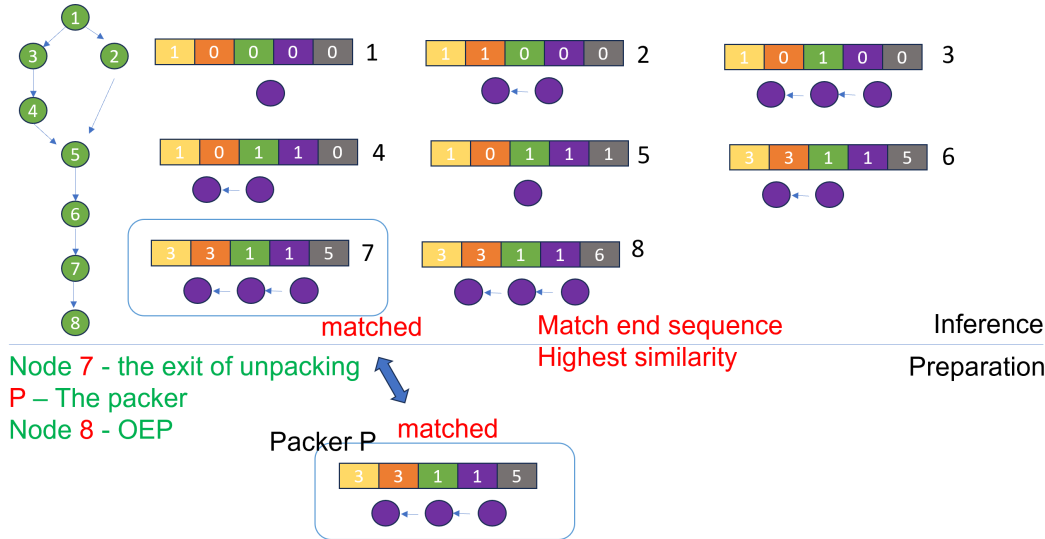

Figure 8.1: Template matching for Packer Identification and OEP detection

- **Step 1**: The first step involves generating predecessor graphs for all nodes in the directed acyclic graph of the CFG of the packed code.
- **Step 2**: Converting these graphs into Weisfeiler-Lehman histogram vectors.
- **Step 3**: For each predecessor graph from previous steps, When the tail of the graph matches with the end sequence of a template, we check the similarity between its Weisfeiler-Lehman histogram vector and that in templates.
- **Step 4**: Finally, we can find an optimal pair of a template and a predecessor graph. In which, the similarity between the histogram vector in the template and that of the graph is highest. Denote the sink node of the graph as k, k will be the exit of the unpacking stub, OEP can be detected based on k. Besides, the packer corresponding to the template serves as the answer for packer identification.

Fig. 8.1 shows steps 3 and 4 in the template matching procedure after we have converted all predecessor graphs into Weifeiler-Lehman histogram vectors. In this figure, the Weisfeiler-Lehman histogram vector generated from note 7 and that vector in template packer P has highest similarity. In addition, the two corresponding end sequences in node 7 and the template are identical. Therefore, node 7 is the exit of the unpacking stub, node 8 is the OEP, and packer P is the used packer. The complex parts of our methods will be computing Weisfeiler-Lehman graphs, Weisfeiler-Lehman histogram vectors, and template setup. Therefore, these parts will be described in the next two sections.

8.2 Computing weisfeiler-lehman graph and its histogram vector

From our observation in Chapter 7, the predecessor graph at the exit of unpacking stubs characterizes a packer. Now, we also know how to obtain these predecessor graphs from any node in a CFG of packed code. The next task is to find an efficient way that measure the similarity between graphs. There are many ways to compare graphs such as edit distance computation, or graph embedding with the development of deep learning methods. However, these methods have their drawbacks. For these graph similarity methods based on edit distance computation, they need a huge computation. On the other hand, methods based on deep learning require a lot of data. To overcome this, the idea of this section is to introduce an efficient method that generates a vector from a graph, then we can easily measure the similarity between them by using some distance metrics.
The algorithm we choose to transfer a graph into a vector based on Weisfeiler-Lehman kernels. However, this method was originally designed for undirected graphs, meanwhile, graphs in this research are directed graphs. Therefore, it is not reasonable if we apply this method directly to our graphs. As a result, we decided to make a slight adjustment to it. Particularly, there are two changes from Weisfeiler-Lehman Kernels in our research:

- The Weisfeiler-Lehman relabeling function: This change aims to adapt to the targeted graph in our research are directed graph.

- Kernel function: This aims to normalize the similarity into the range 0 to 1, so we choose cosine similarity in this work.

8.2.1 Computing weisfeiler-lehman graph

In the definition 2.2.1, the new label of a node will depend on the label of its neighbour. However, to emphasize of property of a directed graph, we only update the label of a node based on the label of its accessors in the graph. Therefore, the procedure of computing weisfeiler-lehman graph in our work can be described below.

Procedure 8.2.1. The weisfeiler-lehman relabeling function on a directed graph $G = (V, E)$ is a function r such that $r((V, E, l_{i-1})) = (V, E, l_i)$ where l_i is the label function of graph G after iteration i and l_0 is the label function of original graph G without performing Weisfeiler-Lehman relabeling procedure.

The computation of function r will include 3 steps:

- **Step 1**:Multiset-label determination.
  - Each node v in V is assigned $M_i(v) = \{l_{i-1}(u) : u \in accessors(v)\}$.
- **Step 2**: Sorting each multiset.
  - The set $M_i(v)$ will be sorted in ascending order and $s_i(v)$ is created by concatenating all element in $M_i(v)$.
  - $s_i(v) = l_{i-1}(v) + s_i(v)$.
- **Step 3**: Relabeling.
  - For all node v in V, $l_i(v)$ is equal to $s_i(v)$.

Example 8.2.1:

Let's consider the example in the figure 8.2. The left-hand side is a directed graph G, its node labels are $\{A, B, C, D, E\}$. Assume that we are in iteration i, the procedure of relabeling will happen as below:

- Step 1: :Multiset-label determination.
  - $M_i(node\ 1) = \emptyset$
  - $M_i(node\ 2) = \{A\}$
  - $M_i(node\ 3) = \{A\}$
  - $M_i(node\ 4) = \{C, B\}$
  - $M_i(node\ 5) = \{B, D\}$
- Step 2: Sorting each multiset.
  - $M_i(node\ 1) = \emptyset$ & $v_i(node\ 1) = A$

- $M_i(node\ 2) = \{A\}$ & $v_i(node\ 2) = BA$
- $M_i(node\ 3) = \{A\}$ & $v_i(node\ 3) = CA$
- $M_i(node\ 4) = \{B, C\}$ & $v_i(node\ 4) = DBC$
- $M_i(node\ 5) = \{B, D\}$ & $v_i(node\ 5) = EBD$

- Step 3: Relabelling.
  - $l_i(node\ 1) = A$
  - $l_i(node\ 2) = BA$
  - $l_i(node\ 3) = CA$
  - $l_i(node\ 4) = DBC$
  - $l_i(node\ 5) = EBD$

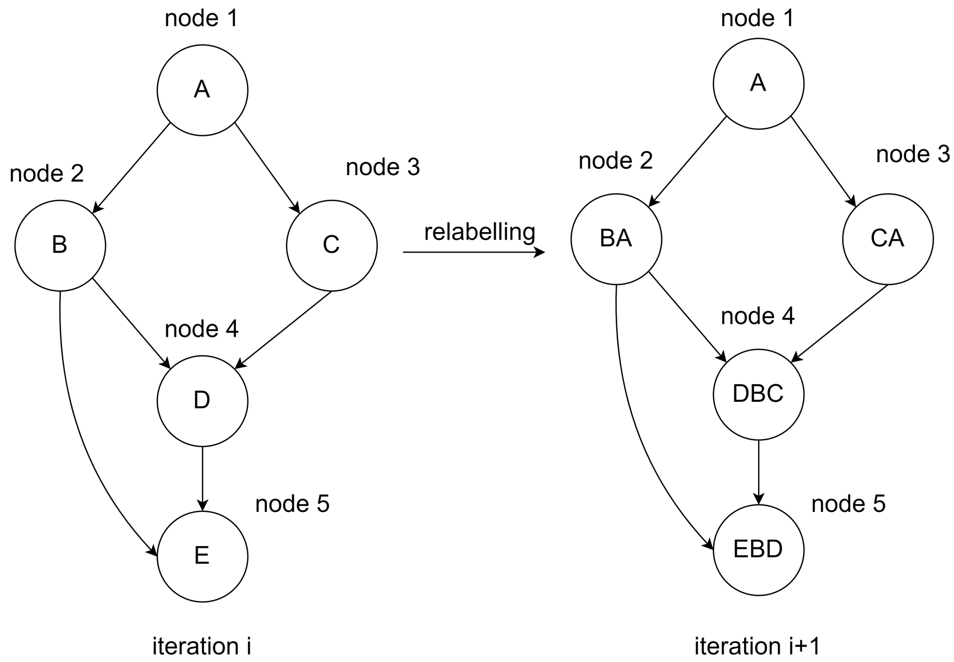

Figure 8.2: An example of relabelling function

So, with the weisfeiler-lehman relabeling function above, we can computer the weisfeiler-lehman graph at height i (denoted as G_i) of the labelled graph $G = (V, E)$ following the definition 2.2.1 by applying relabeling function i times on graph G.

8.2.2 Computing weifeiler-lehman histogram vector

Let us denote:

- Σ_i as the set of node labels of weisfeiler-lehman graph $G_i = (V, E, l_i)$ of graph G.
- Frequency function of graph G_i is:

$$f_i : \Sigma_i \to \mathbb{N} \quad \text{and} \quad f_i(l) = |\{u : u \in V, l_i(u) = l\}|$$

Example 8.2.2:
In figure 8.2, we have $\Sigma_{i+1} = \{A, BA, CA, DBC, EBD\}$, and the results of function f_{i+1} for all nodes are one.

Now, we will define the weifeiler-lehman histogram vector of a directed graph G below:

Definition 8.2.1. The weifeiler-lehman histogram vector of the directed graph G with h iteration is a tuple.

$$\mathbf{v}_G = t_1 \oplus t_2 \oplus \ldots \oplus t_h \tag{8.1}$$

Where,

$$t_i = ((\sigma_{i,1}, f_i(\sigma_{i,1})), \ldots, (\sigma_{i,|\Sigma_i|}, f_i(\sigma_{i,|\Sigma_i|})))$$

Because we may need to computer the similarity between many graphs and the node labels of these graphs may be different, we need to store the node labels in a weifeiler-lehman histogram vector to distinguish the labels of these graphs. For example, we assume there are two vectors and their frequency vectors are $v_1 = ((A, 2), (B, 2), (C, 5))$ and $v_2 = ((E, 2), (F, 2), (G, 5))$. If we do not store the labels these vectors will be considered the same $(2, 2, 5)$ but actually not because their labels are different. Especially, the node labels in our work are really important because they present the instructions of a binary and show us how a binary is executed.
Furthermore, with this strategy, we can see that the dimension of weifeiler-lehman histogram vectors can vary between many graphs. Therefore, we cannot compute the cosine similarity between them. Consequently, it is necessary to convert their vectors to new vectors sharing the same dimension.

Definition 8.2.2. Given n directed graph $\{G_i : 1 \leq i \leq n\}$. Let:

- Γ_i is the set of node labels that occur at least once in $\{\Sigma_i^k : 1 \leq k \leq n\}$. Here, Σ_i^k is the set of node labels of weisfeiler-Lehman graph at height i of graph G_k.
- f_i^k be the frequency function of weisfeiler-lehman graph at height i of graph G_k.

Without loss of generality, assume that every $\Gamma_i = \{\gamma_{i,k} : 1 \leq k \leq |\Gamma_i|\}$ is ordered.

The weisfeiler-lehman histogram vector of graph G_k with h iteration can be defined as:

$$\boldsymbol{\varphi}_{G_k} = c_1 \oplus c_2 \oplus \ldots \oplus c_h \tag{8.2}$$

Where,

$$c_i = (f_i^k(\gamma_{i,1}), \ldots, f_i^k(\gamma_{i,|\Gamma_i|}))$$

To be convenient, when we need to compute the similarity between frequency vectors in this work, we assume that we have performed the above procedure to make these frequency vectors share the same dimension, and the number of iterations h of Weisfeiler-Lehman kernel in our work is 2.

Definition 8.2.3. Given n directed graph $\{G^k : 1 \leq k \leq n\}$, graph similarity between graph G^i and graph G^j is the cosine similarity between frequency vector $\boldsymbol{\varphi}_{G_i}$ and $\boldsymbol{\varphi}_{G_j}$.

Example 8.2.3:

Let's consider the example in figure 8.3. We now have 2 directed graphs $G1$ and $G2$. The Weisfeiler-Lehman relabelling is performed with one iteration. In this case, we have:

- $\Gamma_0 = \{A, B, C, D, E, F\}$
- $\Gamma_1 = \{A, BA, CA, DBC, DBF, EBD, FA\}$

Therefore, the frequency vectors corresponding to $G1$ and $G2$ are:

- $\boldsymbol{\varphi}_{G1} = \langle 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0 \rangle$
- $\boldsymbol{\varphi}_{G2} = \langle 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1 \rangle$

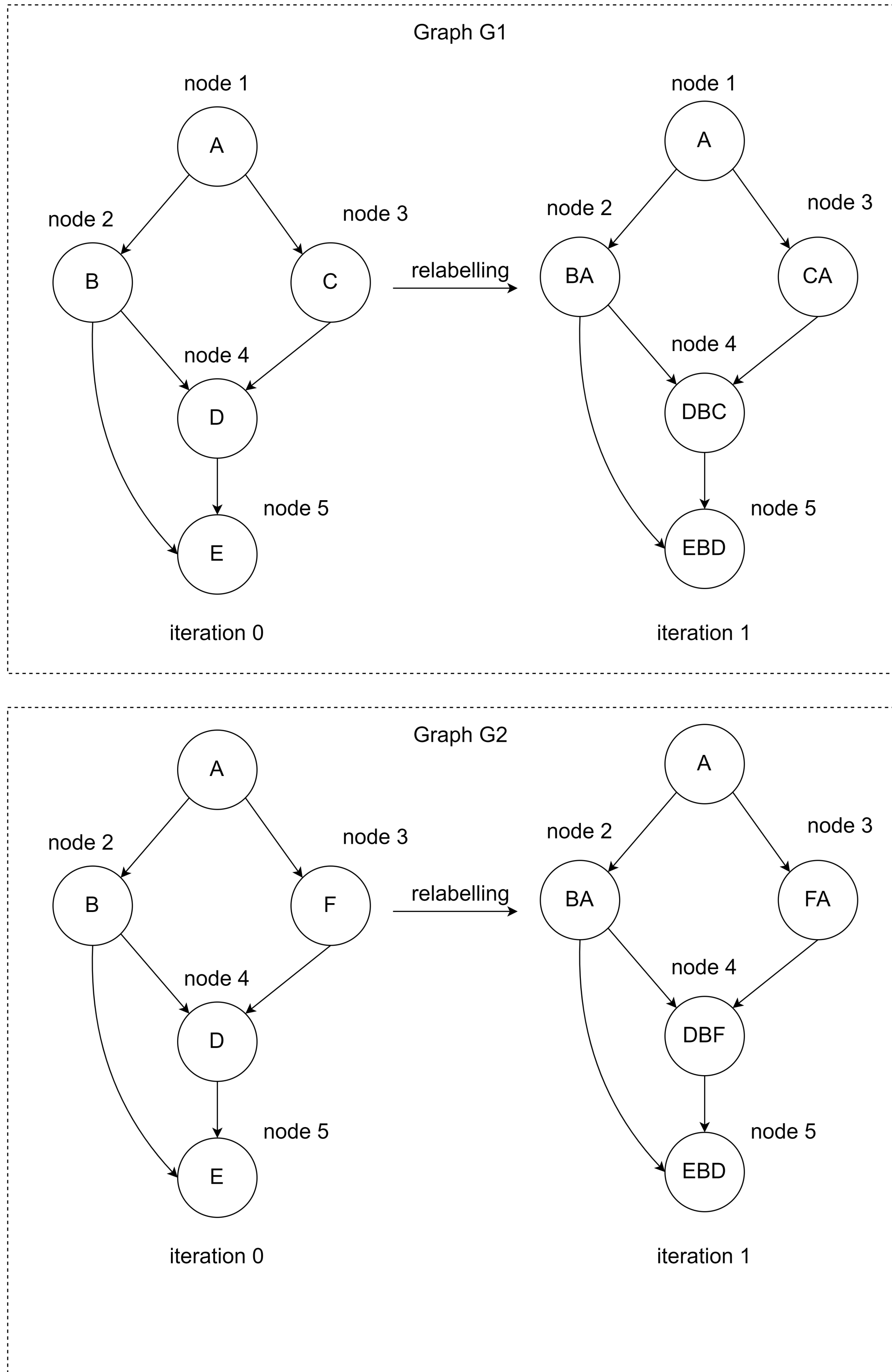

Figure 8.3: An example of frequency feature extraction between two graphs

8.3 Template setup for each packer

8.3.1 Clustering procedure

In the previous sections, we observed that the CFG of unpacking stubs between packed code by the same packer are similar, and WINUPACK packer has two kinds of predecessor unpacking subgraphs. From these observations, it is feasible to find several weifeiler-lehman histogram vectors for each packer.

To reduce the computation cost in weifeiler-lehman histogram vector searching, we want to create several average weifeiler-lehman histogram vector for each packer. From these vectors, we can find the exit of unpacking stub and OEP node later. The process of average weifeiler-lehman histogram vector generation can be described below:

Procedure 8.3.1. The process of average weifeiler-lehman histogram vector generation for a fixed packer includes 5 steps:

- **Step 1**: Generating predecessor graph at the exit of unpacking stubs for each packed code.
- **Step 2**: Computing Weisfeiler-Lehman histogram vector for each graph from step 1.
- **Step 3**: To ensure consistent dimensions, these vectors from Step 2 are converted to align with a uniform dimension.
- **Step 4**: Apply the clustering algorithm (i.e., DBSCAN) to the vectors from Step 3.
- **Step 5**: The Weisfeiler-Lehman histogram vectors within each cluster from Step 4 are averaged and combined with the corresponding end sequence to obtain the template.

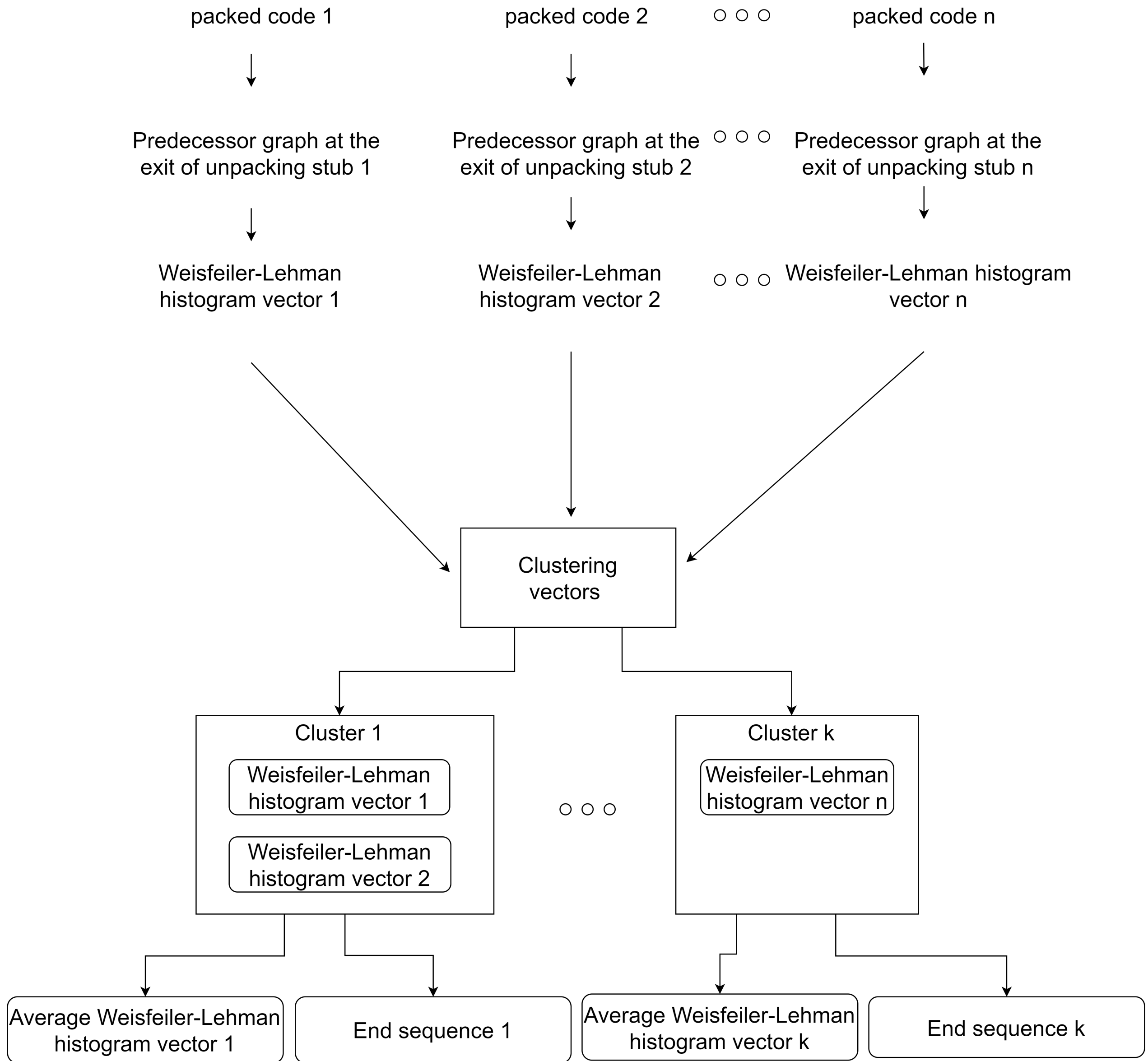

Figure 8.4: The flow of average feature vectors generation

Figure 8.4 show the process of average Weisfeiler-Lehman histogram vectors generation and the sequence of the end of unpacking stub generation for each packer. We assume that there are n packed codes p_i of a packer. In addition, we also assume that we know their original programs because we can use a packer to pack many toy samples to get packed code by this packer. As a result, we will know the exit of the unpacking stub of p_i. Next, we will generate a predecessor unpacking graph of p_i (denoted as Pre^{p_i}). Then, the Weisfeiler-Lehman graph computation will be applied to convert these predecessors graphs into Weisfeiler-Lehman histogram vectors $\mathbf{v}_{Pre^{p_i}}$. Because we need to cluster many frequency vectors of packed codes, these vectors will be converted to the same dimension.

Then, $\varphi_{Pre^{p_i}}$ will be obtained for all packed codes. Next, these vectors will be classified into many groups. Finally, all Weisfeiler-Lehman histogram vectors will be averaged in each group. The detail of the clustering procedure will be described in the 8.3.2, and the process of averaging Weisfeiler-Lehman histogram vectors will be described in 8.3.3.

8.3.2 Clustering Weisfeiler-Lehman histogram vector

After converting Weisfeiler-Lehman histogram vectors into uniform dimensions, we will apply the DBSCAN algorithm to assign these vectors into clusters. This procedure can be illustrated in the figure below:

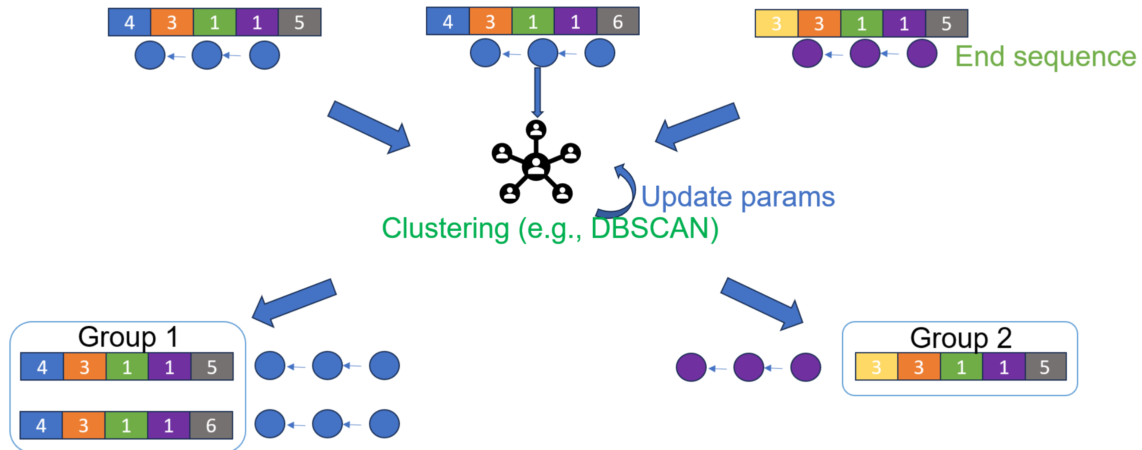

Figure 8.5: DBSCAN clustering procedure

First, we will have many Weisfeiler-Lehman histogram vectors to put into DBSCAN. At this time, the eps parameter will have a default value of 0.05. Next, DBSCAN will cluster these vectors into t groups. We assume vectors in each group are similar and the end sequence of unpacking stubs corresponding to these vectors are the same, and we can average these vectors to obtain an average Weisfeiler-Lehman histogram vector. However, because the result of DBSCAN depends on eps, the condition of the end sequence may not be satisfied. In this case, we decrease the eps value by 0.01 and perform the clustering procedure with the new eps value. The average Weisfeiler-Lehman histogram vector clustering will finish when all end sequences in the same group are consistent. Finally, we will obtain average Weisfeiler-Lehman histogram vectors in each group.

8.3.3 Computing average Weisfeiler-Lehman histogram vector

In this section, we will compute the average Weisfeiler-Lehman histogram vector from the Weisfeiler-Lehman histogram vectors in the same group. First, we also need to convert the dimension of these vectors into the same dimension, then we can take the average. Let's assume we are considering a group d generated from DBSCAN. This group contains

a set of packed codes:

$$C_d = \{p_{d_k} : 1 \leq k \leq |C_d|\}$$

Their corresponding histogram vectors $\{Pre^{p_{d_k}} : 1 \leq k \leq |C_d|\}$ are assigned to graph d.

And, their corresponding Weisfeiler-Lehman histogram vectors are:

$$\{\mathbf{v}_{Pre^{p_{d_k}}} : 1 \leq k \leq |C_d|\}$$

Let's assume these histogram vectors in this form:

$$\mathbf{v}_{Pre^{p_{d_i}}} = ((\omega_1^{d_i}, c_1^{d_i}), \ldots, (\omega^{d_i}_{|\mathbf{v}_{Pre^{p_{d_i}}}|}, c^{d_i}_{|\mathbf{v}_{Pre^{p_{d_i}}}|}))$$

Let:

$$\Omega = \{l : \exists p, q(\omega_q^p = l)\} = (\omega_1, \ldots, \omega_{|\Omega|})$$

Definition 8.3.1. The average Weisfeiler-Lehman histogram vector of a group d is defined as:

$$\mathbf{s}_d = ((\omega_k, a_k), \ldots, (\omega_{|\Omega|}, a_{|\Omega|})) \tag{8.3}$$

Where:

$$a_k = \frac{\sum_{\substack{p,q \\ \omega_q^p = \omega_k}} c_q^p}{|C_d|}$$

Example 8.3.1:

For example, Assuming we have 3 Weisfeiler-Lehman histogram vectors in the same cluster d packed by packer i. Let these vectors be:

- $v_1 = ((A, 1), (B, 3), (C, 5), (D, 1)$
- $v_2 = ((A, 1), (B, 2), (C, 5), (D, 1)$
- $v_3 = ((A, 1), (B, 3), (C, 4))$

So, the average Weisfeiler-Lehman histogram vector of this cluster is:

$$\mathbf{s}_d^i = ((A, 1), (B, 2.67), (C, 4.67), (D, 0.66))$$

After finishing the average Weisfeiler-Lehman histogram vector generation, the generation of end sequences becomes easy. We just take the last k instructions on each cluster in previous sections as the end sequence of the unpacking stub. In this work, we choose the value of k is 5. Now, each packer has obtained its average Weisfeiler-Lehman histogram vector vectors and the end sequence of the unpacking stub, and we have templates for these packers.

Chapter 9

Experiments

This chapter presents our results on packer identification and OEP detection problem. In addition, the information about the dataset and how we confirm an OEP in packed code when the original program is available is also mentioned.

9.1 Experimental environments

Dataset

Our packed code observed and tested in this research is taken from a resource[1]. Besides, TELOCk samples are taken from this repository[2] on Git Hub. Currently, we are focusing on 12 packers which are UPX, FSG, PECOMPACT, PUTITE, WINUPACK, YODA's Crypter, MEW, ASPACK, JDPACK, PACKMAN, and TELOCK. For each packer, not all packed codes we can have their original program. In addition, to verify our method, we must know the OEP of packed codes. To more trust this dataset, we also try to check whether the files in this dataset is packed or not by uploading these file into VirusTotal to confirm the name of the packer.

CFG generation from BE-PUM

Before analyzing packed codes to detect the OEP and the name of the packer, we have to obtain the CFG of packed codes first. Currently, we run BE-PUM on Windows 7 32-bit built on VMware Workstation Pro 17 with Host is Ubuntu 20.04, Processor 13th Gen Intel(R) Core(TM) i9-13900K 3.00 Ghz for 11 packers excepts TELOCk. Samples of TELOCK will be run on a Windows XP environment.

Although BE-PUM can generate precise CFG and precise disassembly of x86 binary code, the time running can be long. Therefore, we set a time limit of running for each packed code as 1 hour. Only packed code whose CFG was generated successfully from BE-PUM will be used in this work.

[1] https://github.com/chesvectain/PackingData
[2] https://github.com/packing-box/dataset-packed-pe

After generating CFG for packed codes, we need to find their corresponding OEPs for template matching and template setup. Therefore, the process to obtain the OEP of a packed code by using its original program will be described in the next section.

OEP acquisition

The procedure to obtain an OEP of packed based on its original program is described in the figure below:

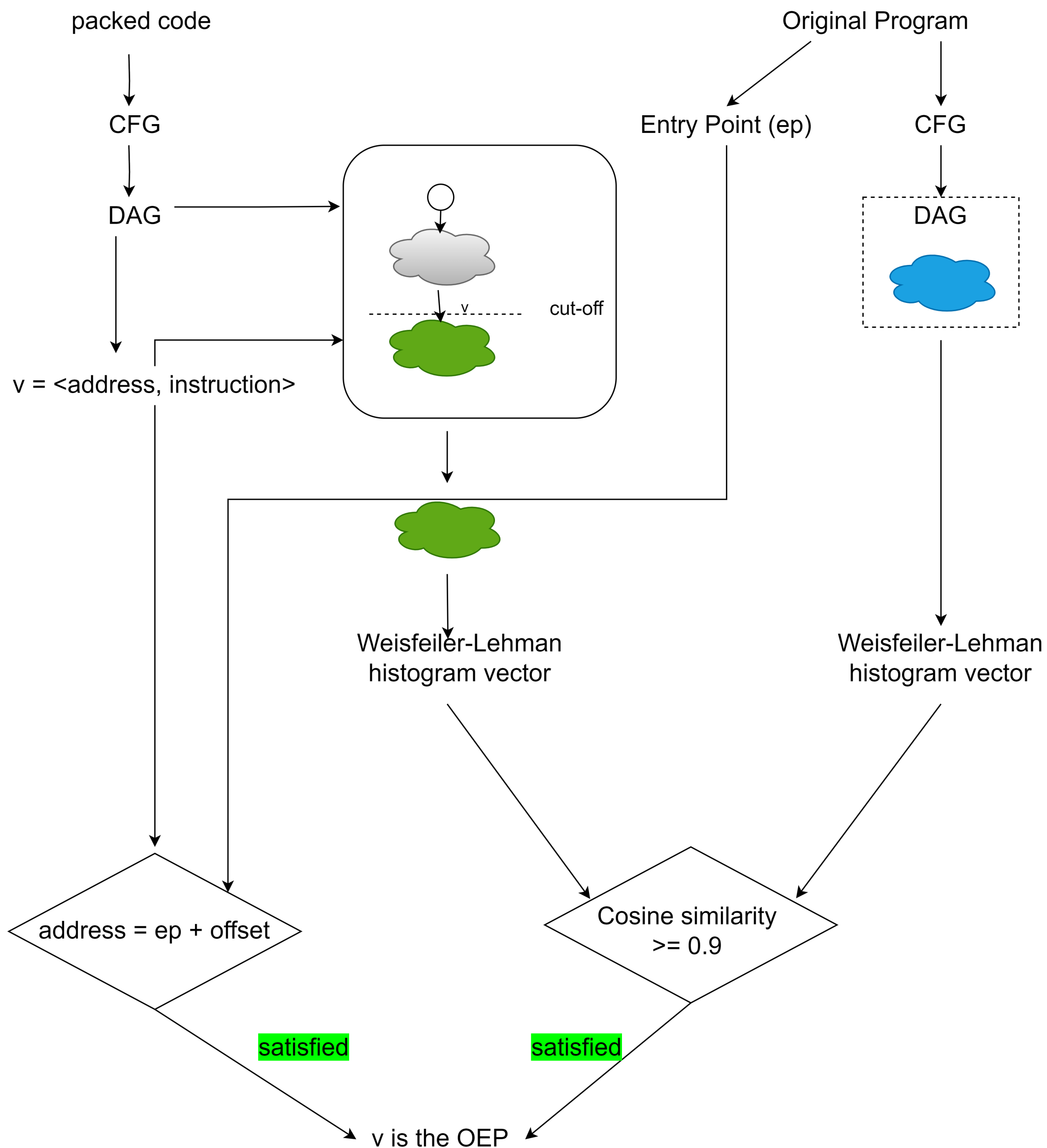

Figure 9.1: OEP acquisition process

There are two conditions in our testing to confirm a $\langle address, instruction \rangle$ is the OEP

of packed code.

1. The address is equal to the entry point of the original program (non-packed program).
2. The graph cut-off from the directed acyclic graph (DAG) of the control flow graph (CFG) of the packed code is similar to the CFG of the original program.

Note that, BE-PUM will stop CFG generation when it identifies the packer name of the packed code, so the CFG of the original program may not be complete. Therefore, we will just extract nodes that have a distance of 20 to the root in both DAG graphs from the packed code and the original program to compare. In addition, we set the threshold for considering two graphs are similar is 0.9. Besides, the entry point of an original program can also be obtained if we have the corresponding packed code of the program by UPX. Because UPX publishes its algorithm, we know the OEP in the packed code by UPX.

After performing the OEP acquisition, we obtained 771 packed codes and their OEPs. The total packed code for each packer is described in the table 9.1.

Packer	Number of samples
UPX	94
ASPACK	75
FSG	83
PECOMPACT	30
MEW	83
YODA's Crypter	82
PETITE	37
WINUPACK	28
MPRESS	86
PACKMAN	87
JDPACK	57
TELOCK	29

Table 9.1: The number of packed codes for each packer

Now, we will split randomly these packed codes into two sets called database and testing set with ratio 1:9. The purpose of these two sets is:

- **Database**: This set contains packed code that we have its original program. So, this set will be used for template setup.
- **Testing set**: In this set, we assume that we don't know the original programs of packed code. Therefore, our method will be applied to detect their OEP and packer names. Then, our results will be verified later by their actual OEP and packer name.

9.2 Packer identification

This section will show our results in the packer identification task. In this experiment, we use 71 samples for obtaining templates, and 700 samples for testing. Besides, we also utilize VirusTotal and PyPackerDetect[3] to identify the packer's name. Particularly, the mechanisms of PyPackerDetect are the use of PEID signatures, Known packer section names, Entrypoint in non-standard section, Threshhold of non-standard sections reached, Low number of imports, and Overlapping entrypoint sections to know the name of the packer.

The table below shows the results of the packer identification task where:

- **Packer**: The name of packer.
- **samples**: The number of tested samples.
- **BE-PUM**: The number of samples predicted the packer's name correctly by BE-PUM.
- **VirusTotal**: The number of samples predicted the packer's name correctly by VirusTotal.
- **PyPackerDetect**: The number of samples predicted the packer's name correctly by PyPackerDetect.
- **Our method**: The number of samples predicted correctly by our method.

Packer	Samples	VirusTotal	PyPackerDetect	BE-PUM	Our method
UPX	85	85	30	84	85
ASPACK	68	68	68	68	68
FSG	75	75	75	75	75
PECOMPACT	27	27	27	27	27
MEW	75	75	75	75	75
YODA's Crypter	74	74	74	62	74
PETITE	34	34	34	34	34
WINUPACK	26	26	26	26	15
MPRESS	78	78	None	78	78
PACKMAN	79	79	79	None	79
JDPACK	52	51	None	None	52
TELOCK	27	27	27	27	27

Table 9.2: The results of the packer identification task

The table above illustrates the number of cases which detect the name of the packer correctly. Among tested packed code, our method detects the packer name of these files

[3]https://github.com/cylance/PyPackerDetect

without error except for WINUPACK. The reason for this error is that this packer has two kinds of templates. However, the database in our setting has only one of two kinds of unpacking graphs. This led to our methods having some errors in WINUPACK, but we can overcome this case by collecting more toy samples packed by WINUPACK. On the other hand, BE-PUM has some wrong packer names in UPX and YODA's Cyber packer. However, this error can come from the supported version of these packers in BE-PUM being different in the test set and the number of tested samples is not large. Finally, the result of BE-PUM for PACKMAN and JDPACK is none because BE-PUM doesn't have the signature sequences of these packers at this moment. Meanwhile, PyPackerDetect does not have the result on MPRESS and JDPACK may come from the different supported versions of packers. Overall, our method yields results that are either better or at least equal to other tools in 12 packers, with the exception of WINPACK, as we explained before.

9.3 OEP detection

In this section, we will represent our results of the OEP detection task. In addition, we also utilize Gunpacker[4] and QuickUnpack[5] which are generic unpacker to detect the OEP.

Figure 9.2 below shows the Graphical user interface of Gunpacker on Windows.

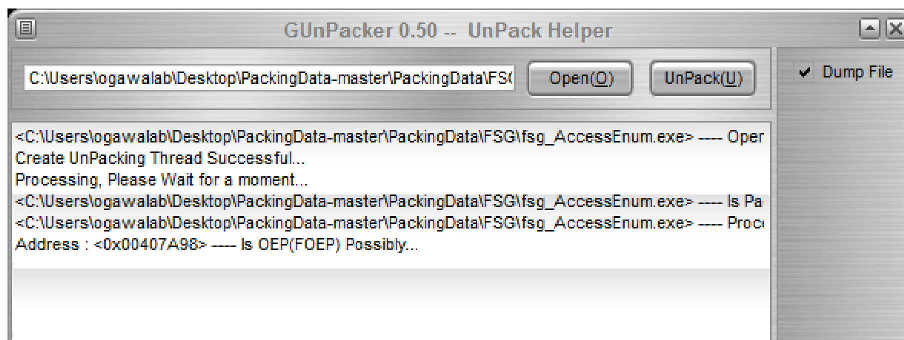

Figure 9.2: GUI of GUnPacker

We can see that GunPacker tools can return the possibility of OEP detected by its algorithm. However, the possibility of OEP from GunPacker is not easy to get automatically from a GUI. Therefore, we have to perform a semi-automated way to get this information. In the GUI of Gunpacker, we have observed that there is an option for a dump file, and we expected that the dump file is an extracted payload from the possibility of OEP in the GUI. So, the entry point of the dump file is equal to the possibility of OEP in the GUI. Therefore, our semi-automated way to get the possibility of OEP from GunPacker includes two steps:

- Step 1: We manually unpack a file by GunPacker from its GUI to get the dump file.

[4]https://webscene.ir/tools/show/GUnPacker-v0.5

[5]https://www.aldeid.com/wiki/QuickUnpack

- Step 2: After obtaining all of the dump files, we use pefile library to read the entry points of these files. Finally, we can obtain the original entry point information from the GUI of Gunpacker.

We also perform the a similar way for QuickUnpacker to obtain OEP detection results. The table below shows the results of the OEP detection task where:

- **Packer**: The name of packer.
- **Samples**: The number of tested samples.
- **Gunpacker**: The number of samples detected OEP correctly by Gunpacker.
- **Our method**: The number of samples detected OEP correctly by our method.

Packer	Samples	Gunpacker	QuickUnpack	Our method
UPX	85	78	78	85
ASPACK	68	56	68	68
FSG	75	70	75	75
PECOMPACT	27	0	8	27
MEW	75	74	8	75
YODA's Crypter	74	73	8	74
PETITE	34	0	8	34
WINUPACK	26	26	4	15
MPRESS	78	0	8	78
PACKMAN	79	79	8	78
JDPACK	52	45	2	52
TELOCK	27	24	1	27

Table 9.3: The results of the OEP detection task

The table above shows the results of the OEP detection task based on graph similarity. Except in one case in PACKMAN and 11 cases in WINUPACK, the OEP of packed codes is correctly detected. From this table, we can observe that our method has a better result than GunPacker and QuickUnpack except in the case of WINUPACK. The reason has also come from we do not have all the templates of WINUPACK.

In addition, we also experience the effect of end-of-unpacking sequences in OEP detection task. In this experiment, we will just only use average histogram histogram vectors to search OEP in packed codes. The two tables below will show our results. The meaning of each column in these tables will be described below:

- **Packer**: The name of packer.
- **samples**: The number of tested samples.

- **Without end sequence**: The number of samples predicted correctly by our method without using the end sequence of the unpacking stub.
- **Using end sequence**: The number of samples predicted correctly by our method using the end sequence of the unpacking stub.

Packer	Samples	Without end sequence	Using end sequence
UPX	85	85	85
ASPACK	68	62	68
FSG	75	75	75
PECOMPACT	27	27	27
MEW	75	75	75
YODA's Crypter	74	73	74
PETITE	34	34	34
WINUPACK	26	15	15
MPRESS	78	73	78
PACKMAN	79	79	79
JDPACK	52	52	52

Table 9.4: The effect of end sequence in packer identification problem

Packer	Examples	Without end sequence	Using end sequence
UPX	85	85	85
ASPACK	68	2	68
FSG	75	75	75
PECOMPACT	27	0	27
MEW	75	67	75
YODA's Crypter	74	72	74
PETITE	34	34	34
WINUPACK	26	15	15
MPRESS	78	69	78
PACKMAN	79	78	78
JDPACK	52	46	52

Table 9.5: The effect of end sequence in OEP detection problem

From these two tables, we can observe that the end sequence of the unpacking stubs has a positive impact on both packer identification and OEP detection problems. Particularly, its impact on the OEP problem is much more substantial than on packer identification. The reason is the answer to the OEP problem needs a specific point in packed codes, so it must be really accurate. Therefore, the end sequence of unpacking stubs can help to remove many incorrect candidates during the process of searching OEP.

9.4 Packer identification and OEP detection on malware samples

In this experiment, we have run 5190 malware samples. Among them:

- There are 1239 samples finished within 1 hour.
- There are 1089 samples, our method detects the sample is not packed by our current support packers.
- There are 1034 samples, BE-PUM detects the sample is not packed.
- Our method detects 150 samples with their packer's name and the end of unpacking stub. The detailed number of samples for each packer is in the table below.

Packer	Number of samples
UPX	80
ASPACK	26
FSG	1
PECOMPACT	27
YODA's Crypter	13
WINUPACK	20
MPRESS	1

Table 9.6: The number of malware samples can detected packer'name and OEP

Although, currently, we cannot verify whether this OEP is correct or not. However, the sequence of the end of the unpacking stub rarely appears by luck, and the similarity of these samples to our average frequency feature vectors is high with more than 0.7. In some sense, our methods worked on these samples.

Chapter 10

Conclusion

10.1 Discussion, Conclusion and Current limitation

10.1.1 Conclusion and current limitation

This thesis proposed an automatic approach to detect the OEP and packer's name of packed codes. In the experiment, 700 packed codes from 12 packers were used to test our methods. Among them, 688 samples have been correctly detected OEP. Furthermore, 5190 malware samples were tested in this work, and there are 1239 samples generated control flow graphs within the time limit of 1 hour. After applying our method, there are 150 samples that have been detected with their OEP and packer's name. We expect that our methodology can extend to many other packers without much effort. In summary, this study has contributed:

- A tool to locate the OEP of packed codes with the presence of their original programs. This can be used to obtain the data for testing and average frequency feature generation for each packer.
- Average Weisfeiler-Lehman histogram vector generation: We have obtained several average Weisfeiler-Lehman histogram vectors for each packer. These vectors represent unpacking stubs. In addition, this module also reduces computation costs for graph similarity computation because the number of average Weisfeiler-Lehman histogram vectors is less than or equal to the number of collected samples of each packer.
- Average Weisfeiler-Lehman histogram vector search: We have proposed an approach to detect the OEP and packer's name based on graph similarity.

Our results show that our hypothesis about the similarity and the consistency of the sequence of the end of the unpacking stub has been worked. From that, we can obtain a signature of the unpacking stub for each packer, and the OEP and the packer's name can be detected correctly. As a result, a lot of human effort can be reduced to unpack a packed code. Finally, malware utilizing packers to hide their malicious behaviours will

be revealed and many further analyses can be conducted to protect machines from this threat.

Advantage:

1. Our method is an automatic approach: Every module in our method can run automatically.

2. Our method is generalized: It can be extended to many other packers just need to collect the packed code for a new packer.

Drawbacks:

1. Dependency on BE-PUM: Because our method needs a precise CFG from BE-PUM and the cost of running time of BE-PUM is heavy, our method also needs quite a long time for the whole process.

2. The repeat of decryption routine: If the packed code can perform the decryption algorithm of unpacking stubs several times before it transfers the control flow to OEP, our method cannot handle this case at this moment.

10.1.2 Discussion

Failed on OEP acquisition using the original program

During the preparation of the OEP for each packed code, we noticed that there are some packed codes we cannot find its OEP although we have its original program. One of the reasons is BE-PUM will terminate its process when it concludes with the packer name, so the CFG of the unpacked payload is not generated in these cases. Therefore, we cannot verify whether a point is OEP or not based on the similarity between the cut-off CFG from a specific point in packed code and the CFG of the original program.

Incorrect OEP detection on PACKMAN

Currently, we have investigated the only wrong case in PACKMAN, and we observed that there is an instruction `jump start` during the execution of the packed code. After that, the unpacking process was repeated one more time before the control flow was transferred to the original entry point. Therefore, although the sequence of the end of the unpacking stub has been matched and the similarity between the frequency feature vector at the incorrect node and the average frequency feature vector of PACKMAN is high, the result is still not correct.

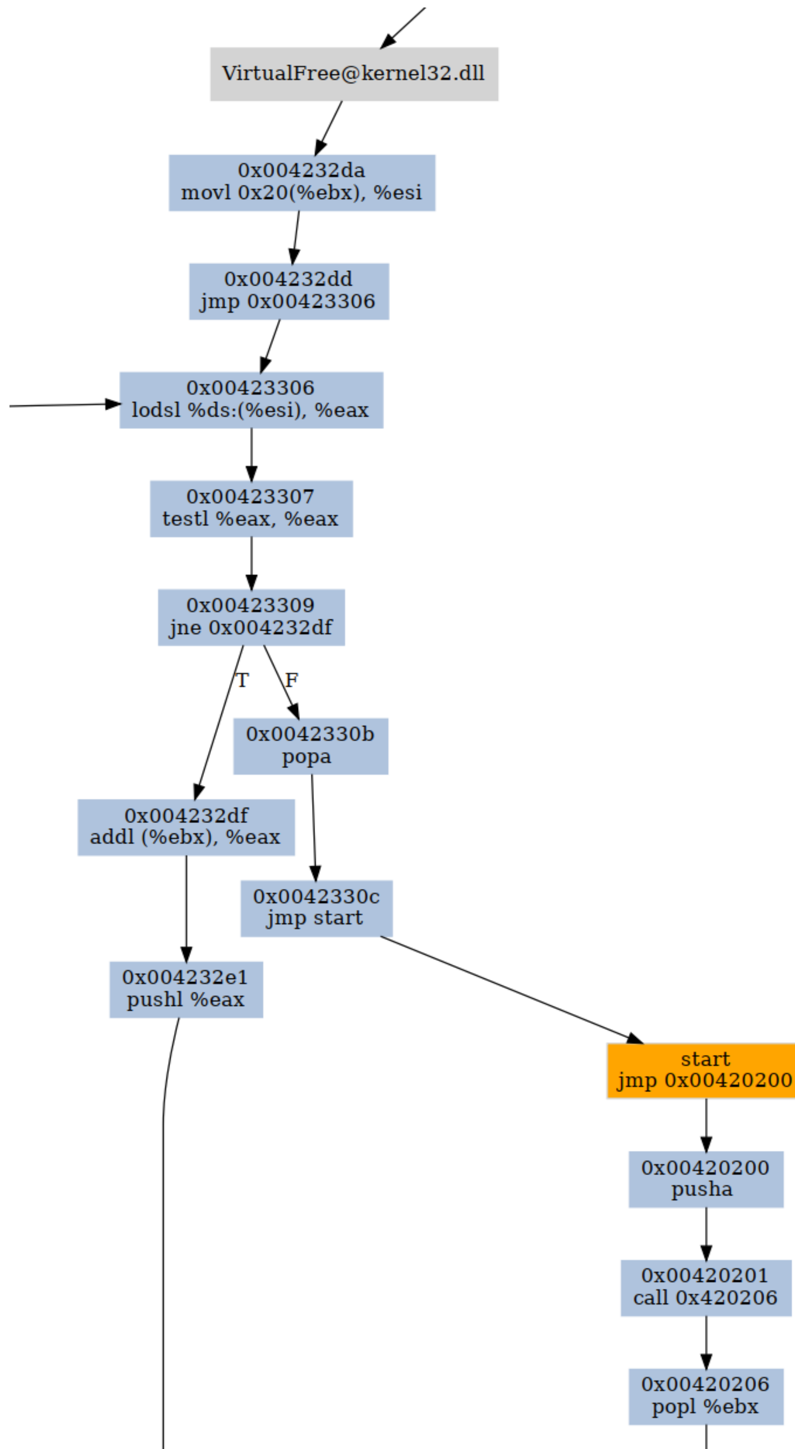

Figure 10.1: a jump start in PACKMAN

Figure 10.1 shows the incorrect the exit of unpacking stub in FileTypesMan.exe packed by PACKMAN. In this case, our method detects `0x004231f4: jmp 0x00420200` (orange node) as the exit of the unpacking stub, but the actual result is

`0x00420200: jmp 0x4109f0`.

10.2 Future works

Currently, our method can detect the OEP and identify the packer's name, but the repeat of the decryption routine can lead to an incorrect answer in our method. Therefore, we intend to continue this work to expand the capacity of our method, make it able to cover more packers, as well as to deal with the repeat of the decryption routine.

1. Extend to more packers: There are many other packers on the Internet, some of the most complex packers like themida are not supported in our work. Therefore, we need to update these packers in our work to make our method stronger.

2. Improve incorrect answer: Our method still yields incorrect results, indicating that there is room for improvement to enhance its capacity, especially in the case of a repeat decryption algorithm.

3. Direct detect OEP: Currently, our method detects the exit of the unpacking stub first, and then the OEP node will be detected later because the OEP node is the successor of the exit of the unpacking stub. Although we observed that the OEP node has only one accessor on the CFG, we aim to enhance the generality of our method by directly detecting OEP, eliminating the need to detect it through an end-of-unpacking node.

4. Dealing with unknown packer: Unknow packers referred to packers that we can access to it (i.e., we don't have the packer program to pack samples). This is really a challenging problem. In the future, we intend to extend our method to deal with it.

In the future, our plan is:

- Collecting more samples of packed code to support many other packers.

- Propose more conditions to confirm whether a point is OEP or not.

- Changing our method to directly detect OEP by marking the OEP node as a special node (e.g, \$), then instead of obtaining the average Weisfeiler-Lehman histogram vectors from the predecessor graph, we obtain these vectors from the predecessor graphs with their sink node is \$.

- Try to observe packed codes with the hidden packer information.

Bibliography

[1] Z. Zeng, A. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: On approximating graph edit distance.," *PVLDB*, vol. 2, pp. 25–36, Jan. 2009.

[2] P. Alimonti and V. Kann, "Hardness of approximating problems on cubic graphs.," vol. 1203, Jan. 1997, pp. 288–298.

[3] M. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," *WORM'07 - Proceedings of the 2007 ACM Workshop on Recurring Malcode*, Nov. 2007. DOI: `10.1145/1314389.1314399`.

[4] G. Jeong, E. Choo, J. Lee, M. Bat-Erdene, and H. Lee, "Generic unpacking using entropy analysis," in *2010 5th International Conference on Malicious and Unwanted Software*, 2010, pp. 98–105. DOI: `10.1109/MALWARE.2010.5665789`.

[5] R. Isawa, M. Kamizono, and D. Inoue, "Generic unpacking method based on detecting original entry point," vol. 8226, Nov. 2013, pp. 593–600, ISBN: 978-3-642-42053-5. DOI: `10.1007/978-3-642-42054-2_74`.

[6] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," Jan. 2007, pp. 289–300. DOI: `10.1109/ACSAC.2006.38`.

[7] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, pp. 431–441. DOI: `10.1109/ACSAC.2007.15`.

[8] S. D'Alessio and S. Mariani, "Pindemonium: A dbi-based generic unpacker for windows executables," in *BlackHat*, 2016, pp. 1–56.

[9] G.-M. Kim, J. Park, Y.-H. Jang, and Y. Park, "Efficient automatic original entry point detection," *J. Inf. Sci. Eng.*, vol. 35, pp. 887–901, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:53549413`.

[10] NtQuery, *Scylla – x64/x86 imports reconstruction*, `https://github.com/NtQuery/Scylla`.

[11] B. Cheng, J. Ming, E. A. Leal, *et al.*, "Obfuscation-resilient executable payload extraction from packed malware," in *USENIX Security Symposium*, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:235246272`.

[12] M.-J. Choi, J. Bang, J. Kim, H. Kim, and Y.-S. Moon, "All-in-one framework for detection, unpacking, and verification for malware analysis," *Security and Communication Networks*, vol. 2019, pp. 1–16, Oct. 2019. DOI: `10.1155/2019/5278137`.

[13] A. V. Phan, M. L. Nguyen, Y. L. H. Nguyen, and L. T. Bui, "Dgcnn: A convolutional neural network over large-scale labeled graphs," *Neural networks : the official journal of the International Neural Network Society*, vol. 108, pp. 533–543, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:53566112`.

[14] C.-H. Bertrand Van Ouytsel and A. Legay, "Malware analysis withnbsp;symbolic execution andnbsp;graph kernel," in *Secure IT Systems: 27th Nordic Conference, NordSec 2022, Reykjavic, Iceland, November 30–December 2, 2022, Proceedings*, Reykjavic, Iceland: Springer-Verlag, 2023, pp. 292–310, ISBN: 978-3-031-22294-8. DOI: `10.1007/978-3-031-22295-5_16`. [Online]. Available: `https://doi.org/10.1007/978-3-031-22295-5_16`.

[15] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *Journal of Machine Learning Research*, vol. 12, pp. 2539–2561, 2011.

[16] *Dbscan sample points*, `https://www.geeksforgeeks.org/dbscan-clustering-in-ml-density-based-clustering/`, Accessed: 2023-08-02.

[17] *An sample of different kind of points*, `https://en.wikipedia.org/wiki/DBSCAN`, Accessed: 2023-08-02.

[18] A. Kleymenov and A. Thabet, *Mastering Malware Analysis: The Complete Malware Analyst's Guide to Combating Malicious Software, APT, Cybercrime, and IoT Attacks*. Packt Publishing, 2019, ISBN: 9781789610789. [Online]. Available: `https://books.google.co.jp/books?id=Nj4txgEACAAJ`.

[19] *Pe structure*, `https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/basic-packers-easy-as-pie/`, Accessed: 2023-08-02.

[20] *User mode and kernel mode in windows*, `https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode`, Accessed: 2023-08-02.

[21] A. Silberschatz and P. Galvin, *Operating System Concepts, 4th edition.* Addison-Wesley, Jan. 3, 2002, ISBN: 0-201-50480-4.

[22] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[23] V. V. Anh, "Formal semantics extraction from natural language specifications for arm," Master's Thesis, School of Information Science, JAIST, December 2018.

[24] M. Nguyen, M. Ogawa, and T. Quan, "Packer identification based on metadata signature," in *The 7th Software Security, Protection, and Reverse Engineering Workshop (SSPREW-7), ACM*, 2017.

[25] N. M. Hai, M. Ogawa, and Q. T. Tho, "Obfuscation code localization based on cfg generation of malware," in *FPS*, 2015.

[26] J. Kinder, F. Zuleger, and H. Veith, “An abstract interpretation-based framework for control flow reconstruction from binaries,” in *Lecture Notes in Computer Science*, vol. Year, Springer Berlin Heidelberg, 2009, pp. 214–228.

[27] L. de Moura and N. Bjørner, “Z3: An efficient smt solver,” vol. 4963, Apr. 2008, pp. 337–340.

[28] L. Vinh, “Automatic stub generation from natural language description,” Master Thesis, JAIST, August, 2016.

[29] W. Leitner, S. Wahl, S. Popkin, J. Gaertner, T. Åkerstedt, and S. Folkard, “Ras representation analysis software,” Jan. 2003.

[30] M. S. Hecht and J. D. Ullman, “Flow graph reducibility,” in *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC ’72, Denver, Colorado, USA: Association for Computing Machinery, 1972, pp. 238–250, ISBN: 9781450374576.

[31] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in *Knowledge Discovery and Data Mining*, 1996. [Online]. Available: `https://api.semanticscholar.org/CorpusID:355163`.