YAŞAR UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

PhD THESIS

IMAGING AND EVALUATING THE MEMORY ACCESS FOR MALWARE

ÇAĞATAY YÜCEL

THESIS ADVISOR: ASSOC.PROF. AHMET HASAN KOLTUKSUZ, Ph.D.

DOCTOR OF PHILOSOPHY

IN

COMPUTER ENGINEERING

PRESENTATION DATE: 29.08.2019

BORNOVA/IZMIR
AUGUST 2019

We certify that, as the jury, we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of the Doctor of Philosophy.

Jury Members:	**Signature:**
Assoc. Prof. Ahmet Hasan KOLTUKSUZ, Ph.D. Yaşar University	....
Prof. Şaban EREN, Ph.D. Yaşar University	....
Assoc. Prof. Tuğkan TUĞLULAR, Ph.D. Izmir Institute of Technology	....
Assoc. Prof. Murat Osman ÜNALIR, Ph.D. Ege University	....
Asst. Prof. Mutlu BEYAZIT, Ph.D. Yaşar University	....

Prof. Cüneyt GÜZELİŞ, Ph.D.
Director of the Graduate School

ABSTRACT

IMAGING AND EVALUATING THE MEMORY ACCESS FOR MALWARE

YÜCEL, Çağatay
Ph.D., Computer Engineering
Advisor: Assoc. Prof. Ahmet Hasan Koltuksuz
August 2019

Malware analysis is a forensic process. After infection and the damage represented itself with the full scale, then the analysis of the attack, the structure of the executable and the aim of the malware can be discovered. These discoveries are converted into analysis reports and malware signatures and shared among antivirus databases and threat intelligence exchange platforms. This highly valuable information is then utilized in the detection mechanisms in order to prevent further dissemination and infections of malware. The types of analysis of the malware sample in this process can be grouped into two categories: static analysis and dynamic analysis. In static analysis, the executable file is reverted to the source code through disassemblers and reverse engineering software and analyzed whereas dynamic analysis includes running the sample in an isolated environment and analyzing its behavior. Both static and dynamic analysis have limitations such as packing, obfuscation, dead code insertion, sandbox detection, and anti-debugging techniques. Memory operations, on the other hand, are not possible to hide by these limitations and inevitable for any software since the inventions of the computational models. Therefore, in this research, memory operations and access patterns for the malicious acts are examined and a contribution of a novel approach for extracting of memory access images is presented. In addition to extraction, methods of how these images can be used for detection and comparison is introduced through an image comparison technique.

Key Words: Malware Analysis, Malware Imaging, Memory Analysis, Dynamical Binary Analysis, Memory Operations Analysis.

ÖZ

ZARARLI YAZILIMLAR İÇİN BELLEK ERİŞİMLERİNİN GÖRÜNTÜLENMESİ VE DEĞERLENDİRİLMESİ

YÜCEL, Çağatay
Doktora Tezi, Bilgisayar Mühendisliği Bölümü
Danışman: Doç. Dr. Ahmet Hasan Koltuksuz
Ağustos 2019

Kötü amaçlı yazılım analizi adli bilişsel bir süreçtir. Zararlı yazılım; başarılı bir şekilde hedef bilgisayara bulaştıktan, amaçladığı zarar hedef bilgisayarda oluştuktan ve yazılım kendini tam ölçekte gösterdikten sonra ancak çalıştırılabilir dosyanın hedefi ve yapısı gerçek anlamda anlaşılabilir. Zararlı yazılım analizi ile elde edilen bu bulgular kötü amaçlı yazılım imzalarına dönüştürülmekte; antivirüs veritabanları ve tehdit istihbarat değişim platformları arasında paylaşılmaktadır. Bu çok değerli bilgiler daha sonra kötü amaçlı yazılımların daha fazla yayılmasını önlemek amacıyla saptama/önleme mekanizmalarında kullanılır. Bu süreçte kötü amaçlı yazılım örneğinin analizi iki kategoriye ayrılır: statik analiz ve dinamik analiz. Statik analizde çalıştırılabilir dosya, tersine mühendislik yazılımları aracılığıyla kaynak koduna geri döndürülüp analiz edilirken, dinamik analiz, çalıştırılabilir dosyanın dışarıya kapalı bir ortamda çalıştırılmasını ve davranışlarının analizini içerir. Hem statik hem de dinamik analiz, paketleme, perdeleme, ölü kod ekleme, sanal makinenin algılanması ve hata ayıklama önleme teknikleri gibi analiz önleme teknikleriyle sınırlıdır. Öte yandan bellek üzerinden gerçekleştirilen analiz işlemleri bu sınırlamalarla gizlenemez ve bilgisayar sistemlerinin modellerinin icadından bu yana herhangi bir yazılım için kaçınılmazdır. Bu nedenle, bu araştırmada, kötü niyetli eylemler için bellek işlemleri ve bellek erişim örüntüleri incelenmiş, bellek erişim görüntülerinin çıkarılması için yeni bir yaklaşımın katkısı litaretüre sunulmuştur. Bu çıkarma yöntemine ek olarak, bu görüntülerin tespiti ve karşılaştırma için nasıl kullanılabileceği görüntü karşılaştırma tekniği ile ortaya konulmuştur.

Anahtar Kelimeler: Zararlı Yazılım Analizi, Zararlı Yazılım Görüntüleme, Bellek Analizi, Dinamik Çalıştırılabilir Dosya Analizi, Bellek Operasyonları Analizi.

ACKNOWLEDGEMENTS

First and foremost, I want to thank my advisor, Prof. Ahmet Hasan KOLTUKSUZ, not only for the valuable insights and vision of his, also for the opportunity, support, patience, and guidance he has given me since I was an undergraduate. Without his vision this work wouldn't have been written, and his enlightenment on many aspects had kept me on the right track many times.

This work is a product of the Cyber Security Research Laboratory of Yaşar University. For this lab to exist, Prof. Koltuksuz had put years of knowledge and his significant expertise. I want to express my sincere gratitude to him once more; since throughout my computer security research career in this lab, not a single day went by without excitement.

My thesis committee guided me through all these years. Very special thanks for my thesis committee members, Prof. Tuğkan TUĞLULAR and Prof. Mutlu BEYAZIT, for their valuable insights, guidance, and support throughout my thesis. Prof. Tuğlular has always shown me the angles and perspectives, which makes this work better in many ways, many thanks to him. Prof. Beyazıt had supported me not only on the scientific merits of this thesis, but also he had provided me lots of support as well throughout the writing and completion of this thesis.

I would like to thank Jury Members, Prof. Şaban EREN and Prof. Murat Osman ÜNALIR. With the guidance and help of Prof. Eren, this work had sit on solid grounds scientifically and statistically. My sincere gratitude to Prof. Ünalır for his advice on Data Analytics and Big Data.

I want to thank Prof. Timur KÖSE on his help on Classification and Machine Learning subjects. Special thanks to Prof. Hüseyin HIŞIL for his moral support and guidance for all these years. I would also like to thank my lab mates Mr. Anas M'uaz Kademi and Mr Murat ÖDEMİŞ. I also would like to thank my roommate Dr. Gökhan DEMİRKIRAN and Dr. Mustafa BÜYÜKKEÇECİ for their support on MATLAB. Many thanks to my student Mr. Armağan YILDIRAK for his support in coding and automation.

Very special thanks to my beloved wife, Mrs Özgün YÜCEL for her loving support and extraordinary patience for all of these years. And finally, last but by no means least, I would like to thank my mother Ms Muazzez YÜCEL and my brother Mr. Çağan Selçuk YÜCEL for supporting me spiritually and patiently throughout writing of this thesis.

Çağatay Yücel
İzmir, 2019

TEXT OF OATH

I declare and honestly confirm that my study, titled "IMAGING AND EVALUATING THE MEMORY ACCESS FOR MALWARE" and presented as a Ph.D. Thesis, has been written without applying to any assistance inconsistent with scientific ethics and traditions. I declare, to the best of my knowledge and belief, that all content and ideas drawn directly or indirectly from external sources are indicated in the text and listed in the list of references.

Çağatay Yücel

Signature

....

September 16, 2019

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGEMENTS ix

TEXT OF OATH xi

TABLE OF CONTENTS xiii

LIST OF FIGURES xvii

LIST OF TABLES xix

ABBREVIATIONS xxi

CHAPTER 1 INTRODUCTION 1
1.1. HISTORY OF MALWARE 3
1.2. PROBLEM DEFINITION, MOTIVATION, AND AIM 7
1.2.1. OBFUSCATION 8
1.2.2. OLIGOMORPHIC MALWARE 8
1.2.3. POLYMORPHIC MALWARE 8
1.2.4. METAMORPHIC MALWARE 8
1.2.5. DEAD CODE INSERTION 9
1.2.6. ANTI-VM AND VIRTUAL INTERFACE DETECTION 9
1.2.7. ANTI-DEBUGGING 9
1.2.8. EXECUTION STALLING 9
1.2.9. LOGIC BOMBS – EXTENDED SLEEPS 10
1.2.10. NATIVE DLL CODING 10
1.2.11. PROCESS INJECTION 10
1.3. THESIS OUTLINE 13

CHAPTER 2 BACKGROUND: MALWARE STRUCTURE, FAMILIES AND CONSTRUCTION 14

2.1. GENERAL LAYOUT OF PE FILES 14
2.2. STRUCTURE OF A VIRUS 16
2.3. STRUCTURE OF A WORM 20
2.4. STRUCTURE FOR A TROJAN 22
2.5. MALWARE FAMILIES 23
2.6. CONCLUSION 25

CHAPTER 3 STATIC ANALYSIS OF MALWARE 27
3.1. STATIC ANALYSIS TECHNIQUES 27
3.2. LITERATURE ON STATIC MALWARE ANALYSIS AND MACHINE LEARNING METHODS 34
3.3. TOOL SUPPORT FOR STATIC MALWARE ANALYSIS AND VISUALIZATIONS 37
3.4. A GRAPH DATABASE APPROACH FOR STATIC ANALYSIS 37
3.5. VISUALIZATIONS OF STATIC ANALYSIS 39
3.6. LIMITATIONS OF STATIC ANALYSIS 41
3.7. CONVOLUTIONAL NEURAL NETWORKS (CNN) ON STATIC IMAGES 45
3.8. CONCLUSION OF CHAPTER 3 45

CHAPTER 4 DYNAMIC ANALYSIS OF MALWARE 47
4.1. DYNAMIC ANALYSIS TECHNIQUES 47
4.2. THE LITERATURE OF DYNAMIC MALWARE ANALYSIS 49
4.3. TOOL SUPPORT FOR DYNAMIC MALWARE ANALYSIS. 53
4.4. A SHOWCASE OF DYNAMIC ANALYSIS WITH INTEL PIN TOOL. 54
4.5. CONCLUSION AND DISCUSSION OF CHAPTER 4. 59

CHAPTER 5 MALWARE MEMORY FORENSICS 61
5.1. MEMORY FORENSICS 61
5.2. MALWARE OPERATIONS ON MEMORY 63
5.2.1. PACKING AND COMPRESSION 63
5.2.2. CODE INJECTION 65
5.2.3. DLL INJECTION AND REFLECTIVE DLL INJECTİON 66
5.2.4. PROCESS HOLLOWING 67
5.3. MANUAL ANALYSIS DETECTION OF MEMORY OPERATIONS 69
5.4. THE LITERATURE ON AUTOMATED MALWARE DETECTION WITH MEMORY ANALYSIS 70
5.5. CONCLUSION 71

CHAPTER 6 MALWARE MEMORY IMAGING AND EVALUATION 73

6.1. MOTIVATION 73
6.2. INSTRUMENTING THE MEMORY OPERATIONS 75
6.2.1. MEMORY LAYOUT 75
6.2.2. FLOWCHART OF THE METHODOLOGY 76
6.2.3. ALGORITHM FOR THE PIN TOOL 78
6.2.4. A MEMORY IMAGE EXTRACTION EXAMPLE 79
6.3. GRAPH DATABASE MODEL FOR MEMORY ACCESS AND A SHOWCASE... 80
6.4. STRUCTURED SIMILARITY INDEX MEASURE (SSIM) 83
6.5. CONCLUSION OF CHAPTER 6 84
CHAPTER 7 TESTS AND RESULTS 85
7.1. SOFTWARE SUPPORT 85
7.2. MALWARE DATASET 85
7.3. MALWARE MEMORY PATTERNS 86
7.4. MALWARE FAMILIES 95
7.5. DISCUSSION 95
CHAPTER 8 CONCLUSION 99
8.1. PROBLEM DEFINITION RE-VISITED 99
8.2. CONTRIBUTIONS 99
8.3. DISCUSSIONS AND CHALLENGES 100
8.4. FUTURE WORK 101
REFERENCES 103

LIST OF FIGURES

Figure 1.1- Problem Tree of Signature Generation with Malware Analysis. 11

Figure 2.1. Structure of PE file Format (Pietrek, 2011)... 15

Figure 2.2. (a) The virus inserts itself into another executable. (b) The virus alters the entry point in the header. (c) New entry point jumps to decryptor of the static encrypted virus body. (d) The decrypted virus body executes... 19

Figure 2.3. A Flow Chart of a Computer Trojan. 22

Figure 3.1 CFG of Virus Xpaj.C with hash 32

Figure 3.2 Graph Database Output of a CFG of the Virus Xpaj.C with hash....................... 33

Figure 3.3 Function-Imported and API Functions illustrated Together of the same executable in Figure 2. 33

Figure 3.4 - The schema of the Graph Database.. 37

Figure 3.5- Insertion process of the Static Analysis Results.. 39

Figure 3.6 - The visualization of the malware sample with Sha1 33e8e894297e0f94c5df36cb4e5b3ee68662ceff (a) An overview of the malware (b) Hovering on a function node (c) Hovering on an API node (d) Hovering on an edge between functions... 40

Figure 3.7 - 3d model of the same sample in Figure 6.. 41

Figure 3.8 The visualization of the malware samples with Sha1 0f241d84aa44034c924197d3bce94faa07811f35, f53e68832af99cf553471cf87cc5da332c695659, ada9efdf8dee612599377f6ade3e78e06d4069f4, a9accc4fe6cd45b9a54c25a1447ed74cc61d5675 respectively, showing (a) dead code insertion (b) encryption engine (c) obfuscation (d) packing.......................... 43

Figure 4.1 Architecture of Pin Tool... 53

Figure 4.2. Static Analysis Graphs of the samples. 55

Figure 4.3. Function Hit Traces(a), (b) and Function Traces (c), (d) of Ramnit v1 and v2 respectively. 56

Figure 4.4. DLL Addresses and Sequence graphs of the Samples. (a) Ramnit v1 (b) Ramnit v2.. 58

Figure 4.5. Instruction Pointer Traces of the Samples. (a) Ramnit v1 (b) Ramnit v2. 59

Figure 5.1. Illustration of packing of an executable. 65

Figure 5.2. Illustration of Code Injection. 66

Figure 5.3. Remote DLL Injection using LoadLibrary method..67

Figure 5.4. Illustration of Process Hollowing..69

Figure 6.1. Memory Layout of a PE Format. 76

Figure 6.2. Flowchart of the Proposed Methodology. 77

Figure 6.3. The Extracted image of one of the images from the Keygen Trojan family. The md5 of the malware is 5fe2aebb2fe4abe503d297c318a37a62. 80

Figure 6.4. Memory Accesses of a Sample connected with Euclidean Distances.................81

Figure 6.5. Consecutive writes of the malicious sample. 82

Figure 6.6. The nodes with Euclidean distances are smaller than 100 are merged in this example. An apparent clustering of the memory access can be seen from this figure. 83

Figure 7.1 UPX patterns for the samples respectively, (a), (b) portable benign executables, (c), (d), (e) Trojans from Keygen Family, (f) a trojan from Sub7 family. The average similarity ratio for these patterns is 0.6724712. 88

Figure 7.2. Observed Packing and Self-decrypting patterns. 90

Figure 7.3. Process Injection Indications on Various Patterns. 91

Figure 7.4 Ransomware Patterns with the pairwise similarity average of 0.822487.............93

Figure 7.5. Viruses and Infectors Fingerprints. For the family samples of Rex Virus given above, the similarity rate is 0.999994, and for the Autorun samples, the same ratio is 0.947181...94

Figure 7.6. Pairwise Similarity ratio average values for all the families in our dataset. 95

Figure 7.7 - Heatmap of the Malware Dataset Similarities. 97

LIST OF TABLES

Table 3.1 Suspicious strings of W32.eternalrocks creating and installing an onion network node in the Microsoft updates folder.. 29

Table 3.2 Suspicious strings of W32.carberp extracted possible signature filenames. (sha256:4297ad0f5bb72616337d88f14c07a6c6d6e0c93d2a9bb5eaa7e09219556aafdb) 30

Table 3.3 Imports table extracted from the symbol table of malware Win32.Emotet. (sha256:6393fe8dd4721190f240e22feeb769675b6194a70cabd5a415c2364686a9089c)... 30

Table 3.4. Comparison of Researches that utilizes Static Analysis Techniques................... 36

Table 3.5 - Properties of the Nodes of the Graph Database... 38

Table 4.1. Comparison of Researches that utilizes Dynamic Analysis Techniques. 52

Table 4.2. Sha1 hashes and compilation information of the malware samples 54

Table 6.1. Linked list of memory images of the malware Keygen 79

Table 6.2. Sample Details for Example Analysis. 80

Table 7.1. The distribution of the malware samples in our dataset. 86

ABBREVIATIONS

API	Application Programming Interface
APT	Advanced Persistent Threat
ASLR	Address Space Layout Randomization
BBS	Bulletin Board System
CERT	Cyber Emergency Response Team
CFG	Control Flow Graph
CNN	Convolutional Neural Networks
COFF	Common Object File Format
DBI	Dynamic Binary Instrumentation
DLL	Dynamically Loaded Library
DTA	Dynamic Taint Analysis
EPO	Entry Point Obscuring
FN	False Negative
FP	False Positive
FTP	File Transfer Protocol
IDPS	Intrusion Detection Prevention System
IP	Internet Protocol
JIT	Just-in-Time
KNN	K Nearest Neighbor

NOP	No Operation
OOA	Objective Oriented Association
PDF	Portable Document File
PE	Portable Executable
PEB	Process Environment Block
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RF	Random Forest
SSIM	Structured Similarity Index Measure
SVM	Support Vector Machine
TAN	Tree Augmented Naive Bayes
TF-IDF	Term Frequency - Inverse Document Frequency
TN	True Negative
TP	True Positive
UPX	Ultimate Packer for Executables
VAD	Virtual Address Descriptor
VCL	Virus Creation Laboratory
VM	Virtual Machine

CHAPTER 1
INTRODUCTION

A malware, short for malicious software is a software to accomplish harmful, unwanted, and illegal tasks on a computer system. There are many types of malware as of today; viruses, worms, trojans, spyware, ransomware, botnets and so on. Malware analysis is the forensics process performed to reveal the aim, structure, characteristics, damages, and impacts of malicious software.

The ever-evolving race between malware developers and cybersecurity professionals is an arms race. Just as in any other regular software development community, malware developing community evolves as well. There are newly developed techniques, new anti-detection mechanisms and zero-day vulnerabilities that have come to light every day. These improvements lead to trends between malware developers and these crafts are shared on the market (usually on the darknet, which is the common name for the hidden parts of the Internet that is not routed with the general routing algorithms). Malware also evolves through a process called anonymization: when a new feature for developing malware is present, it is shared with this community of malware developers for utilizations and modifications. Hence, every malware developer modifies the code for their selves. Within days, the same feature is integrated into several other malware or altered versions of the same malware are accustomed to harm other computer systems and networks. Therefore, this process of anonymization provides a large domain for a malware to evolve, whilst keeping the original version unknown (Ding, Fung, & Charland, 2016).

Until now, anti-malware techniques that are based on signature matching have been successful in known types of malware. A signature is a predefined pattern of the malicious software extracted by the analysis and scanning of the machine code of the software (Gandotra, Bansal, & Sofat, 2014). There are two types of malware analysis: Static Analysis and Dynamic Analysis. Static analysis has the means to analyze the binary of the malware through reverse-engineering and disassembling. Specific strings and patterns are extracted from the reversed code and shared as signatures of the

sample. In dynamic analysis, the executable is run in a preset secure environment (called sandbox), and the behavior of the malicious is observed through debugging and hooking on the network communications, system calls, memory accesses, and disk operations.

Malware developers have integrated several countermeasures to their software to evade from static analysis and signature detection. These are mainly; obfuscation, packing, metamorphism, polymorphism, and encryption techniques. Obfuscation is a process applied to the source code so that it is not readable by humans. Several methods are involved in the process of obfuscation; dead code insertion, instruction substitution, register substitution, function reordering (Farhadi et al., 2015). Dead code insertion is to insert code blocks and random instructions to the original code that are never going to be executed. This technique leads to loss of performance of the malware; however, as the source code is modified, the signature detection can be evaded. Another technique is the instruction substitution. In this technique, the instructions that can be used interchangeably such as jump instructions are replaced. Register substitution similarly substitutes data registers for the alteration of the source code. The signature and function reordering is, as the name suggests, reorders all the subroutines in the executable (You & Yim, 2010). A packer malware archives and combines one or more malicious files and codes to modify its code structure. In addition to the packing operation, a packer malware deletes its import address tables to complicate the analysis (Cheng et al., 2018). In 2006, it is reported that 92% of the malware used similar techniques to evade detection (Wei, Zheng, & Ansari, 2008). A metamorphic malware contains a mutation engine that alters itself in each execution via the packing, encryption, and obfuscation techniques. Although the structure and the instructions of the code is altered in every different version, the aim and functionality of the malware stay the same. In a polymorphic virus, a part of the code that does encryption/decryption is visible in every alteration. Therefore, polymorphic viruses are easier to detect compared with the other techniques.

Dynamic analysis is running the malicious sample in an isolated, secure laboratory environment while examining the behavioral analysis on the sample. Dynamic analysis eliminates the anti-analysis measures of the static analysis, as the dynamic analysis strips the malware out of any encryption, packing, and obfuscation by running it on a sandbox and monitoring its behaviors. Therefore, in dynamic

analysis, the malware is analyzed and caught red-handed during the execution, and any obfuscation, encryption, or alternation to the original code, therefore, becomes meaningless at this stage. This is the main advantage compared to static analysis (Shijo & Salim, 2015). However, dynamic analysis has its limitations. Dynamic malware analysis at its core is accomplished in two significant techniques: binary instrumentation and debugging. The binary instrumentation is a method of fooling the malware by hooking the Application Programming Interface (API) calls and providing the necessary responses to these requests to the malware. These hooks are then logged and converted into behavioral signatures under the assumption that malicious behavior is generally accomplished through utilizing the underlying operating system's API calls. This assumption holds for many cases unless the functions from APIs are natively coded in the malware. The second technique, debugging, suffers from detection by the malware authors since the debugging generally slows down the execution of the steps even though the debugging scheme for detection is automated.

Moreover, Dynamic analysis involves memory analysis techniques such as taint analysis (Korczynski & Yin, 2017) and memory image differentiating (Teller & Hayon, 2014). Taint analysis includes marking some specific memory locations and tracing them along the execution and memory image differentiating is taking snapshots of the malicious process memory on an interval basis or with predefined triggers and contrasting those memory images for maliciousness. These techniques and the literature have been explained in detail in the following chapters in this thesis.

1.1. History of Malware

John Von Neumann gives the first formal definition of a virus in the title of "Self-reproducing automata" (Neumann, 1969). This definition of this kinematic machine is assumed to be the first mention of a machine that is designed to reproduce itself, given the parts and the algorithm. The algorithm in this definition is written on an infinite tape which is considered to be analogous to the definition of the memory. In this tape, a set of instructions which define the definition of the machine itself are stored. These instructions include (i) creating another machinery just as it is from the infinite number of parts that constructs the machine. (ii) creating a tape for the new machine and copying the contents of its tape. (iii) Attaching the new tape to the new machine and thus completing the self-reproduction. This kinematic machine had some physical,

mechanical, and logical limitations such as the infinite number of parts and an infinite tape. With the suggestions of Stanislaw Ulam, one of the simultaneous inventors of the Cellular Automata (Neumann, 1969), Neumann shifts his kinematic model to a Cellular Automata, where the self-reproducing automaton is defined as a Cellular Automata with a finite number of states. An implementation of this work is realized in 1972 by an Austrian computer scientist named Veith Risak (Risak, 1972). In his work, he implemented a fully functional model of a self-replication program on SIEMENS 4004/35 computer system (Miles, 1986).

In 1971, a computer program named “creeper” which was the first version of a worm invented by Bob Thomas was spreading in the DARPANET. This program is not considered to be malicious at all although it copies itself through the network on Tenex Operating Systems on DEC PDP-10 computers and runs its copies on the memory space of another process. This self-propagation property in malicious executables leads to the definition of a computer worm. The name “worm” is originated from the science fiction novel by John Brunner named “Shockwave Rider” in 1975 describes a universe of networked phones and a shutdown software propagating in this universe (Brunner, 1984). Moreover, an early version of anti-virus, Reaper, a worm removal program is then written to remove the Creeper worm from the network and infected computers.

The Creeper and Reaper software both having the properties of a worm and chasing one another inspired A. K. Dewdney to design a game called Core War. The name comes from the early designs of the memories with ferromagnetic cores. The game includes writing two computer programs in a language called Redcode and letting them hunt each other in the memory until one of them dies and erased from the memory completely (Dewdney, 1989).

The first of the significant attacks to the Internet was in 1988. A worm named “Morris Worm” is created and released by the computer graduate from Berkeley, Robert Tappan Morris. The worm was released from a terminal in Massachusetts Institute of Technology laboratories to cover his tracks, and it hits approximately 6000 of computers where an estimate of 60.000 computers was connected to the Internet. This outbreak then leads to the creation of the first Cyber Emergency Response Team (CERT) (FBI, 2018).

In 1984, Kenneth Thompson in his work of "Reflections on Trusting Trust", he has shown how to modify a compiler to insert a backdoor on any computer program that contains "login" command (Thompson, 1984). A Trojan is a computer program that installs itself as a legit program on the host system. In the Thompson's version of a Trojan in this case, the compiler was being the Trojan itself because of inserting a backdoor to a source code even though the source code does not have any vulnerabilities in it. The first Trojan being reported was a game named "ANIMAL" which was a self-replicating program disguised as an animal guessing game. The game asks users questions to find out which animal the user was thinking of while in the background; it copies itself to all the folders of the UNIVAC system which was designed as a folder shared operating system (Miles, 1986).

A critical improvement, self-mutation in virus programming was present in 1990. In 1990, as a part of an analysis project of virus families, a polymorphic virus family called Chameleon was invented by Mark Washburn. A polymorphic virus is a combination of self-reproduction and a mutation engine. The mutation engine is provided by a cipher in this family, and this research has proven that many of the antivirus programs were useless against such a mutation engine (Kaspersky, 2019).

The malicious programs are designed to harm the host systems by inserting backdoors, spreading and attaching itself into files and operating systems or copying themselves through the network and do harm on file systems and disks of the computers. Thereafter, a new type of malware had been released in the early 2000s, the spyware. Although the name "spyware" was invented to mock the business strategies of Microsoft in a Usenet post in 1996, the term then used to define hardware designed for espionage purposes and after that, a software that installs without permission, collects user-related information secretively and transfers the data without the consent of the user. The term is used in a press release in 2000 by Gregor Freund, the founder of Zone Labs for the first time and widely used since then (Avoine, Oechslin, & Junod, 2007). The spyware that is being used for advertisement purposes only is called Adware. As of today, this type of spyware can be found in many freeware/shareware software bundles and installed automatically at the installation of another software.

There were many epidemics in malware history. The first malware that spreads out of the laboratory it was written (in the wild) was the ELK Cloner. It was a virus

for the Apple computers written in 1982 by Rich Skrenta for Apple DOS systems, and the spread was through the floppy disks. Not much long after, in 1989, a hacker with a nickname of “Dark Avenger” in Sofia, Bulgaria, had written a virus for MS-DOS systems that spread globally. It was corrupting up the storage space, directories and files with random codes and the sentence “Eddie lives… Somewhere in time”. The spread was so big that it was all around Europe, even USA and Australia (BitDefender, 2010).

Around December 1989, the first sample of the ransomware has been produced by Joseph L. Popp. It was before crypto coins and even the internet. The virus was spread through a floppy drive, disguised as an educational floppy diskette about AIDS virus. After the infection, the virus was encrypting all the files and folders of the computer and asked the user to send money to a post office box in Panama.

The first virus exchange platform was set up in Bulgaria at the beginning of the 1990s as a Bulletin-Board System (BBS). The virus database was open to anyone who uploads a new virus code. This system had led the malware writers to evolve and improve while letting the malware to be anonymized. In 1992, these contributions to the virus databases had resulted in the creation of tools and engines that generates viruses such as Self-mutating engine (MtE) and Virus Creation Laboratory (VCL). These engines contain prearranged payloads and scripts, with which, even script kiddies could generate new viruses by mixing the viruses in their databases.

One example of utilization of these databases was the Loveletter virus and its 90 variants in the 2000s. The virus also is known by the name “ILOVEYOU” or “The Love Bug”. The malicious code was spread through emails disguised as love letters. Within ten days of the first outbreak, %15 of all networked computers were infected with one of the samples of Loveletter (BitDefender, 2010).

Malicious worldwide spreads have continued from the 2000s to the present day. Conficker (or Downadup) Worm was first of the greatest hits in malware industry, affecting 15 million systems worldwide in 2009s (Touchette, 2015). The Rebirth of the ransomware: Cryptolocker affected 250.000 machines in 2013. The ransomware families have grown a lot since then with the Locky, CryptoWall, CryptoDefense, WannaCry and several other ransomware.

In 2007, the malware had evolved into a new type of attack mechanism with the Stuxnet worm. Malware has been used as a weapon by the governments with the pronouncements of cyberspace as one of the war domains. This very sophisticated worm had been tailored for a specific device, and a persistent campaign is followed until the attack had reached its goals. This type of attack is called an Advanced Persistent Threat (APT). Similar aimed malware had been developed after the Stuxnet: Flame, Duqu, Duqu2. All of them were tailored and weaponized malware aimed at a specific purpose and used in a campaign of a government or groups.

Malicious outbreaks and infections are continuing today with the malicious code databases, state-sponsored actors, ransomware and botnet developers, code exchange platforms on unsolicited and unmapped domains of the Internet such as Darknet or Deep web. As our security measures evolve, the malware and malware families are evolving at a fast pace as well.

1.2. Problem Definition, Motivation, and Aim

The purpose of malware is to hide from the infected host system and conduct malicious acts using the functions, resources, and communications of the system when triggered. Therefore, malware developers intend to find new solutions of hiding from anti-virus systems day to day and integrate these solutions and evolve their malware into new versions by these solutions.

A signature of malware is a unique set of bytes that shows the existence of the malware in a file or on memory. This set of bytes can also be the hash or checksum value of a specific portion of the code, a particular file that the malware drops to host system, or malware-specific indicators on memory when the malware runs.

There are also behavioral signatures that show if a malicious binary is present such as special registry keys that the malware creates and alters or a connection to a malicious IP address which is generally the command and control (C2) server. These signatures can also be an operating system kernel call to allocate some memory to unpack and decipher itself on memory, or an attempt to shut down some a property of an operating system or anti-virus to continue executing without detection.

There are many ways of malware to alter itself and evade from the static signature detection.

1.2.1. Obfuscation

Obfuscation is a defense mechanism for static analysis. It is a technique to transform the malicious code into identical but differently represented new code. A signature generated by a signature analysis consists of a piece of codes represented in their byte form and/or relationships between jumps and calls. For such signatures, instruction-level obfuscations such as using redundancies in instruction sets, dead variable and code insertions, obfuscating the imports table are generally used to evade signature detection.

1.2.2. Oligomorphic Malware

An oligomorphic malware is a type of polymorphic malware where there is a simple decryptor engine for changing the malicious code with encryption. Usually a simple, low-cost and, with a small key size encryption mechanism is used to change the byte code of the malware, and since the number of possible alterations of the code is low because of the small key size, it is possible to detect such malware by generating signatures for all possible keys.

1.2.3. Polymorphic Malware

A polymorphic malware changes its shape at every infection and execution through encryption. This type of malware contains a mutation engine inside its code. The mutation engine generates not only a new key, a new decryption routine as for every execution as well. Therefore, polymorphic malware with a moderately complex mutation engine typically can reshape itself into around a billion of different versions of itself.

1.2.4. Metamorphic Malware

A metamorphic malware changes its shape and form at every infection and execution just as a polymorphic malware but to do so, it utilizes code renaming, adding random codes to itself, changing used registers in the code and, changing the level of optimization provided by compilers.

1.2.5. Dead Code Insertion

Inserting codes and functions that are never going to be executed by the code is called Dead Code Insertion. This technique aims to evade detection by the signatures generated from file hashes. By adding random extra functions and codes to its file, the hash value of the executable changes and the detection is tried to be avoided.

There are also problems in signatures that are extracted from dynamic analysis as well.

1.2.6. Anti-Vm And Virtual Interface Detection

Sandboxing techniques for dynamic analysis involve creating a safe and isolated environment for running the executables. Due to their ease at maintenance and reproduction, Virtual Machines are selected for this task. However, Virtual Machines and their interfaces such as Virtual Network Cards uses particular keys in the registry and also leave particular imprints on the memory on the operating system of the Virtual Machine. Malware writers analyze these imprints and use them to avoid sandboxes by adding extra countermeasures such as not decrypting itself if an indication of Virtual Machine is present.

1.2.7. Anti-Debugging

There are conventional techniques among malware writers for understanding debugging. The interrupt INT 3 is commonly used for debuggers to break execution at each step and hand the control to the debuggers and malware writers generally check these interrupt flag to avoid being analyzed. Another common technique is to check the time interval between two instructions. If the wait is longer than a regular fetch-decode-execute cycle, the malware does not reveal itself.

1.2.8. Execution Stalling

The typical approach for running an executable on a sandbox is to pre-set a duration for the analysis. This time duration is generally set by the analyst to automate the analysis of the malware. For another anti-sandboxing measure, the malware writers add a sleep cycle at the beginning of their code, so that if it is a sandbox, the malicious activities will not be caught by the sandbox as the analysis time would be over.

1.2.9. Logic Bombs – Extended Sleeps

Similar to the execution stalling, malware writers set a random date and time to execute their malicious goals and let the code stay dormant and inactive until this time comes. The malware won't be caught by the sandbox as the analysis time would be over. Another stalling technique to add extended sleeps to avoid being analyzed in the timespan of sandbox analysis.

1.2.10. Native DLL Coding

Most behavioral and dynamic analysis and signature extraction techniques are based on the API calls of the malware. The decision of maliciousness, the aim of the executable, the countermeasures being taken by the malware writer and, the communications are generally detected by hooking API calls by the sandbox Operating System or a particular DLL that is injected into the binary that is being analyzed. Malware writers try to avoid these signatures by writing their native DLLs and system calls. However, coding native system calls require expertise on operating systems and hardware, and therefore, it is not common among malware to be written in such an expert-level way.

1.2.11. Process Injection

As mentioned above, the malware is being monitored and analyzed by the API and system calls. In sandboxing, the malicious code and the child processes that it starts are monitored. However, some malware tries to inject itself into address space of another process and run their API and system calls from another process that is not being monitored. This technique, of course, requires to integrate a module of exploitation into the malicious code.

The summary of these evasion techniques, both static and dynamic, and their countermeasures are given in a problem tree in Figure 1.1. In Figure 1.1, the techniques are given in rounded rectangles, and the countermeasures are given as rectangles.

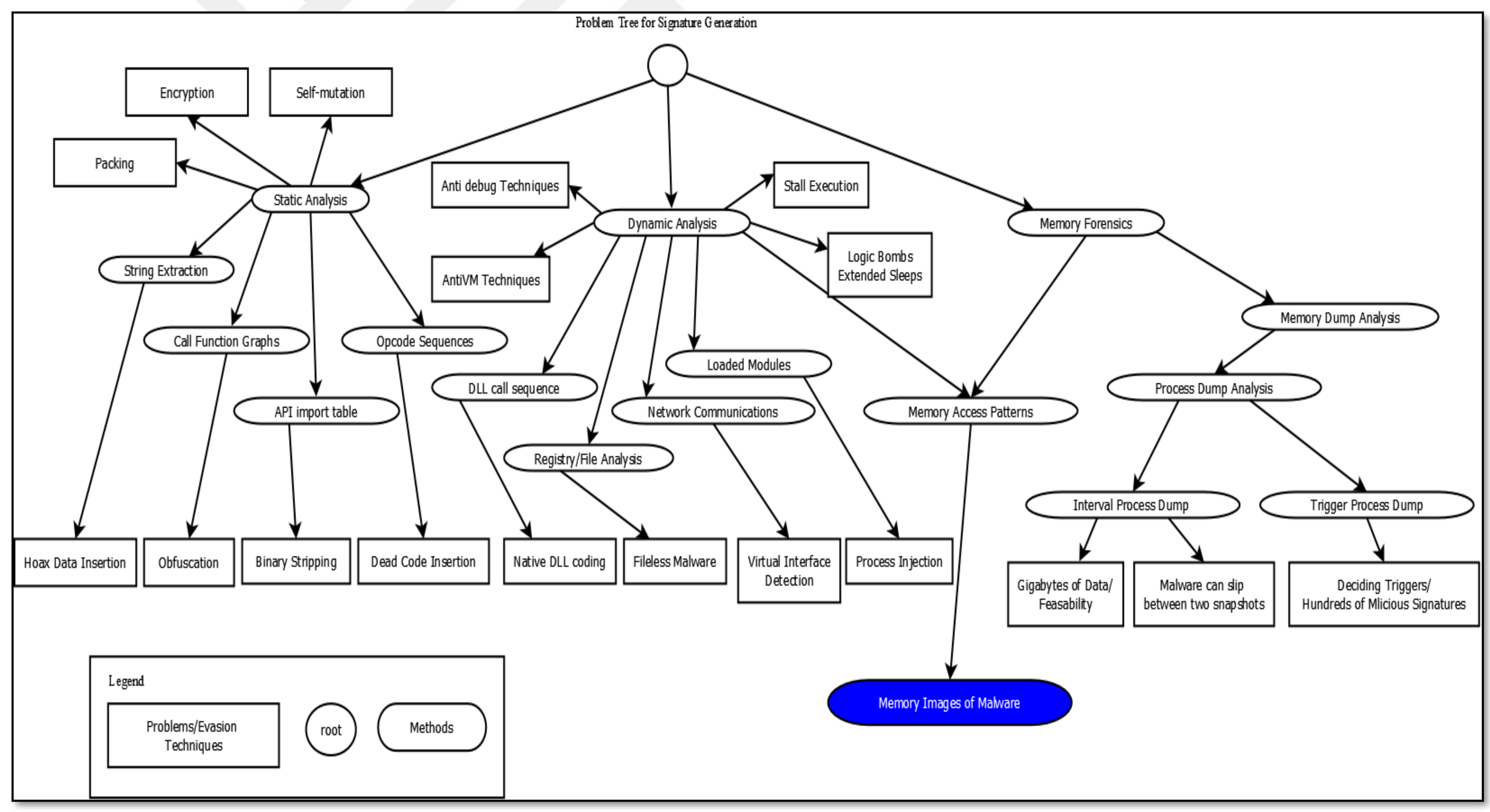


Figure 1.1- Problem Tree of Signature Generation with Malware Analysis.

Moreover, memory forensics had provided several ways to generate signatures from the address space of the executable from memory. This kind of signature generation involves capturing the address space either by taking the snapshot of the process address space between time intervals (Interval Process Dump in Figure 1.1.) or taking the snapshot of the memory space by some triggers such as suspicious API calls. Considering the case of saving the regular snapshots, the problems within this technique are (a) it produces gigabytes of memory data to analyze (b) the malicious actions can slip between two intervals. When utilizing system calls to trigger taking snapshots of the memory, with this evolving structure of malware, it is hard to decide which system and API calls are suspicious.

In this thesis, the primary motivation is to propose a novel approach to imaging the malware through the artifacts from the memory operations, and the primary aim is by utilizing these artifacts and patterns, to identify malicious acts visually and test a software rapidly for maliciousness.

So far, in the literature, Deep Learning and Machine Learning approaches have been explored. Several metrics are constructed utilizing Call Function Graphs, and API calls from the static and dynamic analysis. However, to the best of our knowledge, there are no comprehensive comparison and classification technique that uses memory patterns has been suggested to the literature. This thesis aims to accomplish this by extracting unique runtime patterns of malware from memory. In the purpose of comparison, the patterns are converted into 3d images, and a very well-known pattern and image comparison technique that is widely used in the literature has been integrated into this research.

The results of this work generated a framework for the visual detection of maliciousness with memory patterns. These patterns are inserted into a graph database for having a structural and queryable platform. The 3d images are stored along with the memory data. This implementation and collection of patterns will act as a visual aid and a fingerprint database.

1.3. Thesis Outline

This thesis is outlined as follows:

- Chapter 2 – Defines the modularity and anonymity of malware, introduces the malware families, and explains the spread of malware.
- Chapter 3 – Presents the static analysis methods and the literature along with results of static analysis with their graph database representations.
- Chapter 4 – Surveys the literature of dynamic analysis.
- Chapter 5 – Identifies the memory layout of an executable, presents the memory forensics literature and explains the idea used in this thesis.
- Chapter 6 – Reveals the methodology, the algorithms for binary instrumentation, and the extraction of the memory patterns.
- Chapter 7 – Shows the results of a malware dataset that is constructed for this research.
- Chapter 8 – Concludes this thesis with novelties, results, and future work.

CHAPTER 2
BACKGROUND: MALWARE STRUCTURE, FAMILIES AND CONSTRUCTION

This section contains the general background information on the structure of the malware, how the malware is designed and executed and, how they exploit the general structure of executables on target hosts. Because of the extensive usage of the Portable Executable (PE) Format among malware authors, this information is exampled in this format throughout this chapter.

In a highly abstract view, the structure of different malware characteristics can be captured. While the syntactic features can be altered through evasion techniques that are mentioned in Chapter 1, structural properties are not likely to change as easy. In general, malware is designed to exploit a vulnerability of either an operating system component or a user application to accomplish its tasks. No matter how much the code itself modified, the target and the exploitation methodology stay the same or very similar for malware. This chapter aims to reveal these characteristics in a general view. Along with the general algorithms and methodologies of malware types and the modular malware are presented in this Chapter.

2.1. General Layout of PE Files

The PE format is the general data structures for the binary files compiled for Windows applications in 32 and 64 bits. The format is used both on-disk and in-memory representations. Because of the memory address alignments and dynamically loaded modules of a binary, the memory layout is slightly different from the on-disk representation.

The general format of a PE file is depicted in Figure 2. The explanation of each part of the header is as follows:

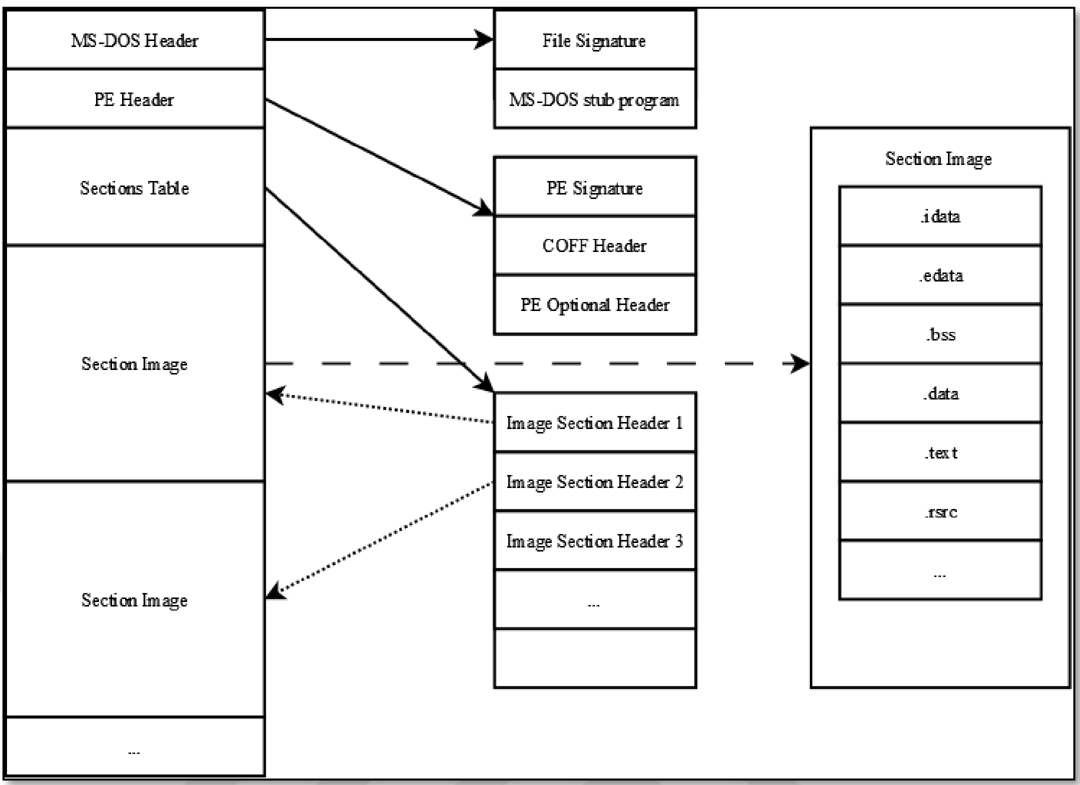


Figure 2.1. Structure of PE file Format (Pietrek, 2011).

- MS-DOS Header: This header contains the file signature of executables MZ (initials for Mark Zbikowski) and an MS-DOS stub program that is left in this header for backward compatibility. This MS-DOS stub only prints out a message that states that the program can not be run under MS-DOS.
- PE Header: It contains two separate headers: image file header structure or namely the COFF header and image optional header as stated in Figure 2.
- COFF Header: This header has seven fields in its data structure:
    - Machine: It holds the information about the architecture of the target machine.
    - NumberOfSections: it defines the number of sections in the PE file.
    - TimeDateStamp: holds the compilation time and date.
    - NumberOfSymbols: Number of symbols in the Symbol Table.
    - SizeOfOptionalHeader: Size of the optional header in bytes.

  - Characteristics: contains information about the PE file.

- Image Optional Header: This optional header contains the critical information about the executable file such as entry point, alignment of the executable file sections and memory sections, beginning of the code section, DLL characteristics and so on.
- Section Headers defines the size, location, and permissions of each section in the PE file.
- In section images, there are several sub sections that contains data (.data), uninitialized data (.bss), code (.text), resources (.rsrc), import address information (.idata) and export address information (.edata).

Malware authors try to hide their mal-intended codes inside the parts of this data structure using obfuscations, encryptions, polymorphism techniques, etc. The techniques for understanding this data structure, and the sections without executing the code, therefore has limitations that are emerged from this structure itself. These limitations are discussed and shown through 2-dimensional and 3-dimensional models in this thesis in Chapter 3 and Chapter 6. Following parts of this Chapter concentrates on the clean and stripped versions of the types of malware.

2.2. Structure of a Virus

The definition of a virus is a piece of malicious code that copies itself on other files and executes a payload to accomplish its malicious tasks. Thus, there are at least two parts for a virus, which are the infection code and the payload (Sharp, 2007). General pseudocode for a virus is presented in Listing 1.

Listing 1 – Pseudocode for a schematic virus (Sharp, 2007).

```
Begin Procedure:
if spread_condition := true:
    for v ∈ victim_files :
        if not_infected(v):
            begin_code, end_code = get_placement_for_virus(v);
            copy_virus(begin_code_ end_code, v);
            modify_to_execute_inserted_virus(begin_code, v);
        endif;
    endfor;
endif;
execute_payload();
start_execution_of_infected_program();
End Procedure
```

The procedure of a virus starts with checking the replication environment first. In the execution of this part, malicious code might check whether the machine is an actual machine or a known sandbox, how much ram does the host machine has, how many users are registered to this computer, any known signature of antivirus is present and so on. These are the critical decisions of malware regarding its purpose and aim. There are some types of malware produced explicitly for particular machines as a part of highly advanced campaigns in Advanced Persistent Threats (APT) (Bencsáth, Pék, Buttyán, & Félegyház, 2012; Falliere, Murchu, & Chien, 2011).

The virus then lists the potential targets in the host system and starts iterating them for replication. If the file is not already infected (as every copy of this abstract virus will work concurrently), it finds the suitable section in the file for replication and drops itself inside the file. During this environment, if the target file is an executable (generally preferred to be such), the structure of the executable is exploited, and the entry point of the target executable is altered for the replicated virus to start. After all the potential files are infected, the payload is executed. Execution of the payload is the part where actual harm is done besides the illegal replication process. In general, after

the job of the virus is completed, the infected program is also run so that the virus itself lives undetected in the infected program.

As an example of this process, the virus code with the name of **Virus.W32.Virut.ce** is presented and analyzed in this part. The technical analysis of this virus is done by Kaspersky Laboratories and presented as a web page in 2010 (Zakorzhevsky, 2010). This sample is known as one of the fastest spreading types of a virus, and it utilizes many evading techniques such as polymorphism, obfuscation, anti-debugging, and anti-virtual machine.

As for the spreading conditions, this sample checks the tick count of the host machine by the instruction **rdtsc** and **GetTickCount()** function of Windows API. Getting the tick count between instructions are aimed for determining if the virus is being debugged or analyzed in a sandbox environment. If the tick count holds the virus continues with the spreading.

This virus tries to replicate itself through a Portable Document File (PDF) plugins vulnerabilities of browsers. The virus copies a download link line for the .htm, .php, .asp files of the target computer. Also, it attaches itself to small executable files such as keygens and crack programs.

The virus uses Entry Point Obscuring (EPO) methods to run its payload. It inserts itself to the address space of Explorer.exe (or services.exe, iexplorer.exe) and it alters the entry point line in the optional header of the PE structure so that the payload executes. This strategy is illustrated in Figure 2; it contains injection into another process address space, obscuring the entry point, decrypting the original virus code and execution.

Afterward, it connects to a Command and Control address and retrieves further instructions. Although this property is a Trojan-type malware property, these lines in-between malware types are quite blurred as every malware author tries to improve their code by adding several functions from other malware. This property is discussed in the malware families part later in this Chapter.

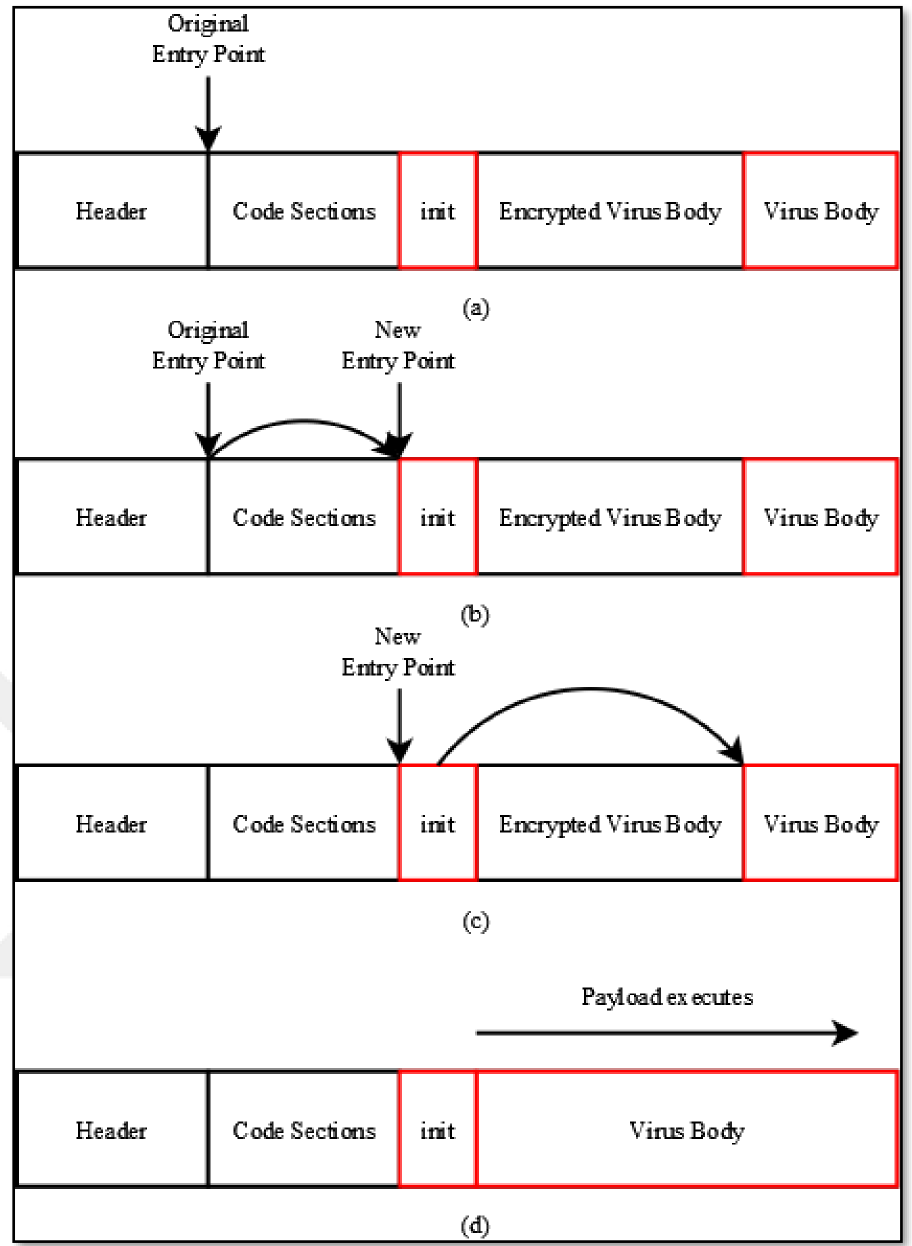


Figure 2.2. (a) The virus inserts itself into another executable. (b) The virus alters the entry point in the header. (c) New entry point jumps to decryptor of the static encrypted virus body. (d) The decrypted virus body executes.

The virus uses two engines for modifications while spreading to avoid signature-based detections. While the first one changes the code itself, which decrypts the malware, the other one decrypts the static body of the malware. More details on this malware can be found in the reference article (Zakorzhevsky, 2010).

2.3. Structure of a Worm

Similar to a virus, a worm also has the capability of reproduction on a network environment. The anti-inspection and disguise techniques are similar, but a worm generally contains other parts for finding new targets on the network and exploiting some vulnerability to a new host. Therefore, all worms contain the three essential parts: the sniffer, the propagator, and the payload. Since the infection is on a target network rather than on a target host, there are some other managemental parts for a worm such as life-cycle manager, remote control and update manager (Szor, 2005).

A general worm strategy can be given in the following pseudocode in Listing 2.

Listing 2 – Pseudocode for a schematic worm (Sharp, 2007).

```
Begin Procedure:
if spread_condition := true:
    victim_hosts_addresses = search_for_victims();
    for v ∈ victim_hosts_addresses :
        if not_infected(v):
            fingerprint_network_host(v);
            transfer_code(v_address, port, payload);
            send_message_to_execute_on_host(v);
        endif;
    endfor;
endif;
execute_payload();
check_for_updates(); //optional
listen_remote_connections(); //optional
End Procedure
```

Many versions of worms rely on human interactions to execute on the targeted hosts. It can be an email attachment to view or a network connection to accept. However, as the sophistication of malware improves, the vulnerabilities on target hosts are utilized, and the need for the interaction is decreased for the propagation.

New targets of the worm can be found in many ways depending on the propagation strategy. The most common ones are as follows:

- Email Discovery: In this type of spreading strategy, the worm looks for contacts and email addresses in human readable files and send itself to these addresses through an email client.
- File Share Discovery: The worm searches for shared files, folders and drives and locates itself to these drives or files for propagation.
- Communications Discovery: This strategy includes the worm sniffing the network communications of the current host and trying to exploit vulnerabilities of network communications.
- Network Discovery: This strategy needs the worm to have network scanning modules to provide propagation. Some search strategies on a networked environment for a worm would be random scanning, permutation scanning, localized scanning, hit-list scanning, topological scanning, meta-server scanning (Smith & Matrawy, 2008).

As an example of this strategy, the analysis report for W32.Waledac worm is used in this part (Tenebro, 2009). The worm Waledac is a multipurpose worm; it has the functionalities of emailing, vulnerability exploiting, mining the host, acting as a proxy and acting as a binary downloader.

The first propagation strategy of this worm is social engineering, which involves sending Christmas themed emails, phishing emails about the election campaigns of 2008-2009 and a popular news site. The second strategy is about the websites that host this virus; these websites exploit browser vulnerabilities for the victims to download the malware.

This malware aims to create a botnet through spamming and steal FTP information from the infected targets. After successfully running on a host, the emails in user files (other than video and music files) are gathered and spamming continues. As for the payload, the malware creates a node list of other bots in its neighborhood and generates a fast-flux network structure for malware hosting nodes. The nodes act as a proxy for the hosting websites to avoid detection.

2.4. Structure for a Trojan

A trojan is engineered to resemble a useful, harmless program that gains control of the target computer and do harm or steal data and network information. A trojan is a software that aims to acquire full control of the target machine. There are many types of Trojans such as a banking trojan, a fake anti-virus trojan, a distributed denial-of-service trojan, a backdoor trojan, a ransom trojan, an info-stealer trojan, and so on. Therefore, it is hard to derive a single procedure to cover all these aspects. However, their code structure contains an installment procedure, a sign-on procedure, a privilege escalation procedure, and a connection interface with the command and control (C&C). These procedures are depicted in Figure 2.3.

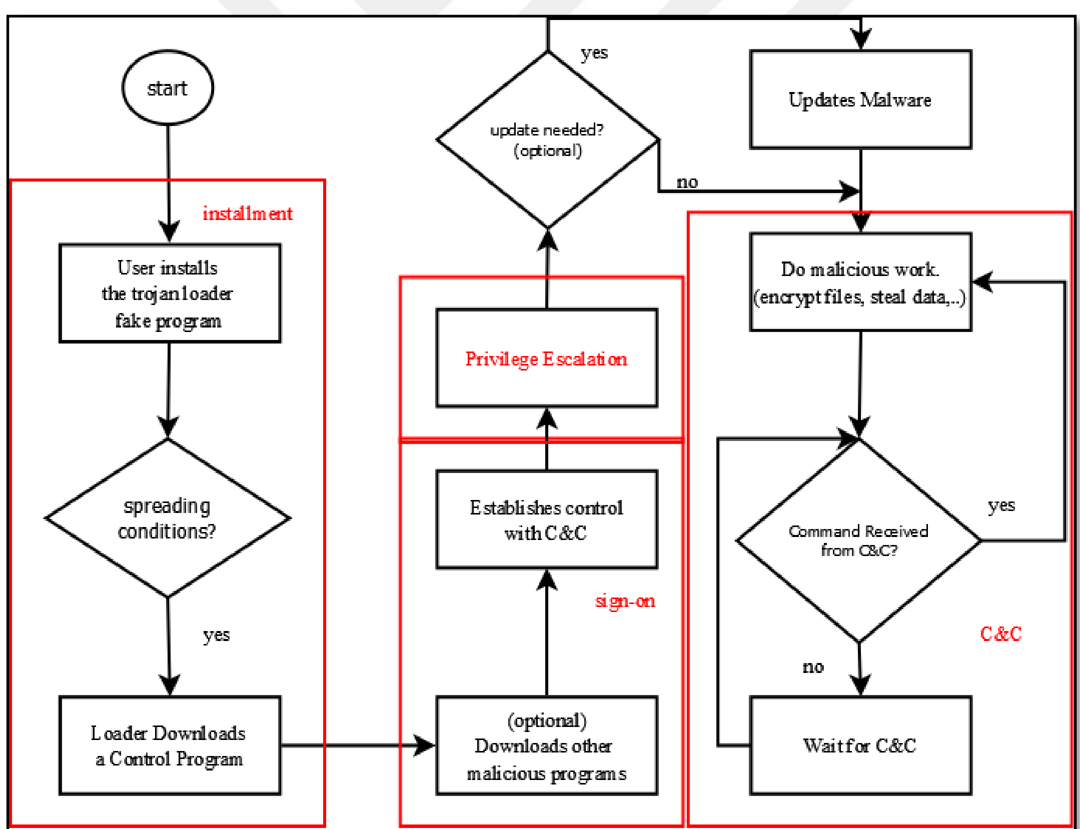


Figure 2.3. A Flow Chart of a Computer Trojan.

In the installment procedure, the malicious code is generally hidden inside another executable, such as an online gaming client or a keygen. The fake cover

executable is designed to trick the user into installing to the target system. It can also use social engineering techniques to install itself on the potential host.

The sign-on procedure involves connecting to the master server for further instructions. It generally includes getting general information on the host machine to create a unique identification code for the host computer. An example of this procedure is the banking Trojan.Dyre (Symantec, 2015). Trojan.Dyre aims to steal banking credentials by directing the user of the host computer to fraud websites. It uses spam email messages as an infection vector and installs itself through the Downloader.Upatre which is used in many attack campaigns in computer crimes history. After Trojan.Dyre has been installed on the target computer, it collects the banking information from vulnerabilities of popular browsers, generates a campaign Id, host Id and logs in to the C&C server to transfer the credentials.

In the privilege escalation procedure, computer trojans use an operating system vulnerability to gain root access on the host system. In the example of Trojan.Dyre, a vulnerability in a database compatibility service program, is utilized to redirect root privileges to run the malware component. As for the C&C procedure, either a full remote control is provided, or an interface or an encapsulation module for the desired commands are provided to the attacker.

2.5. Malware Families

A group of malware that shares common properties and functionalities is called malware families. There are many reasons for malware to grow into a family. The first one is a process called anonymization. Anonymization is done by the malware authors to hide the actual author of the malware; they upload their malware to a code-sharing platform, and within hours, several new instances of the malware are created. Other malware authors can get parts of it and turns into some other weapon for another attack, or they can directly use it as happened in the case of the Downloader malware family (Gupta, Kuppili, Akella, & Barford, 2009). Many trojans use one instance from the Downloader family to spread their malware; and as mentioned in the previous part of this chapter, a downloader may be configured to download several different malware. In the case above, it was used for downloading several botnets so that once a computer is part of a botnet, it is automatically becoming part of several others.

Another reason the number of malware instances from the same sample increases in the wild is the self-mutation engine written by the author. It is done to avoid detection, especially from the signature detection mechanisms. In general, the tactics involve adding several NOP operations to change the cryptographic hash values of sections, and the number of NOP operations picked randomly, using interchangeable assembly instructions such as conditional jumps that check the same condition, pushing and popping values from the stack without the actual need or using unnecessary swap operations between registers. Utilized these and similar approaches, hundreds of different versions of malware can be encountered in the wild.

There is an arms race between malware authors and computer security specialists. As the anti-virus vendors find new solutions and detection techniques to stop malware, the authors of the malware continue to patch their malware to survive in the wild. As mentioned in the structure of trojans section, many new generation malware has capabilities of updating themselves while the malware authors manage their lifespans through controlled C&Cs. A recent study researching the questions about the lineage, lifespan, and the number of generations of a malware family is presented in the work Gupta et al. (Gupta et al., 2009). In this study, 669 different malware families identified over 19 years of malicious code metadata. This research shows that some malware families can survive for a few years using the patches and updates coming from the malware authors community. Malware lineage is the problem of identifying the versions of malware among samples which is another research field approaching the same problem (Haq, Chica, Caballero, & Jha, 2018; Heinricher & Jilcott, 2013; Karim, Walenstein, Lakhotia, & Parida, 2005; Walenstein & Lakhotia, 2007; Xin & Zhang, 2007).

Contemporary attack vectors include several malware in their weaponization: trojans are used for initial attacks, worms are included for their lateral movement and rootkits are deployed for remaining the communication with the targeted systems. This is another reason because of which a number of malware samples in a family increase. Malware authors constructing several malware as a bundle and generating malware families in a modular, compartmented fashion. These malwares are called *modular malware*. In this type of malware, the attacks are staged into phases, and for every phase, a different portion of the code harms the targeted system. A recent example of such malware family is the botnet service named DiamondFox, which is a malware-

as-a-service platform. It has several modules and functionalities such as browser password stealer, FTP stealer, DDoS, Email Grabber, RAM scraper, Spam Function, CryptoWallet Stealer and so on. All of these functions are provided as a plugin to this malware. It has a technical support staff; all of these features bundled as a fully-fledged business service.

2.6. Conclusion

In this chapter, the general structure of an executable and, the capabilities of contemporary malware are investigated. It is intended to explain how the malware evolves, how the malware families are constructed, and to what extent the capabilities of malware authors reach. In the following chapters, the malware analysis mechanisms, both static and dynamic, and a novel contribution of memory analysis is presented.

CHAPTER 3
STATIC ANALYSIS OF MALWARE

This chapter is aimed at identifying the contemporary techniques of static binary analysis along with the literature of malware detection and classification based on these techniques. The limitations of static analysis and the evasion techniques from static signature-based detection schemas are introduced in the previous chapters. In this chapter, while a novel methodology of visualizing and fingerprinting of malware is provided, also the limitations of static analysis are exampled and illustrated through this developed methodology.

Moreover, in this Chapter, an implementation of Convolutional Neural Networks (CNN) to the static features of malware is provided. For this aim, first, a graph database representation of malware is presented along with the feature extraction queries. This representation is created by the extraction of static properties such as Call function graphs and API calls; which are extracted with the reverse engineering tool RADARE, and these results of the static analysis are inserted into a graph database which is created by the Neo4j Graph Database application. Secondly, the implementation of a CNN classifier is demonstrated on the images that are extracted from such data. For demonstration, a collection of a recent malware sample space is analyzed. Finally, the limits of static analysis are also discussed on the results of the implemented CNN.

3.1. Static Analysis Techniques

Static analysis of malware is the collection of the techniques that are used on the binary file without mapping on the memory and without executing. It provides a rapid overall inspection of the binary file, reveals general information about how the binary is compiled, gives insight about which API calls and libraries are used and, presents the structure of the malicious file. It is also used for understanding if the binary has been encrypted, obfuscated, and packed. In general, tools such as disassemblers, PE file analyzers, and tools for searching for strings and binary patterns are utilized. The section headers, mapped resources, symbolic links, and dynamically linked libraries

and modules are also available if no counter-measures are taken when compiling and preparing the binary.

Extraction of Strings

Hardcoded strings inside a binary can sometimes provide useful insight into the file. By utilizing string search algorithms, the following indicators of a suspicious file can be found.

- In the cases of creating malicious files on the targeted system, malware can be matched using these hardcoded names extracted from the executable.
- Most current malware searches for the processes with the names of most common anti-virus vendors. Finding these names in the executable files can be an indicator of maliciousness.
- When a malware tries to connect with the C&C, it is done by resolving a domain name or trying to establish a connection with an IP address. This information is generally hard-coded in the executable; therefore searching for a string with the format of IP addresses or URLs are useful.
- When the malware drops another executable as a backdoor or a bot service, it registers the file as a service and therefore inserts and alters the registry keys. Because of this reason, searching for a registry key in the strings can be identifying for malware as well.

The string extraction can be used to quickly check for suspicious things in a binary, although generally, it does not provide a clear picture. The string extraction of malware is exampled in the following Table 3.1. and 3.2.

Table 3.1 Suspicious strings of W32.eternalrocks creating and installing an onion network node in the Microsoft updates folder.

(Sha256:1ee894c0b91f3b2f836288c22ebeab44798f222f17c255f557af2260b8c6a32d)

Ordinal	Virtual Address	Physical Address	Size	Length	Section-Type-String
298	0x0000a79e	0x0040c59e	14	15	(.text) ascii 020430Project1
299	0x0000a7ad	0x0040c5ad	7	8	(.text) ascii 0-C000-
300	0x0000a81d	0x0040c61d	4	5	(.text) ascii orm1
301	0x0000a826	0x0040c626	5	6	(.text) ascii Form1
302	0x0000a839	0x0040c639	5	6	(.text) ascii Form1
303	0x0000a8b0	0x0040c6b0	4	5	(.text) ascii VB5!
304	0x0000a928	0x0040c728	8	9	(.text) ascii TorUnzip
305	0x0000a931	0x0040c731	8	9	(.text) ascii Project1
306	0x0000a93b	0x0040c73b	8	9	(.text) ascii Project1
308	0x0000aac8	0x0040c8c8	47	96	(.text) utf16le *\AC:\Users\tmc\Documents\TorUnzip\Project1.vbp
309	0x0000ad70	0x0040cb70	8	9	(.text) ascii Project1
310	0x0000ad7c	0x0040cb7c	5	6	(.text) ascii Form1
312	0x0000adec	0x0040cbec	59	60	(.text) ascii C:\Program Files (x86)\Microsoft Visual Studio\VB98\VB6.OLB
313	0x0000ae54	0x0040cc54	12	13	(.text) ascii WindowsUnZip
314	0x0000ae78	0x0040cc78	45	92	(.text) utf16le \Program Files\Microsoft Updates\temp\tor.zip
315	0x0000aedc	0x0040ccdc	37	76	(.text) utf16le \Program Files\Microsoft Updates\temp
316	0x0000af60	0x0040cd60	24	50	(.text) utf16le ripting.FileSystemObject

Table 3.2 Suspicious strings of W32.carberp extracted possible signature filenames. (sha256:4297ad0f5bb72616337d88f14c07a6c6d6e0c93d2a9bb5eaa7e09219556aafdb)

Ordinal	Virtual Address	Physical Address	Size	Length	Section-Type-String
6	0x00025d10	0x0044e110	12	26	(.rsrc) utf16le BuML8ymRlYnf
7	0x00025d32	0x0044e132	15	32	(.rsrc) utf16le FileDescription
8	0x00025d54	0x0044e154	10	22	(.rsrc) utf16le 0mJ8otjGpz
9	0x00025d72	0x0044e172	11	24	(.rsrc) utf16le FileVersion
10	0x00025d8c	0x0044e18c	12	26	(.rsrc) utf16le S3BF7IZ2ZLiF
11	0x00025dae	0x0044e1ae	12	26	(.rsrc) utf16le InternalName
12	0x00025dc8	0x0044e1c8	6	14	(.rsrc) utf16le KayU1y
13	0x00025dde	0x0044e1de	16	34	(.rsrc) utf16le OriginalFilename
14	0x00025e00	0x0044e200	9	20	(.rsrc) utf16le 7AhVva8ai
15	0x00025e1a	0x0044e21a	11	24	(.rsrc) utf16le ProductName
16	0x00025e34	0x0044e234	14	30	(.rsrc) utf16le 08ZkvxeDt8DPLE
17	0x00025e5a	0x0044e25a	14	30	(.rsrc) utf16le ProductVersion
18	0x00025e78	0x0044e278	7	16	(.rsrc) utf16le lxqR7dS
19	0x00025e90	0x0044e290	10	22	(.rsrc) utf16le arFileInfo

Symbols

Symbols are the various entities about the executable such as variable names, imported functions, function names, and objects. This information is stored in the Symbol Table by the compiler. This table reveals useful information about the code itself when the binary is not stripped. An example of symbols in a malicious binary is demonstrated in Table 3.3.

Table 3.3 Imports table extracted from the symbol table of malware Win32.Emotet. (sha256:6393fe8dd4721190f240e22feeb769675b6194a70cabd5a415c2364686a9089c)

Ordinal	Virtual Address	Physical Address	Type	Size	Name
107	0x00001134	0x00401134	FUNC	0	imp.MSVBVM60.DLL__CIatan
108	0x00001138	0x00401138	FUNC	0	imp.MSVBVM60.DLL___vbaStrMove
109	0x0000113c	0x0040113c	FUNC	0	imp.MSVBVM60.DLL___vbaStrVarCopy
110	0x00001140	0x00401140	FUNC	0	imp.MSVBVM60.DLL___vbaR8IntI4
111	0x00001144	0x00401144	FUNC	0	imp.MSVBVM60.DLL__allmul
112	0x00001148	0x00401148	FUNC	0	imp.MSVBVM60.DLL__CItan
113	0x0000114c	0x0040114c	FUNC	0	imp.MSVBVM60.DLL___vbaAryUnlock

114	0x00001150	0x00401150	FUNC	0	imp.MSVBVM60.DLL___vbaVarForNext
115	0x00001154	0x00401154	FUNC	0	imp.MSVBVM60.DLL__CIexp
116	0x00001158	0x00401158	FUNC	0	imp.MSVBVM60.DLL___vbaFreeObj
117	0x0000115c	0x0040115c	FUNC	0	imp.MSVBVM60.DLL___vbaFreeStr

Control Flow Graphs (CFGs)

Control Flow Graph is one of the most common used signatures in the malware analysis community to identify the characteristics of a binary. It is a directed graph where each node represents a block of code or a function, and each arrow represents the flow of execution *calls* or *jumps* in the executable binary (Nguyen, Nguyen, Nguyen, & Quan, 2018).

To obtain these graphs, the complete disassembly of the binary should be searched for cross-references. A cross-reference can be a call to a function, a jump in the address space, a return function, and so on. An example of CFG is illustrated in Figure 3.1. As can be seen in the example, the flow of execution is given for the sample Virus.Xpaj.c with the given cryptographic hash function sha256 of 5cb89de13b078839bf8c56549de1fbf99a73dd8179d150d2cd975722e9f70e5. Each node represents a function either from the code section (named as fcn symbols) or a function from an imported library (named as dll_). Figure 3.1 contains only a portion of the given sample as the whole CFG is too large to fit. In the second figure, another version of CFG of this sample is provided. Figure 3.2 is an output of the graph database that is constructed for this research.

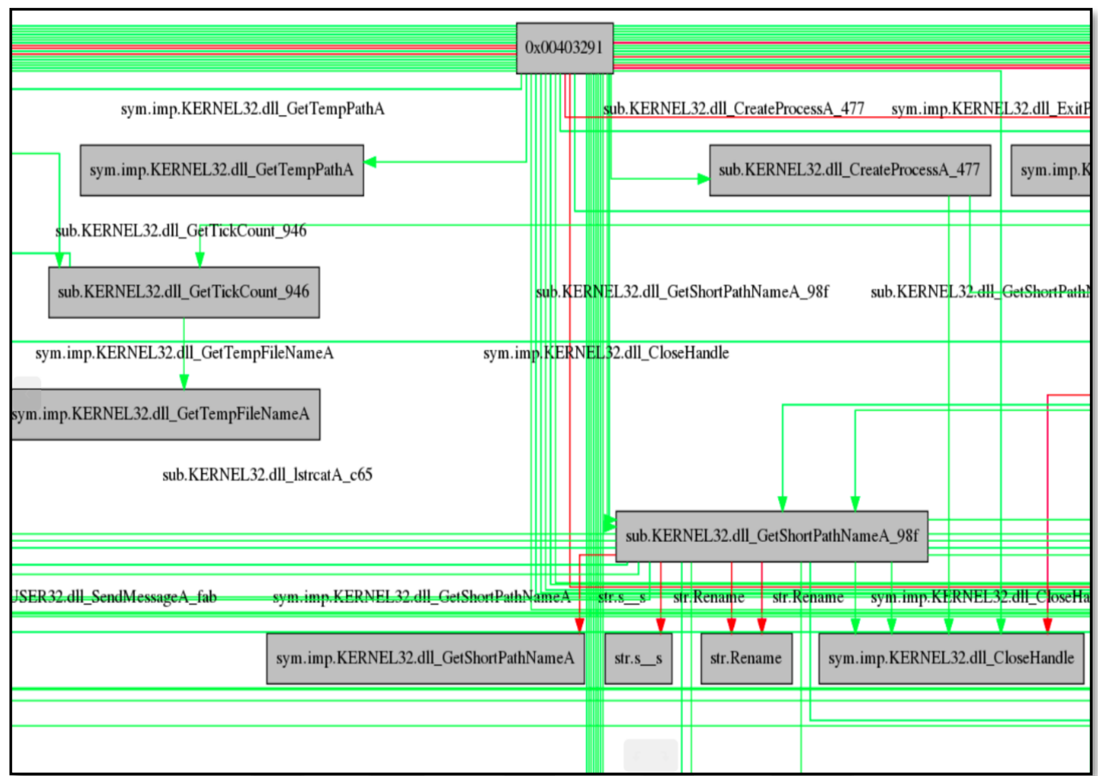


Figure 3.1 CFG of Virus Xpaj.C with hash 95cb89de13b078839bf8c56549de1fbf99a73dd8179d150d2cd975722e9f70e5

These CFGs are converted into nodes and edges in a graph database to store much more characteristics than a regular CFG such as degrees of a node, size of a function, addresses of jumps, calls as well as the beginning of the function itself. These are detailed later in this Chapter. The function calls within the code section of the executable is given in Figure 3.2., Figure 3.3. contains the imported library and API functions together in one graph.

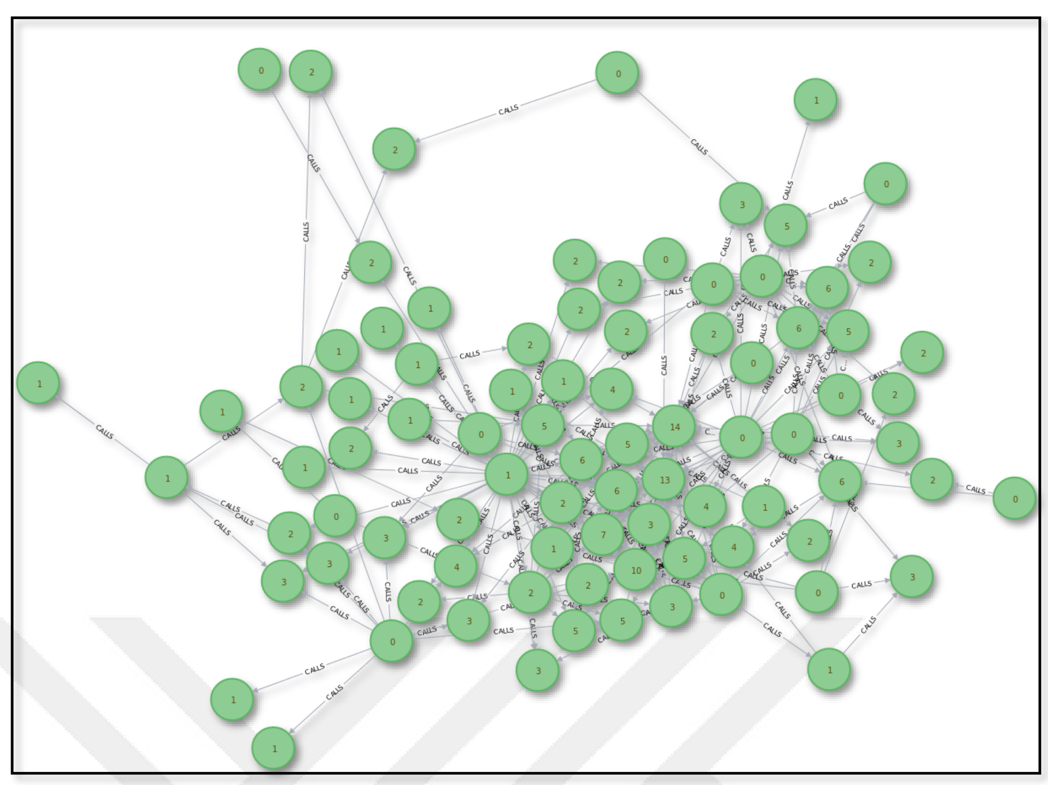


Figure 3.2 Graph Database Output of a CFG of the Virus Xpaj.C with hash 95cb89de13b078839bf8c56549de1fbf99a73dd8179d150d2cd975722e9f70e5

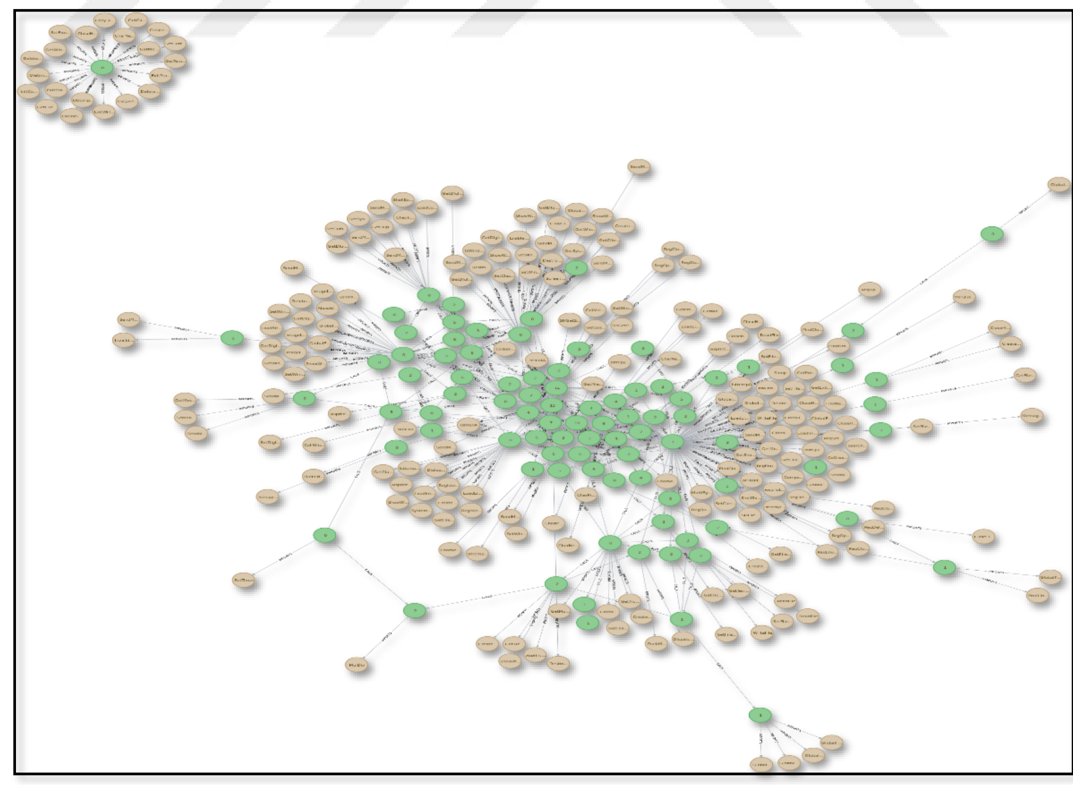


Figure 3.3 Function-Imported and API Functions illustrated Together of the same executable in Figure 2.

3.2. The literature on Static Malware Analysis and Machine Learning Methods

The problem of detecting new malicious executables is not new. Majority of the computer security community have identified this problem as a classification problem where the classes are *benign* and *malicious*. Another problem is that the community tries to tackle is the problem of identifying the family of a malware which has been reduced to a multi-class classification problem.

With the success of data mining techniques on Intrusion Detection and Prevention Systems (IDPs), the first research on malware detection is conducted in 2001 by Schultz et al. (Schultz, Eskin, Zadok, & Stolfo, 2001). This paper discusses the significant limitations of signature detection methodology and joins the data mining and machine learning approaches with the static analysis techniques that are mainly based on DLL and imported functions extraction.

Next, in the paper of (Christodorescu & Jha, 2003), a malicious code checking algorithm based on CFGs is defined. This research mainly attacks the problem of detection of obfuscated malware; a solution is generated with the annotation of obfuscated CFGs, and an algorithm is developed for checking the maliciousness. Another significant milestone in the literature is the introduction of graph isomorphism and utilization of graph algorithms in similarity checking. In the work of Bruschi et al., graph isomorphism is used for discovering self-mutating malware (Bruschi, Martignoni, & Monga, 2006). The research is based on the intuition that self-mutating malware will result in isomorphic malware, and by converting the malware into a graph, similarities can be reported with the detection of isomorphisms.

Machine learning algorithms are researched in the same year in the work of Kolter and Maloof; the bytecodes of the executables are vectorized into fixed-size vectors by n-gram methodology, and learning techniques are applied. Another research on learning technique called Objective Oriented Association (OOA) which is based on API sequences (Ye, Wang, Li, & Ye, 2007). This study generates rules based on the API sequences extracted by the static analysis and reaches 93 percent of accuracy in detecting the malicious executables.

Extracting opcode sequences by disassembling malware is another technique that is heavily used in the literature. This technique has been researched by learning

algorithms in the works of Santos et al. and Shabtai et al. (Santos, Brezo, Ugarte-Pedrero, & Bringas, 2013; Shabtai, Moskovitch, Feher, Dolev, & Elovici, 2012). N-grams, KNN, RF, SVM, Naïve Bayes, K2, Hill Climber, TAN learning algorithms are used in these researches and 91, and the results of these studies have reached 95 percentages of accuracy.

Static analysis methodologies are investigated and tried with Deep Learning methodologies as well. Convolutional Neural Networks (CNN) which is the state-of-the-art Deep Learning technique on images and matrices have been investigated with malware classification in the works of (Karbab, Debbabi, Derhab, & Mouheb, 2018; Ni, Qian, & Zhang, 2018). In the paper Maldozer, API sequences extracted from mobile malware are converted into matrices and fed into a CNN classifier; resulted in 98 percent of accuracy. Another work exploiting CNN's success in image classification; converting the malware hashes into images and researched the CNN on these images had resulted in approximately 99 percentages of accuracy.

Classifying and analyzing the malicious codes based on the data from the static analysis are summarized in Table 3.4. Some of the research is attacking the problem of detection while some focuses on obfuscation, polymorphism, and self-mutation. The aims of the studies and the features that have been used are given in the Features column. The technique that has been invented or incorporated is provided in the Used Technique column, and the results in the formats of classification accuracy (accuracy), false positive (FP) and false-negative (FN) rates are given in the last column.

In this thesis, a version of CNN is applied on Function-API call graphs to show the applicability of our graph database approach with deep learning. Function Call Graphs reveal an essential characteristic for the code to be analyzed, whether it is malicious or not. This research also integrates this vital data into account as the branching in the execution provides a backbone structure for the malicious code. Another significant characteristic is the API calls of the malicious code as it reveals the intention of the analyzed code by showing the interaction with the underlying operating system. Although the main focus on this thesis is extracting characteristics from malware analysis instead of the exploitation of deep learning on these results; the applicability of these images on a Deep Learning methodology is discussed in this Chapter.

Table 3.4. Comparison of Researches that utilizes Static Analysis Techniques.

Reference Work	Dataset	Features	Used Technique	Success Rate /Classification Accuracy
(Schultz et al., 2001)	3,265 malware and 1,001 benign programs.	Function Calls, DLLs, Opcodes, Strings	Naïve Bayes, Ripper, Multi-Naïve Bayes	97,11% Naïve Bayes Accuracy on Strings
SAFE (Christodorescu & Jha, 2003)	10 Obfuscated Viruses	Annotated Function Call Graphs of Obfuscated Executables	Malicious Code Checking Algorithm	FN and FP rates are 0
(Bruschi et al., 2006)	115 samples of Metaphor Virus	Detection of Self-mutated malware on Control Flow Graphs.	Graph Isomorphism	70% of equivalent viruses and %100 of different software.
(Kolter & Maloof, 2006)	1,971 benign and 1,651 malicious executables	n-grams	Naive Bayes, decision trees, SVM, and boosting	TP 98%., FP 0.05%
IMDS (Ye et al., 2007)	12214 benign and 17366 malicious samples	Objective Oriented Association (OOA)	Rule Mining Based	93% Accuracy
(Shabtai et al., 2012)	7,688 malicious and 22,735benign files.	Opcode Sequences represented as n-grams	n-grams	91% Accuracy with DF Classifier with n=2
(Santos et al., 2013)	17,000 malware, 585 malware families	Opcode Sequences	KNN, RF, SVM, Naïve Bayes, K2, Hill Climber, TAN	95.90% Accuracy with SVM
Maldozer(Karbab et al., 2018)	Malgenome, Drebin and Maldozer set of 33k malicious apps.	API Call Sequences	Convolutional Neural Networks (CNN)	Over 98% Accuracy on datasets and Over 99% on Family Classification.
MCSC(Ni et al., 2018)	10,805 samples	Hash results are converted into visual images.	Hashing and CNN	99.260% Accuracy

3.3. Tool support for Static Malware Analysis and Visualizations.

The construction of the graph database is accomplished through the reverse engineering framework Radare2 version 2.3.0 (Sergi Alvarez, 2006). All the functions are analyzed, and all the calls and imports are retrieved by automation of this tool written in Python scripts given by the open-source project R2graphity (GDATAAdvancedAnalytics, 2016). The extracted information inserted into the graph database instance created on the engine of Neo4j version 3.5.6. The 2d and 3d visualizations are designed with the 3d-force JavaScript library (Asturiano, n.d.).

3.4. A Graph Database approach for Static Analysis

The static analysis of the malware constructs the graph database. This analysis includes the complete disassembling and matching the calls that are made to the code section of the malware. All the functions are analyzed to the parameters of *address, size, API references, in-degrees (how many times the function is called) and out-degrees (how many times the functions call another function)*. These properties then converted into the queries of the graph database. API references are also inserted as nodes in this graph database with the address of the calling instruction stored as relationships. General schema of the database is in Figure 3.4, and information on nodes and relationships can be found in the following Table 3.5.

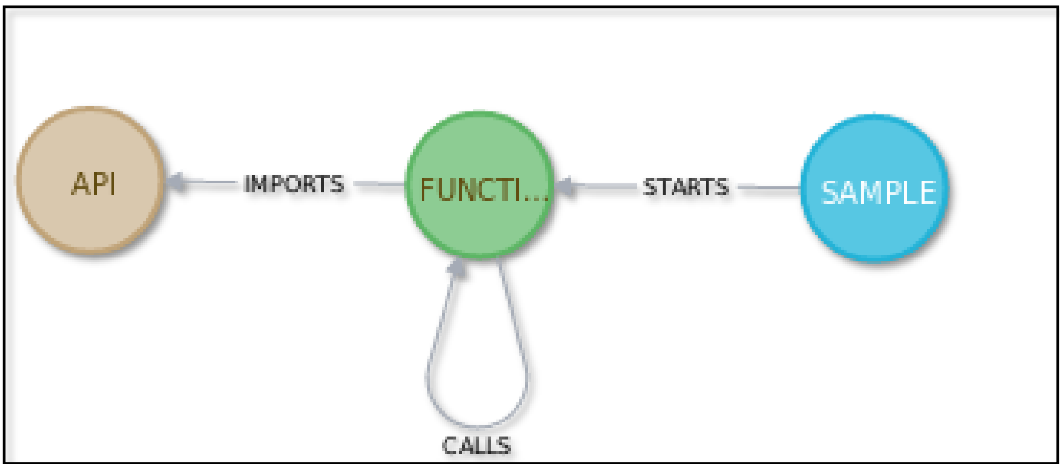


Figure 3.4 - The schema of the Graph Database.

Table 3.5 - Properties of the Nodes of the Graph Database.

Node/Relationship Label	Properties
SAMPLE	Sha1 hash, File Name, Section Count, Binary Type, Compilation Date, File Size, Section Entry Point, Original File Name, Entry Point Address.
FUNCTION	In degree, Out degree, Number of API calls, Function Size, Call Type, Function Type.
API	API name
CALLS	Distance, Calling Instructions Address
IMPORTS	Calling Instructions Address
STARTS	<none>

Some of these properties play a significant role in the visualization process, which is discussed in the Visualizations part of this Chapter. The distance in functions is calculated as the absolute value difference in addresses between functions. The addresses of calling instructions are the relative addresses of the instruction that made the API or the function call and not presented as an offset notation. The function size is the size of the function in the code segment, the entry point is the entry point provided by the static analysis, and in many cases, it is an obfuscated value showing some function other than actual main function of the malware.

The process of inserting static analysis results is quite straightforward. First, the static analysis of the sample is done, and a SAMPLE node is created. All the functions of the sample are extracted and inserted as FUNCTION nodes. Thereafter, all the cross-references captured by the static analysis are iterated and matched with API names and functions and inserted as API nodes. The flowchart of the construction of the graph database is illustrated in Figure 3.5 below.

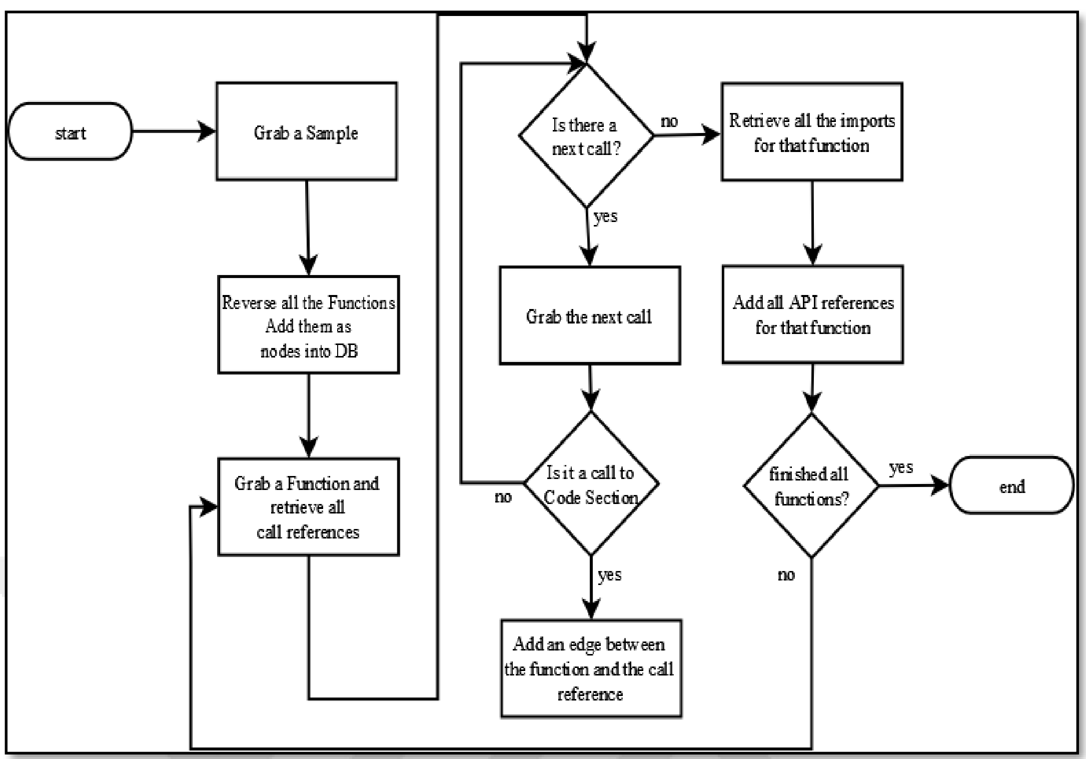


Figure 3.5- Insertion process of the Static Analysis Results.

3.5. Visualizations of Static Analysis

In this part of the thesis, the visualization methodology developed for static analysis and for the aim of producing fingerprints is presented. This fingerprinting technique is formed in the hope of being helpful to a malware analyst to provide a general idea about the analyzed malicious sample rapidly.

In Figure 3.6, the round nodes represent functions, while the square nodes present API calls or Imports. As can be seen in Figure 3.6, the node size of the function nodes on visualizations varies according to their actual sizes. Moreover, the distances between function nodes are scaled from the exact sizes the functions have from the graph database. However, as many operating systems integrate memory address randomization schemas such as Address Space Layout Randomization (ASLR), the size of and the distances to API calls are ignored in this visualization as these measures may provide inconsistent results due to this randomization process.

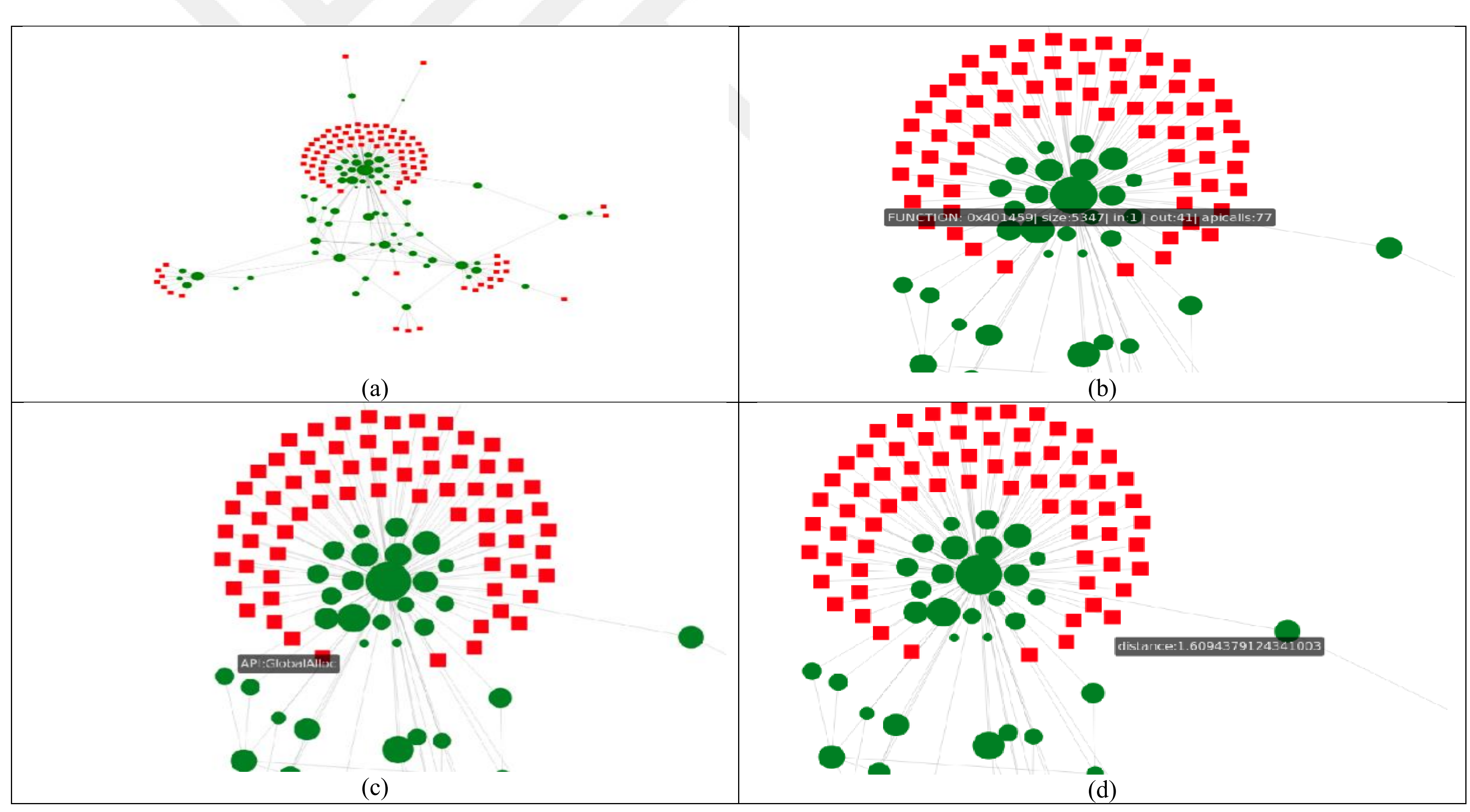


Figure 3.6 - The visualization of the malware sample with Sha1 33e8e894297e0f94c5df36cb4e5b3ee68662ceff (a) An overview of the malware (b) Hovering on a function node (c) Hovering on an API node (d) Hovering on an edge between functions.

For this research, a 3-dimensional visualization model is also presented. The 3d visualization is analog to the 2d model and can be seen in Figure 3.7 below.

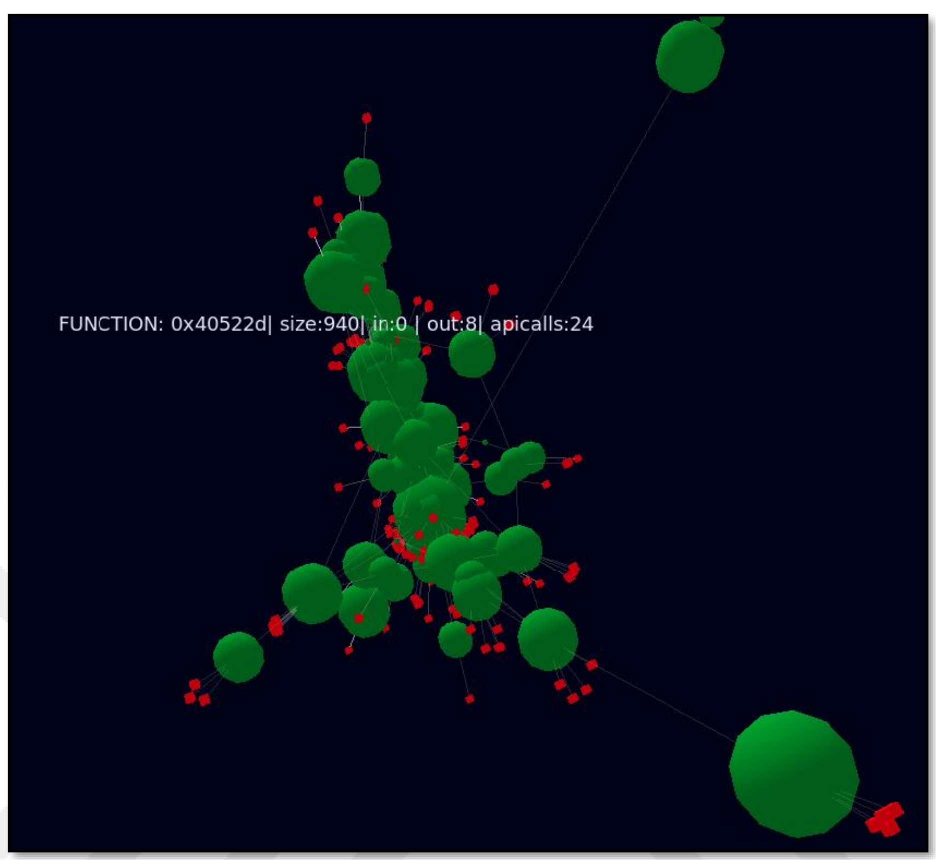


Figure 3.7 - 3d model of the same sample in Figure 6.

3.6. Limitations of Static Analysis

The limits to the static analysis and its results are given in the work of (Moser, Kruegel, & Kirda, 2007) through a custom designed obfuscator that is available on the binary without having the source code itself. In this work, the polymorphism, metamorphism, and obfuscation properties of the malicious executables are discussed, and a binary obfuscation approach is presented to show the insufficiency of the static analysis alone in detection malware. The fundamental limits of obfuscation are also provided in a trade-off with performance. In this part of Chapter 3, it is aimed to show the limitations of the static analysis by examples from the designed and deployed graph database.

The following Figure 3.8 presents four different types of anti-static-analysis techniques. In Figure 3.8a, it is possible to see unreachable and dead functions that are designed to evade signature level detection. Figure 3.8b contains API calls related to

the cryptography scheme and virtual memory allocations; while Figure 3.8c has randomly generated, same-sized functions in the numbers of the order of 10, therefore showing the features of obfuscation. In Figure 3.8d, only the unpacking functions are present since a kind of packing is applied.

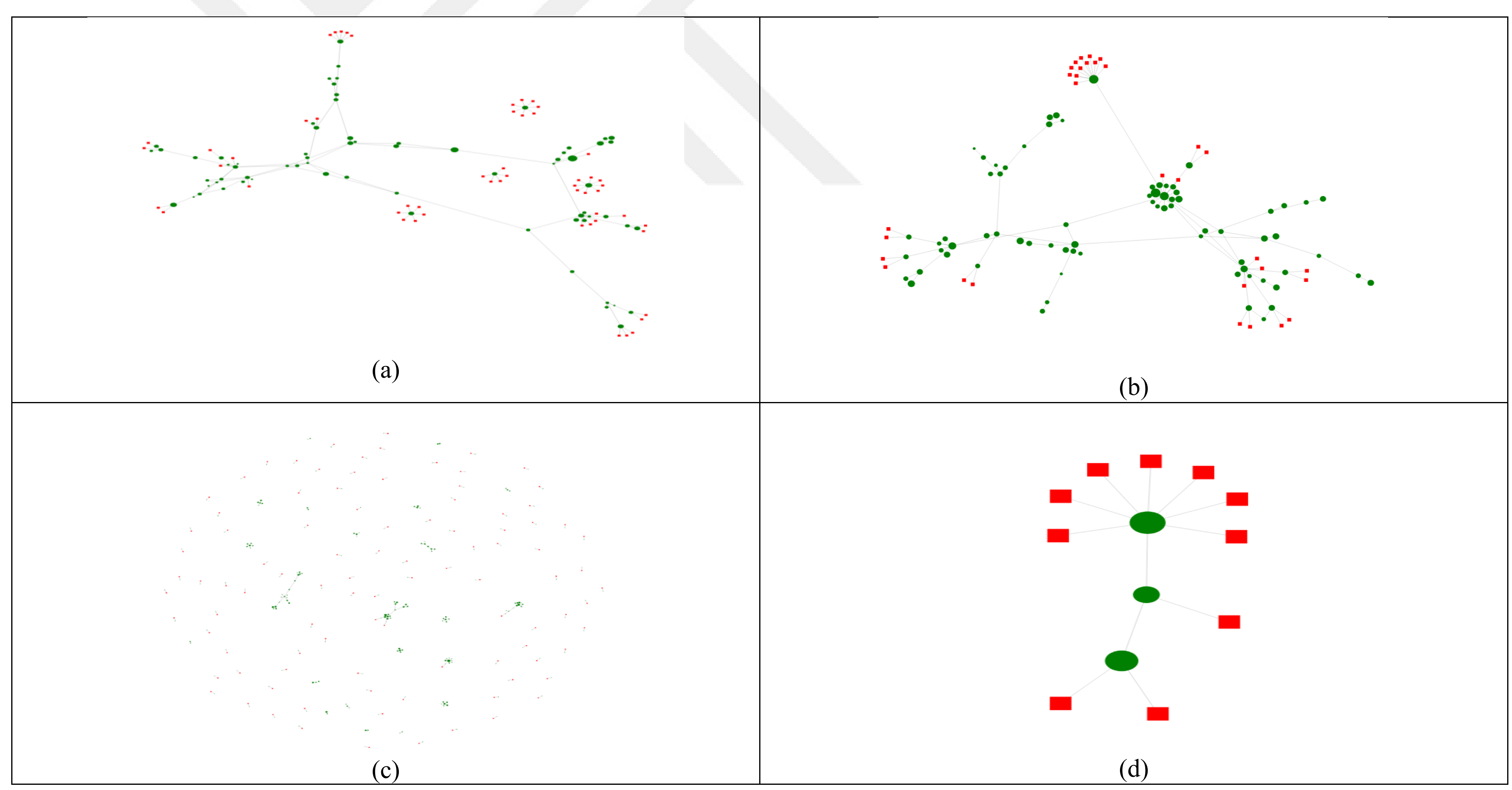


Figure 3.8 The visualization of the malware samples with Sha1 0f241d84aa44034c924197d3bce94faa07811f35, f53e68832af99cf553471cf87cc5da332c695659, ada9efdf8dee612599377f6ade3e78e06d4069f4, a9accc4fe6cd45b9a54c25a1447ed74cc61d5675 respectively, showing (a) dead code insertion (b) encryption engine (c) obfuscation (d) packing.

3.7. Convolutional Neural Networks (CNN) on Static Images

CNN's, are a particular type of neural network for implementing a matrix or grid-like data. It utilizes a specific operator, *convolution* operator, which is specialized kind of a linear operator and a CNN is a neural network that uses at least one convolution layer in its layers (Lecun & Bengio, 1995).

In this research, a CNN network with three times two convolutional ReLU layers plus a max-pooling layer, and a SoftMax layer as output have been tried. However, because of the aforementioned limitations, no satisfying result could be achieved. Although with a fine-tuned CNN approaches worked well on the examples and studies in the literature, as the focal point of this thesis is to extract analysis data particularly on memory, the continuation of Deep Learning approaches is left as future work. In the following chapters, a successful training attempt on the memory patterns is presented in both Machine Learning and Deep Learning methods.

3.8. Conclusion of Chapter 3

As shown in this chapter, static analysis has some serious drawbacks when applied for detection. The evasion techniques mentioned in this chapter are commonly used in contemporary malware, and therefore, in many cases, the methods based on only static techniques seem to be failing. One useful methodology in this context would be stripping malware from all the static evasion techniques, extracting the executable from the memory address space and applying static analysis after the extraction. However, this technique will also suffer from randomization techniques employed by the malware authors.

In the following chapters, the dynamic analysis and memory forensics techniques are discussed and evaluated before presenting the novel methodology used in this thesis.

CHAPTER 4
DYNAMIC ANALYSIS OF MALWARE

Dynamic analysis techniques involve running the malware instance in an isolated environment and extracting characteristics of the instance by monitoring it. In the dynamical analysis, the malware is analyzed red-handed during the execution, and the analysis system becomes robust to any obfuscation, encryption or alternation to the original code. This is the main advantage of dynamic analysis compared to the static analysis. However, every instance of the malware should be run separately which would result in a computational overhead which is the main disadvantage (Shijo & Salim, 2015; Yadegari, Stephens, & Debray, 2017).

In this chapter, the methods of Dynamic Analysis are presented, the methodologies are exampled and visualized; it's relevance and pitfalls are discussed. Moreover, a dynamic analysis method, Dynamic Binary Instrumentation, which constructs one of the building blocks of this thesis, is introduced in this Chapter.

4.1. Dynamic Analysis Techniques

A *sandbox* system is a secure dedicated or virtual system for running and testing unknown executables. Sandbox systems employ a simulating environment similar to or identical to the system that is to be protected. The idea is to detect and monitor the effects of the unknown executable on the simulating system without compromising the actual hosts and users.

The sandbox systems are based on the architecture of a client-server model where the simulating system acts as a client, and analysis machine is the host. The client's APIs and kernel functions are hooked and monitored to collect relevant information about the binary that runs on the client, and communication between analysis machine and client is established either by virtual local networks or by the API functions of the virtualization software.

Most of the sandboxing solutions are closed-proprietary software as of today. However, there are powerful open-source alternatives such as Cuckoo Sandbox (Cuckoo Sandbox, 2019), Joe's Sandbox (Joe Sandbox, 2019) and Zerowine (Zerowine, 2019). These automated sandboxing solutions construct the basics of dynamic binary analysis and provide a rapid and easy method to analyze various features of the binary.

The dynamic analysis makes use of many changes and differentiation within the system. In this part, these indicators are presented and explained.

Registry Snapshots

Registry analysis is a significant indicator for malware as malware needs persistence in the infected system. The persistence of malware is the property that the malware will stay dormant on the compromised computers until some triggering event or some predefined time for the attack to start. In such cases, the malware needs to hide in the system, should be restart resilient and should be reachable to the C&C. Therefore, malware should place the necessary indicators in the auto-start locations of the registry, scheduled tasks, and cronjob events.

Another significant contemporary threat is a new malware type file-less malware. These types of malware reside as scheduled tasks in the registry, retrieves itself every item an event is triggered and run on memory without any filesystem indicators. Because of these reasons, the analysis of the registry holds paramount importance for the dynamic analysis of the sandbox.

API Call Sequences

The API calls are the user-space requests of kernel operations of the operating system. The execution of malware needs various types of API calls such as registry operations, file operations, and virtual memory operations. In most cases, if not evaded, these calls show the actual intention of the malware. However, as the malware authors become aware of these analysis techniques, they are implementing evasion procedures such as code injection, native DLL coding and target process obfuscation (Kawakoya, Iwamura, Shioji, & Hariu, 2013).

Memory Forensics

The dynamic binary analysis integrates the memory forensics procedures and practices with its analysis methodologies. These details are discussed in Chapter 5.

Dynamic Taint Analysis

In dynamic taint analysis, the data originating from or arithmetically derived from untrusted sources such as the network is referred to as tainted. These tainted resources are followed during the execution to detect buffer overwrite attacks. The method is first introduced in (Newsome & Song, 2005).

Dynamic taint analysis (DTA) is utilized for analyzing execution paths that an attacker may use to exploit a system. DTA is studied extensively in the literature: TaintCheck(Newsome and Song, 2005), Dytan (Clause, Li, and Orso, 2007), BitBlaze, DTA++ (Gyung et al., 2011) and SworDTA (Cai et al., 2016).

Logging Network Attempts

Another result of the dynamic analysis of malware is its network connections. With utilizing a network generator sandbox server and attaching the analysis machine as a client, all the network traffic can be diverted over the sandbox server, and the network dump data (such as TCPDUMP) can be collected.

Dynamic Binary Instrumentation (DBI)

DBI is a technique of Dynamic Binary Analysis. It requires an analysis program working on the side with the actual process that is to be instrumented. The analysis program is injected as a DLL or a kernel process to the analyzed system (Kawakoya et al., 2013). With the analysis code inserted into the execution of the target program, instruction level, API level, Stack and Heap level analysis can be done through DBI systems. This thesis covers and utilizes DBI for capturing memory operations of a binary. The methodology and used tools are being included in the next Chapter.

4.2. The literature of Dynamic Malware Analysis

The effectiveness of Dynamic Analysis has been tested rigorously in the malware analysis literature. Machine Learning, Deep Learning, and Graph Matching methods have been applied on API call sequences and graphs, hooking behaviors, kernel-level

executions, memory usage, imported DLLs, and network communications. Table 1 below summarizes the building blocks of the malware analysis literature.

The first study on dynamic malware analysis is the study of (Yin, Song, Egele, Kruegel, & Kirda, 2007). In this work, a particular dynamical analysis technique called taint analysis is used. In dynamic taint analysis, the data originating from or arithmetically derived from untrusted sources such as the network is referred to as tainted. These tainted resources are followed during the execution to detect buffer overwrite attacks.

In the study of (Bailey et al., 2007), dynamic analysis results are collected as event logs and converted into non-transient state changes. These state changes are converted into trees, and the distances between samples are researched. Similarly, in the work of Kolbitsch et al., taint analysis results are converted into behavioral graphs, and a similar rate of success has been achieved with subgraph matching (Kolbitsch et al., 2009). An extension to Kolbitsch et al. is the dependency graph study of (Park, Reeves, & Stamp, 2013), where dynamic system calls are converted into graphs and tested similarly. The results of this work showed a 100 percent success rate for some of the malware families.

Code slicing methodologies are integrated with the extraction of API call sequences in the work of (Lanzi, Sharif, & Lee, 2009), The aim is to extract and use kernel-level operations within the malicious executable, and the data access patterns and data modifications using these calls are comprehended. A similar idea is presented in the work of (Park et al., 2013) where the behavioral indicators are constructed as a graph, and instead of slicing the code and analyzing the flow, the graph data is clustered. The results of this work showed zero false positives.

The tests on API call data is also extended on the machine learning and deep learning subjects. N-gram technique is used in two significant studies in the literature; Uppal et al. utilize Naïve Bayes, Random Forests, SVM and Decision Tree Classifiers on the call sequence n-grams; Kolosnjaji et al. use the same feature on a Deep Learning approach with Convolutional Neural Networks. The results of these two studies are shown in Table 1.

Another study on API calls converted into matrices, utilizes Random Forests on the matrix data by (Pirscoveanu et al., 2015) and a similar research with unsupervised

learning methods presented in the same year, (Fujino, Murakami, & Mori, 2015) utilizing Term Frequency- Inverse Document Frequency (TF-IDF) matrices.

Finally, a research paper on Android operating systems malware is presented to extract TCP/IP features for testing in various machine learning algorithms is presented in Table 1 (Narudin, Feizollah, Anuar, & Gani, 2016).

Table 4.1. Comparison of Researches that utilizes Dynamic Analysis Techniques.

Reference Work	Dataset	Features	Used Technique	Success Rate /Classification Accuracy
Panorama (Yin et al., 2007)	42 malware and 56 benign samples	Taint Graphs	Policy generation based on Taint Graphs	Around 3% FP rate.
(Bailey et al., 2007)	3698 Samples	Non-transient State Changes	Normalized Fingerprint Distances	91.6% detection rate
(Kolbitsch et al., 2009)	Six malware families with 50 samples each	Behavior graphs extracted from taint analysis and program slicing	Subgraph matching	90% at maximum for known malware, 23% for an unknown malware
K-Tracer(Lanzi et al., 2009)	8 Rootkits	Data Access, Triggers, Hardware Events	Dynamic Slicing	Detects all the rootkits that have been tested with it.
(Park, Reeves, Mulukutla, & Sundaravel, 2010)	Six malware families with 50 samples each	Dynamic system Call Dependence Graphs	Graph Similarity Measurement	Some of the families showed 100% accuracy, while some have poorer results.
(Park et al., 2013)	563 and 520 samples in two datasets	HotPath (constructed by kernel objects and system call traces) extraction	Graph Clustering and Matching	No false positives.
(Uppal, Sinha, Mehra, & Jain, 2014)	120 malicious and 150 benign software	Call-grams generated from call sequences	Naïve Bayes, Random Forests, SVM and Decision Tree Classifiers	Accuracy of 98.5%
(Pirscoveanu et al., 2015)	42,000 malware samples	API Call Matrices	Random Forests	89.6% TP and 0.049 FP rates.
(Narudin et al., 2016)	Android Malgenome (1260 Malicious apps)	TCP/IP packages	RF, J48, MLP, Bayesian Net, KNN	99,97% Accuracy with BN and RF
(Kolosnjaji, Zarras, Webster, & Eckert, 2016)	4753 malware samples	Malware System Call Sequences	Sequence-grams on feed-forward, convolutional and hybrid neural networks	85.6% on precision and 89.4% on recall
(Pektaş & Acarman, 2017)	17,900 recent malign codes	API-call sequences	N-grams	Training and Testing Accuracy of 94% and 92.5%

4.3. Tool support for Dynamic Malware Analysis.

The dynamic binary instrumentation system is provided by the Intel PIN Tool version 3.7. Intel pin tool is a dynamic binary instrumentation framework where a process state containing registers, memory, heaps, stacks, memory access, and the execution flow can be traced and analyzed (Luk et al., 2005). In this API, the instrumentation is accomplished by the just-in-time (JIT) compiler. Just-in-time compiler in this API, as distinct from its Java version de facto standard, takes in a native opcode instead of bytecode and observes and generates the native opcode for the executable (Luk et al., 2005). The architectural structure of the Pin Tool is illustrated in Figure 1.

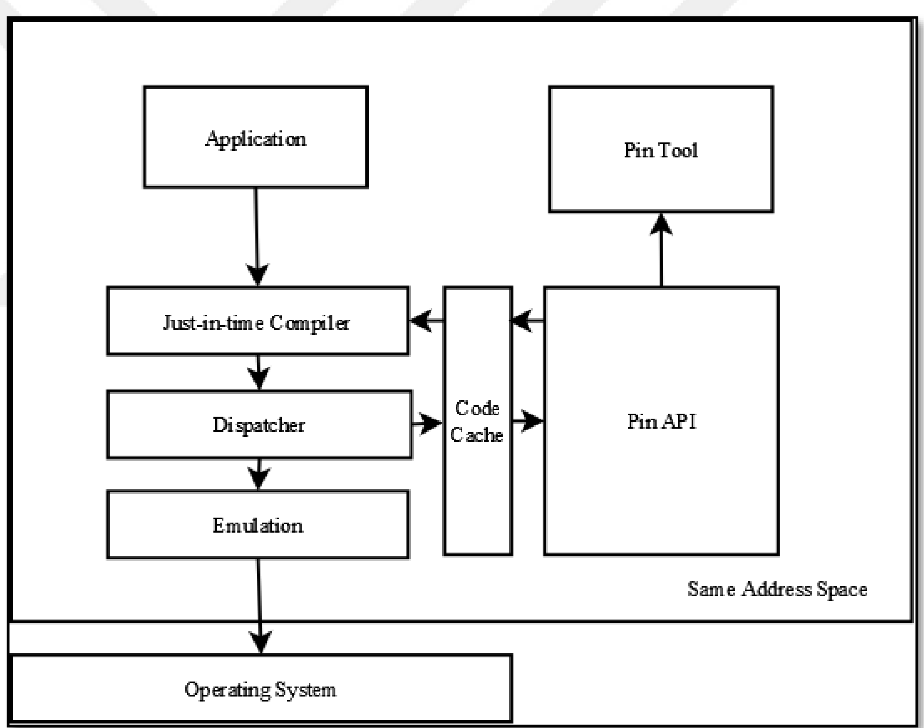


Figure 4.1 Architecture of Pin Tool

As can be seen from Figure 4.1, the application to be instrumented, and the pin tool shares the same address space. The instrumentation application is written as a DLL for this library using the Pin API. The instrumented and traced opcodes are cached in the Code Cache as the execution flows through the Just-in-time compiler and the emulated native opcodes are then forwarded to the Operating System underneath.

4.4. A Showcase of Dynamic Analysis with Intel PIN Tool.

For the demonstration purposes, dynamic traces and indicators are extracted from two malware samples. These samples are selected from the family of Worm.Ramnit malware family and hash values are given in the following Table 4.2.

Table 4.2. Sha1 hashes and compilation information of the malware samples

	Malware Information

Ramnit Worm v1	{
	"sha1": "33e8e894297e0f94c5df36cb4e5b3ee68662ceff",
	"fname": "Worm.ramnit.9a08d9b7853a65fb52f119806b2f3aae.exe",
	"sectionCount": 5,
	"binType": "PE32 executable (GUI) Intel 80386, for MS Windows, Nullsoft Installer self-extracting archive",
	"compilation": "2009-06-18 21:33:23",
	"fileSize": 18430835,
	"sectionEp": ".text|0",
	"originalFilename": "",
	"addressEp": 12577
	}

Ramnit Worm v2	{
	"sha1": "8293f7ddbb7a6163aafed7ebeaea9bc5d60716fb",
	"fname": "Worm.ramnit.9ad7b41a1f0bee2112c1b497094aa085.exe",
	"sectionCount": 5,
	"binType": "PE32 executable (GUI) Intel 80386, for MS Windows, InstallShield self-extracting archive",
	"compilation": "2009-12-05 22:50:46",
	"fileSize": 3363317,
	"sectionEp": ".text|0",
	"originalFilename": "",
	"addressEp": 12860
	}

The execution flow and its indications on the memory address space is the main interest of this research, and the samples are analyzed and dissected according to their instruction flow. As a result, three indicators are extracted for the two samples: Function Hit Trace, Function Trace, and Instruction Trace. For the demonstration, the static analysis features of Function and API relationship graphs are extracted and presented in Figure 4.2, as shown in Chapter 3.

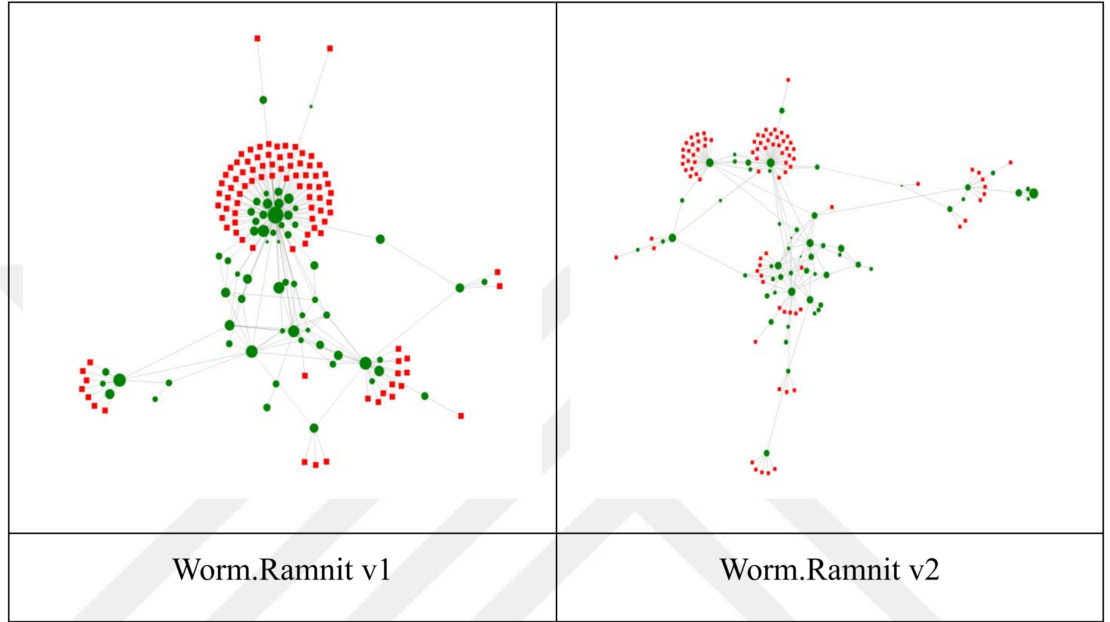


Figure 4.2. Static Analysis Graphs of the samples.

Function Hit Trace and Function Trace

The function hit trace and function trace are the features of the execution flow. These properties are extracted by following the instruction pointer and recording the functions that the instruction belongs as the execution continues. The recordings are then converted to 2d images where the x-axis shows the sequence of the instruction and the y-axis shows the memory address of the function.

- Function Hit Trace is the analysis where a function is recorded only the first time that the instruction pointer is inside the memory space of that function (Luk et al., 2005).
- Function Trace is the analysis where a function is recorded every time the instruction pointer enters the address space of the function (Luk et al., 2005).

For the two samples, the function trace and the function hit trace graphs are given in the following Figure 4.3.

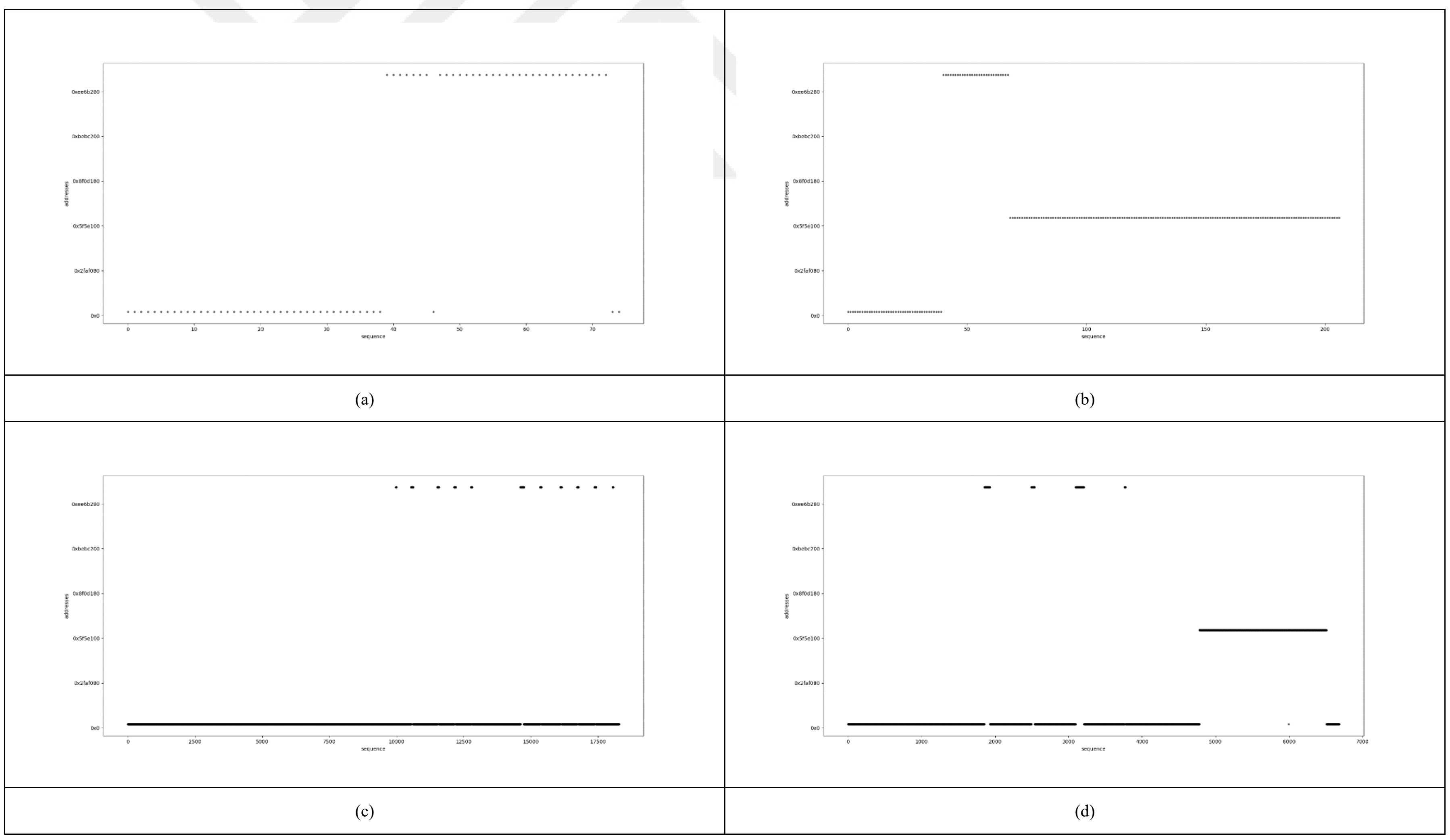
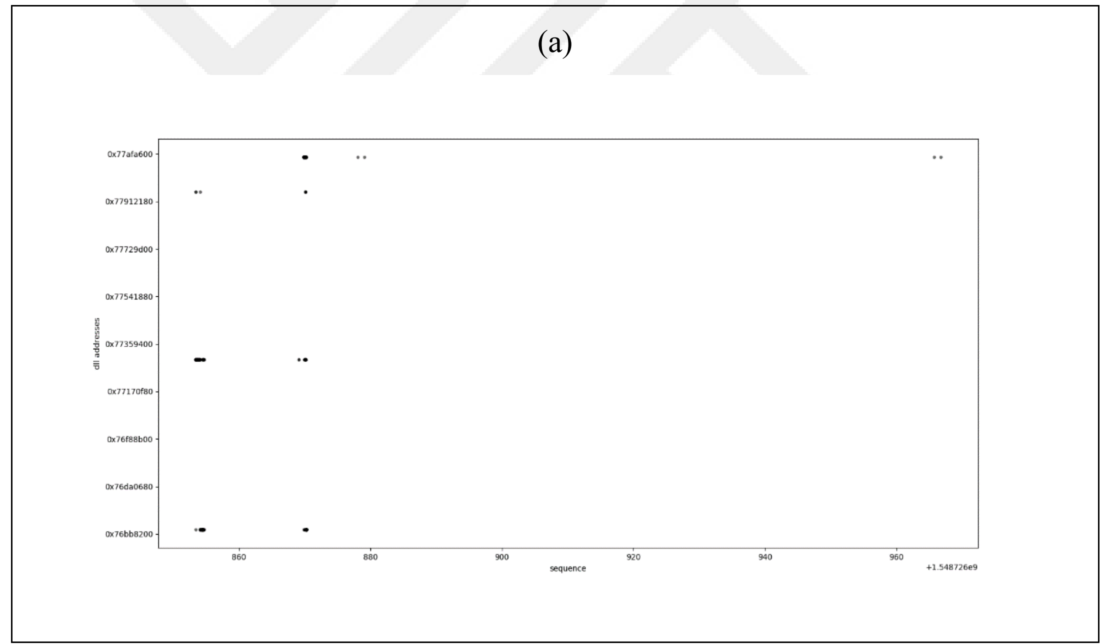


Figure 4.3. Function Hit Traces(a), (b) and Function Traces (c), (d) of Ramnit v1 and v2 respectively.

Function and Function Hit Traces are beneficial when debugging a known function for bug fixing. Moreover, these measures can be used to un-obfuscate function obfuscation and sieve the functions that are used and unused in the code space. For unpacked malware, this measure can be advantageous when used with static analysis. Execution flow can be understood and taken into malware analysis with the identification of all the functions. Function Hit Traces can replace Function Trace when the analyzed malware contains recursive functions which can obfuscate the analysis.

API Call Sequences

Following a similar idea, memory address space access of the imported functions can also be traced using dynamic instrumentation. This feature is the memory representations of the API call sequence property that is being utilized in several works in the literature. The following Figure 4.4 illustrates this feature.



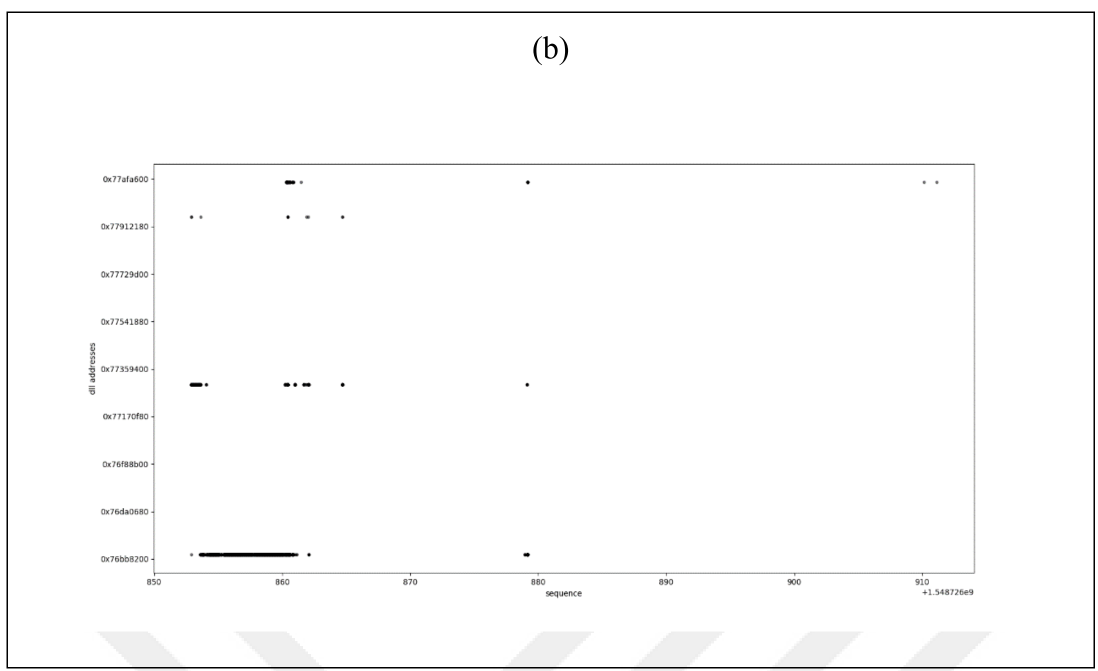


Figure 4.4. DLL Addresses and Sequence graphs of the Samples. (a) Ramnit v1 (b) Ramnit v2.

DLL Addresses and API imports have already been explored in the literature excessively. However, an address-wise representation of imported functions is provided in Figures 4.4 (a) -(b), and executed addresses of the imported calls and functions are shown similarities for this two malware. It is crucial to turn off the ASLR for this measure to have an implementable metric.

Instruction Address Trace

Instruction level tracing shows the traversal of the address space by the instruction pointer. For the vast amount of the processes that have been tested, this trace results in gigabytes of trace data thus become infeasible to graph and analyze. However, for the demonstration purpose, two graphs are generated for the first one thousand entry of these two sample traces.

As can be seen in Figure 4.5, the first 1000 instruction trace of the two malware can result in similarities when the two malware are stripped from any obfuscation. However, the resulting data of such trace for a malware reaches the order of 10 gigabytes thus become unfeasible quickly.

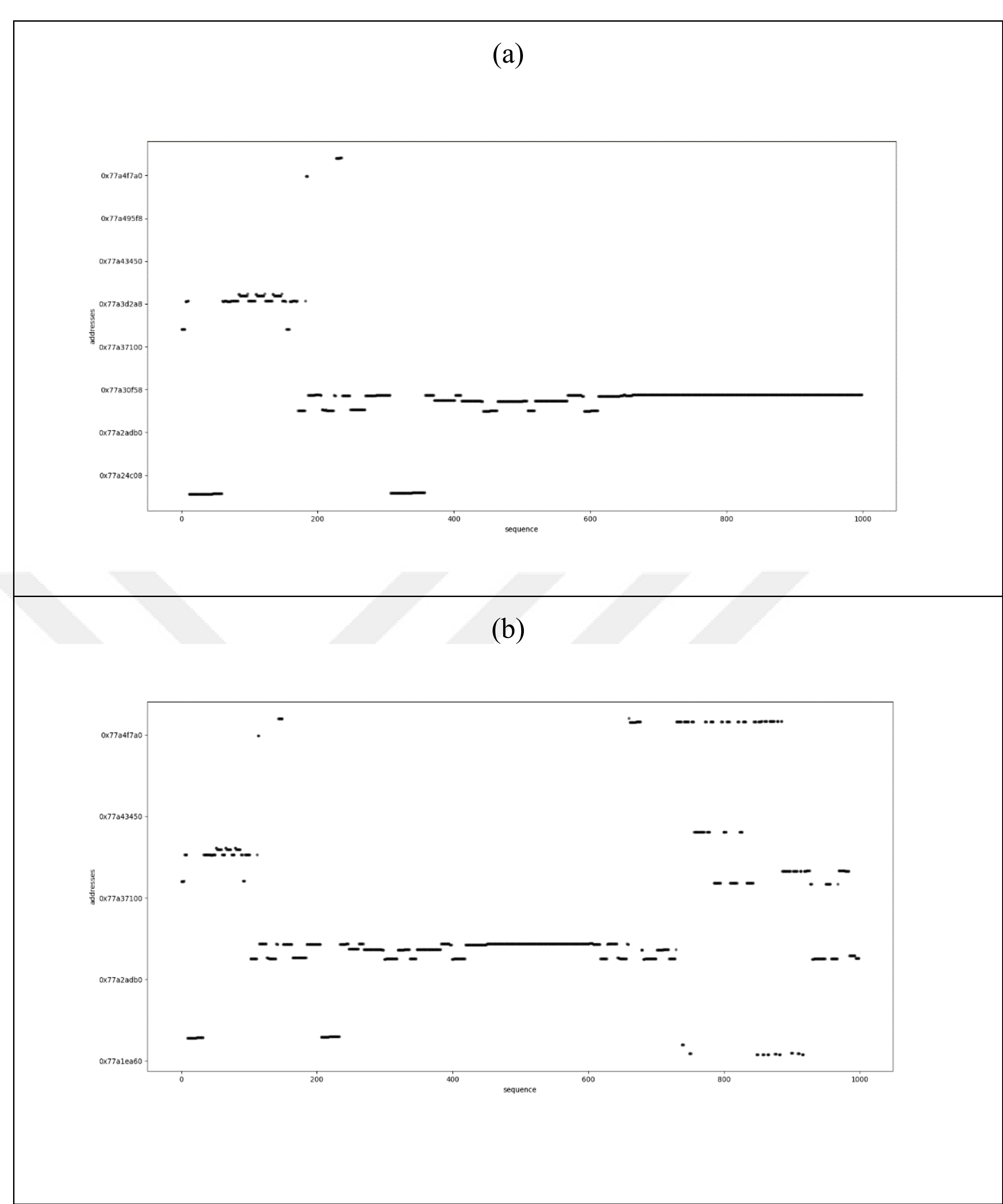


Figure 4.5. Instruction Pointer Traces of the Samples. (a) Ramnit v1 (b) Ramnit v2.

4.5. Conclusion and Discussion of Chapter 4.

The dynamic analysis techniques, the binary instrumentation, and the demonstration of these techniques are presented in Chapter 4. For the following chapters, the utilization of this binary instrumentation on extracting the memory access patterns and the generation of a graph database from this information and the results for this dissertation is provided.

CHAPTER 5
MALWARE MEMORY FORENSICS

As previously mentioned in Chapter 1, this dissertation aims to shed light on the memory access patterns of malware, to visualize these patterns and to compare and contrast the access patterns to identify similar characteristics on memory. For this aim, Chapter 5 identifies the contemporary memory operations of malware. These operations include packing, code injection, DLL injection, and process hollowing methods which malware integrates into their code to avoid being detected.

5.1. Memory Forensics

Memory Forensics is the process of acquiring, dissecting, and analyzing the volatile memory data for suspicious events and operations. The process of memory forensics provides a detailed description of the state of the computation for the time that the memory image is captured. Every operation on a computer either done by an operating system or a user application allocates itself on the memory. These operations are grouped by the data structures of the operating system as processes, and every process has its threads (at least one, the main thread) which are the smallest chunks of executable memory contents. The structure of a process is discussed in Chapter 2 for the Microsoft Windows operating system for which the majority of malware is designed for.

Memory Forensics provides significant data about the processes on the memory such as;

- How many processes are running on the system?
- What is the current state of the processes on the system?
- Which executable files are associated with the processes on the system?
- Which files are open currently? By whom they are opened?

- Which DLLs are loaded and by which processes?
- How many active network connections are there? Which processes are ported to network and at which ports?

Memory Forensics can answer all of these questions and many similar to them. It is the closest analysis method to in-vivo analysis while hard drive forensics is the post-mortem analysis for a system.

As mentioned in Chapter 3 and 4, contemporary malware utilize many techniques to evade from detection. Malware, when running on the memory, is in its exact form, stripped out of all obfuscations, packing and encryption. Therefore capturing the characteristics from memory is one of the best options for automated search mechanisms.

A recent, on-the-rise malware type is file-less malware. This type of malware works only on the memory, leaving no traces on the storage spaces and steals other process's address space to do their malicious work. This type of malware is impossible to analyze without taking memory images and investigating them. This type of malware and their techniques are discussed later on in this Chapter.

Another reason the memory is the actual key to understand malware is that the authors are now aware of malware analysis techniques, and they incorporate this knowledge into their works. The signature detection mechanisms depended on cryptographic hash algorithms to identify malware. However, polymorphic malware with mutation engines is capable of attacking this system stealthily. For that reason, current hash-based detection techniques involve a method called Fuzzy Hashing, which involves hashing of the parts of the malicious executable on memory and comparing/detecting partially (Li et al., 2015; Sarantinos, Benzaïd, Arabiat, & Al-Nemrat, 2016).

Moreover, contemporary malware exploits vulnerabilities in the user applications to rewrite the application code, reuse existing code in a way that is not intended to be used by changing the execution flow with Return Oriented Programming and Jump Oriented Programming methods (Korczynski & Yin, 2017). To detect the aforementioned on-memory strategies of malware leads to on-memory malware detection techniques to be developed, which are based on memory forensics. In the following section, these techniques are detailed.

5.2. Malware Operations on Memory

5.2.1. Packing and Compression

Packing is the method of compressing and/or encrypting the malicious components before infection. The sections of the executable are compressed into a data section, and an unpacking stub of code is inserted into the PE file. For some standard packers such as UPX, the section names can reveal if the binary is packer or not. Otherwise, the section name can be anything as it does not result in any changes in the execution of the program.

After the packed program runs on the memory of the target computer, it unpacks itself by allocating space from its or other processes address space. The DLL functions used for such operations are as follows:

- VirtualAlloc,
- VirtualAllocEx,
- VirtualFree,
- VirtualLock,
- VirtualProtect,
- VirtualQuery.

The function that is used for accessing DLLs and imports from within an address space are also listed below.

- LoadLibraryA
- LoadLibraryW
- LoadLibraryExA
- LoadLibraryExW
- FreeLibrary
- GetProcAddress

Extracting DLL functions of a binary and observing that the above functions are called in a library, does not necessarily specify the maliciousness of a program. However, in most of the studies mentioned in Chapter 4 – Dynamic Analysis of Malware extracts the data into their machine learning-deep learning approaches. In the learning approaches, if a value of a feature does not yield into differences in classification, then their weights become smaller, and their importance in the process of classification degrades. Our research which will be detailed in the next chapter considers these functions and features as it is for that matter.

The operation of packing is illustrated in Figure 5.1. below. The address space illustrated on the left is the version of the executable before it is packed. Moving to the right, the second image shows a packed version of the same executable and the image on the rightmost of the illustration shows the unpacked executable. The unpacked version of the executable is accessible from the moment the executable finishes unpacking itself to the end of the execution(Ligh, Case, Levy, & Walters, 2014).

Although it has been stressed in this dissertation, the unpacking operation can include modules that obfuscate the executable code which will produce different signatures for the leftmost and rightmost images of the executable (Ligh et al., 2014).

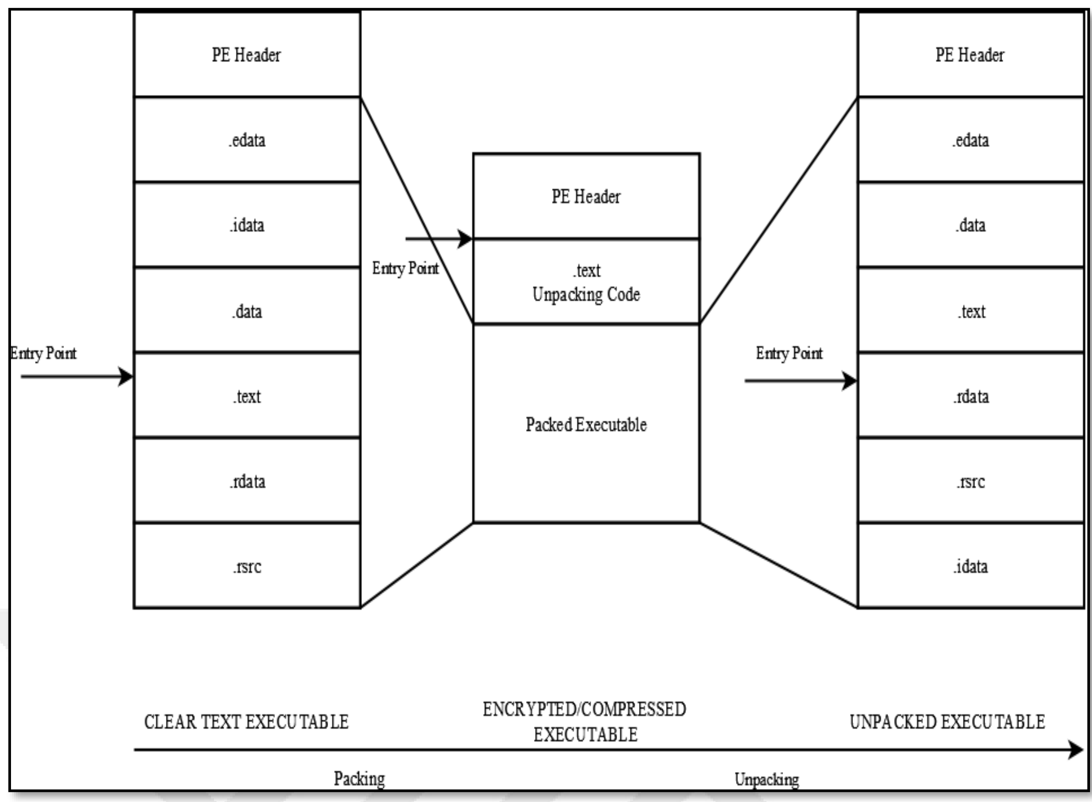


Figure 5.1. Illustration of packing of an executable.

5.2.2. Code Injection

Code injection is the process of copying malicious executable code payloads or malicious pe executables into another process address space and running the code from there. It requires the malware to have the debugging permissions to access another process address space.

The procedure of code injection is as follows and ill (Ligh et al., 2014):

1. Malware process acquires debugging privileges (SE_DEBUG_PRIVILAGE) that enables read and write access to another process address space.
2. Malware process opens the target process and receives its handle through OpenProcess() function.
3. Malware process allocates memory space using virtual memory functions with the PAGE_EXECUTE_READWRITE permission.

4. Malware transfers payload, shellcode, or a complete PE executable to the allocated address space using WriteProcessMemory().
5. The malware calls a CreateRemoteThread() and gives the address of the injected code to the thread.

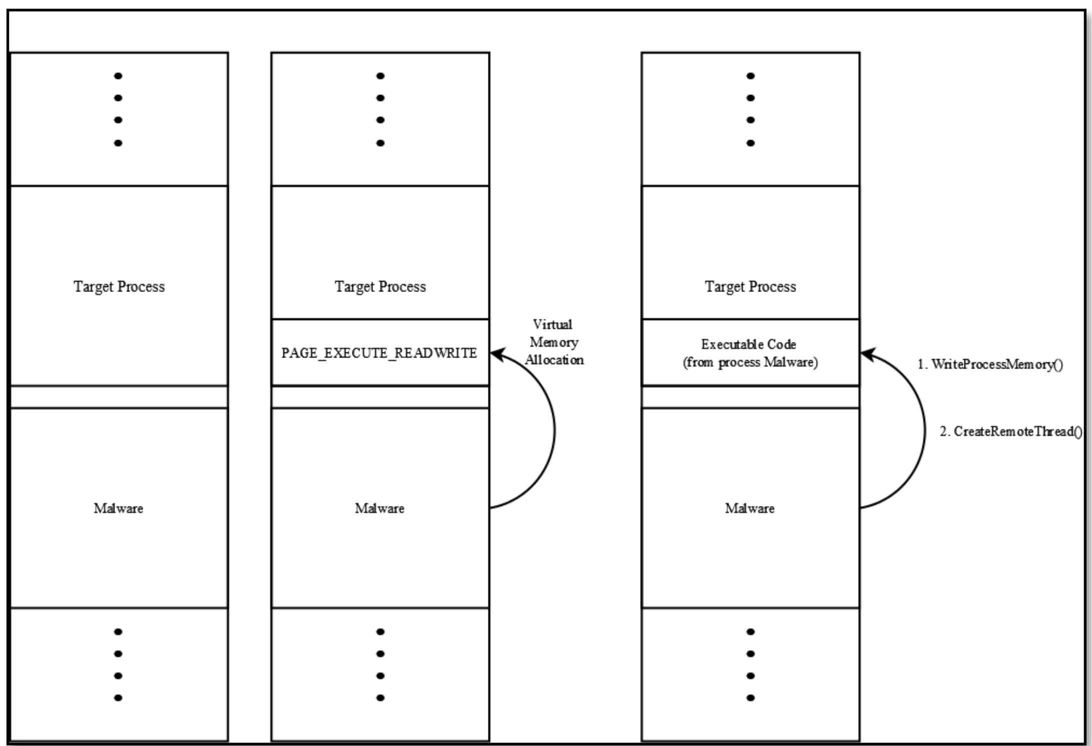


Figure 5.2. Illustration of Code Injection.

5.2.3. DLL Injection and Reflective DLL Injection

DLL injection is a similar approach to the Code Injection with some minor differences that effects the detection of malicious acts. In DLL Injection, the malicious code is loaded from disk to the target process address space using the LoadLibrary() method.

The allocated address space is not required to be executable in this method; instead, PAGE_READWRITE permissions are sufficient enough to insert a DLL to the target process. CreateRemoteThread() method is used again for running the DLLmain.This schema is illustrated in Figure 5.3.

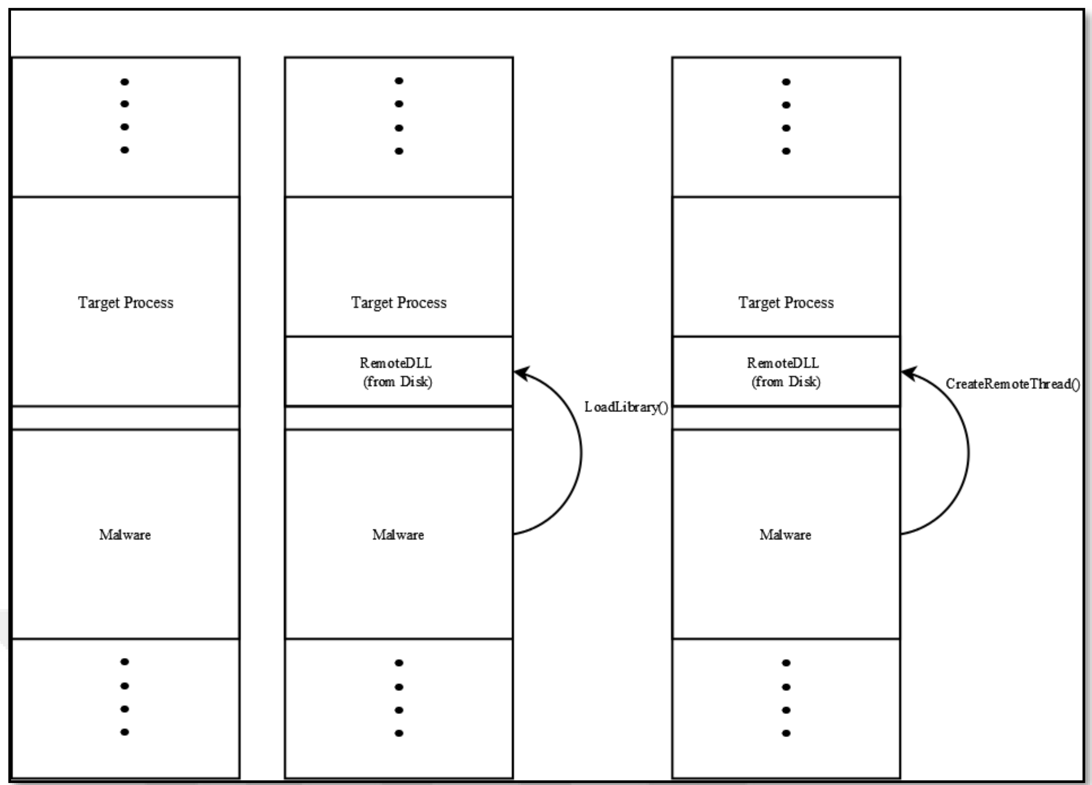


Figure 5.3. Remote DLL Injection using LoadLibrary method.

Reflective DLL Injection is the hybrid process of Code Injection and DLL Injection. This method involves loading a DLL to the target address space from memory and in which the loading process is done by native DLL coding instead of using LoadLibrary() method. This property of a DLL loading itself is making this procedure stealthier as it leaves no mark on the Disk. The loaded DLL can also be downloaded from a URL of the malicious actors, which makes this method a file-less malware.

5.2.4. Process Hollowing

This method is one of the stealthiest methods of hiding a process. This method has been used in highly effective APTs such as Stuxnet, Duqu, and Patchwork (Bencsáth et al., 2012; Cymmetria Reseach, 2016; Falliere et al., 2011).

The process of hollowing a process is illustrated in Figure 4, and it is as follows:

1. The malware starts a new instance of a legit system process such as lsass.exe. This process starts in the suspended state by providing the parameter of the creation flag to CREATE_SUSPENDED.
2. The malicious code is fetched from memory, disk, or over the network.
3. The code section of the target process is unmapped, and the process becomes a hollow process. The commands used here can be ZwUnmapViewOfSection or NtUnmapViewOfSection.
4. A new memory segment with PAGE_EXECUTE_READWRITE permission is allocated from the memory space of this hollow process using the virtual memory allocation calls.
5. PE Header of the malicious process is copied into the hollow process.
6. Each segment of the malicious code is transferred to the proper virtual address space of the hollow process.
7. The start address is set so that the malicious code starts from its entry point.
8. Suspended thread is resumed.

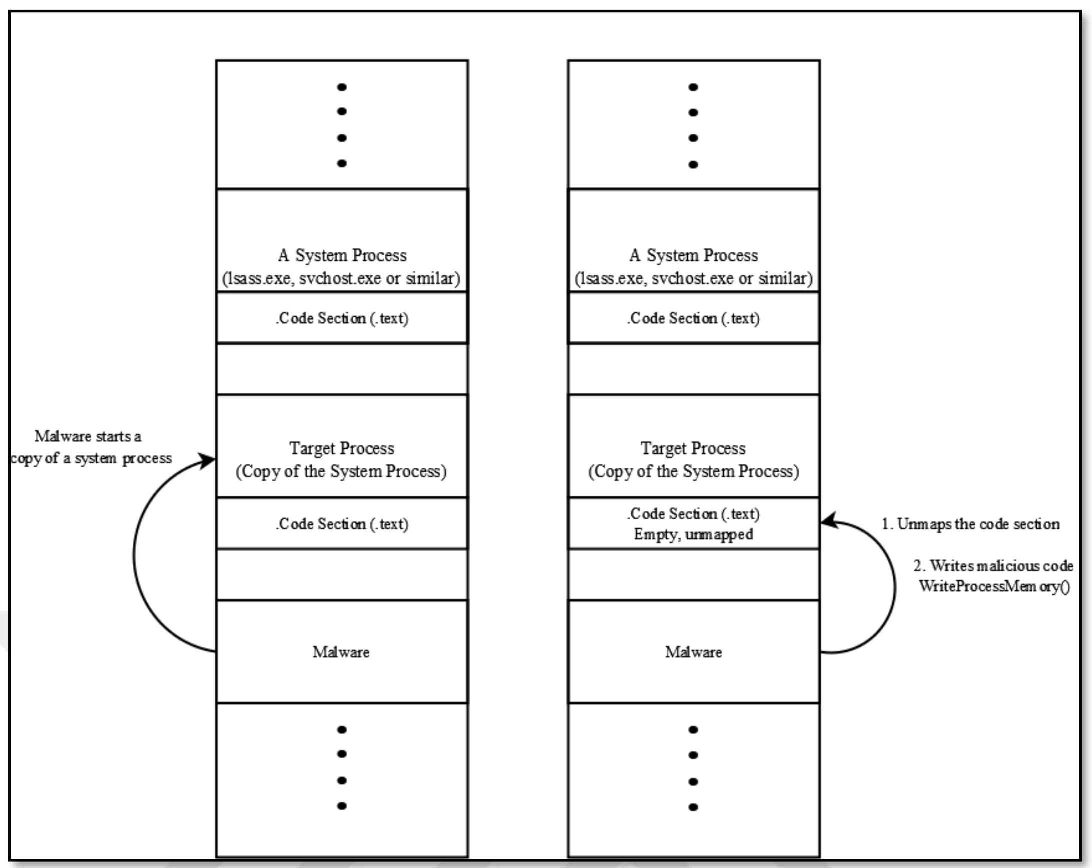


Figure 5.4. Illustration of Process Hollowing.

5.3. Manual Analysis Detection of Memory Operations

A rule of thumb approach for understanding memory operations is to check the Process Environment Block (PEB) structure and cross-reference with the Virtual Address Description (VAD) structure of the Kernel Space of the memory.

PEB structure is a data structure that exists for every process, and it contains the full path of the executable, the full command line that starts the process, pointers to heaps, standard I/O handles and data structure for holding the loaded DLLs and Modules. PEB structure is accessible within the process itself, and malware authors most commonly play with these structures to hide their intentions. However, the VAD structure is in the kernel space of the memory, and under normal circumstances, the information in the VAD and different PEBs should be aligned. Therefore, cross-referencing these two structures would identify most of the malicious code and DLL injection attacks. For example, in DLL Injection, if the injected DLL and its loading place from the disk are not consistent in these two structures, or the DLL has no record in the PEB but exists in the VAD, it means the DLL forced into the process address

space. However, in Process Hollowing and its variants, as the process is a legit process, and only the contents of the sections are modified, it is harder to detect with cross-referencing.

5.4. The literature on Automated Malware Detection with Memory Analysis

While the vast amount of the literature uses API call sequences and call traces for the behavioral graph, there is a relatively short list that uses memory access for the detection for the malware. By utilizing the memory forensics techniques, operating systems resources and their handles, registry keys (Zhu, Gladyshev, & James, 2009), running processes and threads, network connections, loaded DLL files and even commands that had been previously given can be retrieved (Stevens & Casey, 2010). In the study by (Kolbitsch et al., 2009), the memory access of malware families had also been integrated into the behavior graph of the sample. Moreover, in the work of Duan et al., the tool Detective, extracts DLL imports from the memory snapshots and applies HNB classifier to this data (Duan et al., 2015).

In the work of Korczynski and Yin, a tool named Tartarus, a solution to the aforementioned injection techniques and code propagation approaches is given by generating and combining taint analysis with the tracing of the memory writes through execution tracing (Korczynski & Yin, 2017).

Another line of work incorporates memory differences; snapshots of the memory image are taken several times, and the differences are identified. One of the work in this line is (Zaki & Humphrey, 2014); This research has identified the modifications (file system changes, newly loaded driver or a newly loaded image) that are done in kernel space by a rootkit and generates signatures using this information.

A similar memory differencing approach attacking this problem is given by (Teller & Hayon, 2014). A Cuckoo Sandbox plugin is described in their paper, and the applied idea was to take snapshots of the memory whenever something important (defined by API call triggers) comes up such as loading some image into memory or generating a call to a registry call. With a similar approach, the studies (Mosli, Li, Yuan, & Pan, 2016, 2017a) and (Aghaeikheirabady, Farshchi, & Shirazi, 2015) extract DLL, API calls and registry activities using the memory analysis program volatility to generate features to be fed into a machine learning tool.

Observations

Our approach differentiates from the above by binary instrumentation of the memory access instructions and logging every memory access in different regions of the process memory separately. A similar approach is taken in the work of (Banin, Shalaginov, & Franke, 2016). In this study, their memory accesses are reduced to read and write operations regardless of the region and section information of the memory. The order n-grams are then fed into a learning network to identify maliciousness. However, our work differentiates from this one by taking into spatial properties of the memory access in addition to just using the order of the memory access types. In the referenced work, only the order of operation, and the type R/W is taken into consideration.

Taking the snapshots of memory regions within an interval and comparing them through the kernel objects identifies a lot about the malware however in most cases the malware creates several other processes, injects itself to another process or service as explained in the previous section of this Chapter. This contagious behavior of the malware results in a vast region of memory space to be snapshot within the interval, and therefore the method becomes infeasible (Bletsch, Jiang, Freeh, & Liang, 2011).

Another problem of taking a snapshot of the memory is that deciding the frequency of snapshots. If there are too many snapshots, there will be gigabytes of data to be analyzed for just malware, and if the interval is set long enough, there might be a chance that the malicious activity can slip from the memory before getting in one of those memory images snapshots.

5.5. Conclusion

This chapter identifies the memory operations of malware, the tricks, and workarounds that are being used by contemporary malware. The literature of the filed in the intersection of malware detection and memory forensics also provided in this Chapter. In the following Chapter 6, the dissertations main work, the general idea, and the graph database for created for memory access is described and detailed.

CHAPTER 6
MALWARE MEMORY IMAGING AND EVALUATION

This Chapter presents and defines the main aims and contributions of this dissertation, along with the problems and contributions to solutions to these problems. A malware memory image is a 3-dimensional representation of the memory access patterns of malware. The methodology presented in this chapter is a method of identifying these access patterns through binary instrumentation.

As presented in Chapter 5, different malware has different methods of doing malicious work on memory, which results in distinctive behavioral characteristics for malware families as malware adopt code from its predecessors. In this dissertation, it is aimed to show and compare these characteristics by the memory images between several types of different malware families and types.

6.1. Motivation

The vast amount of work is relying on the dynamic API calls and DLLs as discussed in Chapter 4 – Dynamic Analysis of Malware, and it is so far one of the most promising methods in the literature. However, as malware authors become aware of this detection technique, they start to implement a workaround with Native DLL coding. Native DLL is the dynamic library written directly in the machine language, and malware authors prefer to include binary formatted DLLs instead of calling them from Windows libraries. Native DLL coding provides a level of stealthiness against behavioral detection mechanisms.

Another reason the detection mechanisms fail is the general assumption that a malware process should be executed/started from a hidden file somewhere on the disk and it needs to spread to other files. On the contrary, new types of malware are designed to work, spread, and complete their lifecycles only on memory. They are downloaded from network to memory via exploiting vulnerabilities in operating system tools and programs. For this reason, signature-based detection, static analysis, and sandboxing effectiveness are significantly decreased for such malware.

As discussed in the previous chapter, incorporating memory forensics into dynamic analysis and detection provides promising results on these types of malware (Mosli et al., 2016; Mosli, Li, Yuan, & Pan, 2017b; Rughani & Rughani, 2017; Teller & Hayon, 2014). However, the approaches for taking memory dumps and snapshots of process address space suffers from two problems. Firstly, it is vital to decide how often the memory dumps are going to be taken. When it is too frequent, there will be tens of gigabytes data for just one malware, and when the interval between dumps are long, it is possible that the malware execution can slip away between dumps without detection. The second problem is when taking memory dumps of a process; it is possible that the malware injects itself to another process and continue its execution from another address space. In this case, complete memory dumps have to be taken, but unfortunately, this approach also leads to the analysis of gigabytes of data for malware again.

In this dissertation, the developed solution to this problem is to instrument memory access operations. Instead of taking memory dumps, every operation of a process is traced, and the memory usage patterns are observed. This way, the memory operations are captured as the process is still running on the memory, and taking memory dumps becomes unnecessary. Our method is also promising for file-less malware as the instrumentation is done on a memory level.

The memory access operations are converted to 3-dimensional patterns to capture significant characteristics of malware. These 3-dimensional patterns are constructed from a type of access (Read/Write), access sequence, instruction address, and access address. Since the patterns are built from the physical operations to memory, natively coded DLLs are recorded as well as operating systems API calls, functions, DLL calls, and several other procedures that affect memory. It is shown in this research that similar malware samples coming from the same families show similar memory access patterns in 3- dimensional space constructed by the sequence of the access relative to the computation instruction sequence, the instructions address and the memory address that is being accessed. Also, in the defined 3-dimensional space, process injection, packing, and malicious acts affecting other processes address space can be identified by our methodology.

6.2. Instrumenting the Memory Operations

The dynamic binary instrumentation system presented in this research is provided by the Intel PIN Tool version 3.7 (Luk et al., 2005). The executable samples to be imaged are executed by the Intel PIN Tool and our memory tracer DLL is inserted into the executions address space to instrument every single memory operation. The details on Intel PIN Tool is given in Chapter 4.

6.2.1. Memory Layout

The memory layout for all the Win32 applications, which is called Portable Executable (PE) format consists of several images that are dynamically loaded into memory during the execution.

Majority of the malicious files intend to work Win32 based systems and to develop our methodology; selected samples are executables in the PE format. The Structure of Portable Executable format is given in Chapter 2. The standard layout of the executables on the memory is given in Figure 6.1. Two main sections: the header and the sections are present in the PE format (Pietrek, 2011). The execution of an executable starts from the loader of the operating system. The loader first reads the header page of the PE file and retrieves the image base. The process memory space is allocated and divided by sections according to the image base. The import tables are read for loading all the DLLs which are set to be loaded at the initialization of the program. After loading the DLLs, the real addresses of the functions is resolved and stored in the import address table. Afterward, the main thread is created; the instruction pointer is set to the entry point of the main thread and execution starts.

The images of the DLLs and Libraries are also in the format of a PE layout. During an execution when a function is referenced inside an image, the instruction pointer jumps to that specific location of the instruction and the memory accesses occur from these addresses.

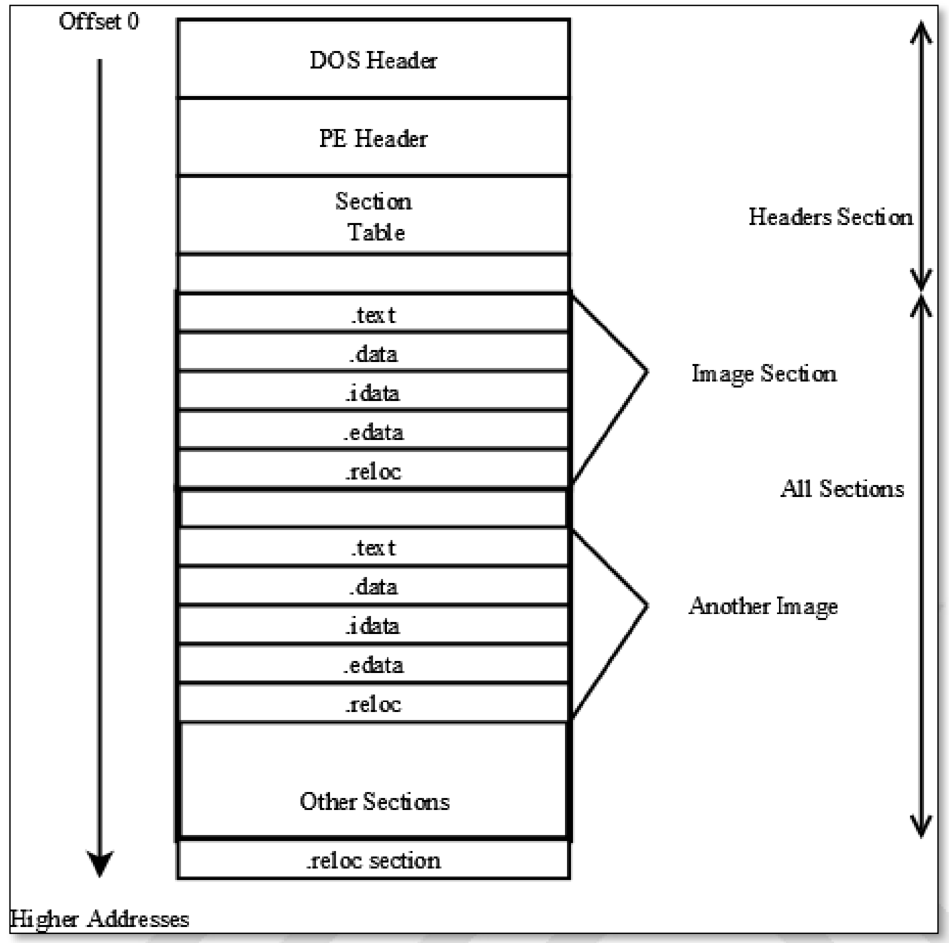


Figure 6.1. Memory Layout of a PE Format.

Even though the memory layout is a one-dimensional address space, there are more dimensions to consider when it is a memory operation. Three compounds are defined in our methodology for a memory operation: the sequence of the operation, the instruction addresses, and the address that is being accessed. The sequence defines the order of the access, which creates an ordered set from these operations. Another valuable property is the mode of the accesses of which can be a read or a write operation.

6.2.2. Flowchart of the Methodology

Figure 6.2 below is the flowchart of our methodology. The process starts with creating a Windows 7 SP2 virtual machine with Intel PIN Tool installed on it. The virtual machine is hardened for the virtual machine detection modules of the malware with the analysis program Paranoid Fish – Pafish (Ortega, 2016). The virtual machine

(VM)'s snapshot is taken with the clean state before any infection, and every new sample is analyzed on this clean state of the VM.

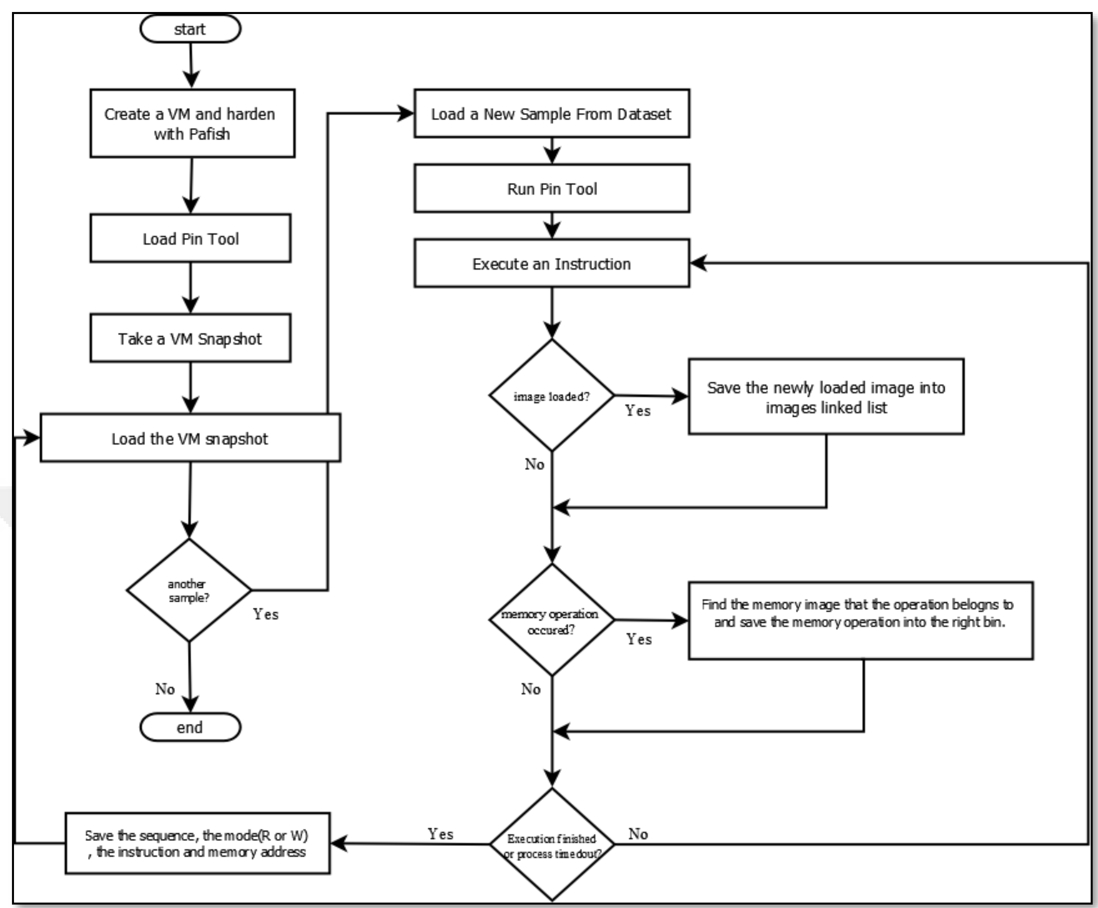


Figure 6.2. Flowchart of the Proposed Methodology.

After the new sample is loaded into the VM, the process is started with the PIN Tool with our memory tracer DLL utilized. During the execution, when an image is loaded, an entry table for images in our module keeps their traces to bin the memory accesses according to their module address spaces. There is a time limit of three minutes for the execution to finish, or the process is ended automatically.

The automation approach taken in this methodology is similar to the one in the work of (Banin et al., 2016) although it differentiates from it by the custom pin tool that is developed specifically for this research. Distinguishing from the work of Banin et al., the developed pin tool captures the memory layout of the process by continually monitoring the image loading and unloading operations. While doing so, the memory access operations are also logged and classified by the memory images inside the process memory. The accesses to stack, heap, and memory images are logged

separately for each image in the process memory layout.

As explained in the previous Chapter 5, most of the malware memory operations are done from the code section. The binary should execute a series of memory writes to unpack itself or to copy and inject its malicious components to other memory spaces. Therefore in our methodology, the memory accesses of the code section are traced and imaged to generate these characteristics.

6.2.3. Algorithm for the PIN Tool

In the algorithm below, the work of our tracer is described. The instrumentation is done one instruction level; every instruction is checked for stack access and memory access. Stack operations can lead to compiler characteristics results as the optimization levels, and stack operations are most likely to be arranged by the compilers. Memory reads and writes are tracked for the cases as described in the algorithm.

Algorithm: Pin Memory Tracer

Input: A program for tracing, An image linked list - *imgs*,

Output: A trace file - *trace.out,* sequence, containing instruction pointer, accessed memory location, mode of the operation, the base of the image that has done the access.

0: *seq* := 0
1: **for** each instruction *ins*:
2: **if** *new_image_loaded* is **true**:
3: Get the base address and highest address of the image and insert it in *imgs*.
4: **if** *ins* has stack read:
5: Write it in output buffer as stack_read.
6: **else if** *ins* has stack write:
7: Write it in output buffer as stack_write.
8: **else**:
9: **if** *ins* is a memory read:
10: Look up *ins* address in the imgs, write it in outbut buffer as <seq, instruction address, accessed memory location, ‘R’>
11: **else if** *ins* has memory read: //such as in adding a value from memory // to a register
12: Look up *ins* address in the imgs, write it in outbut buffer as <seq, instruction address, accessed memory location, ‘R’>
13: **if** *ins* is a memory write:
14: Look up *ins* address in the imgs, write it in outbut buffer as <seq, instruction address, accessed memory location, ‘W’>
15: **Increment** seq by 1

INS_InsertPredicatedCall API Function

An instruction is traced in the PIN tool by the JIT compiler, the dynamic compilation process, by the function **INS_InsertPredicatedCall**. This call inserts our recording function if the next instruction has stack or memory operation in its byte code. Therefore the runtime is affected by the number of memory operations and added overhead for the dynamic compilation. The detailed runtime analysis for PIN tool can be found in (Luk et al., 2005).

6.2.4. A Memory Image Extraction Example

In the example below, a malware from the Keygen Trojan family with the md5: 5fe2aebb2fe4abe503d297c318a37a62 is exampled. By observing the traces of the memory operations and image loadings of this malware, a linked list with images is constructed. The table of the linked list entries is given in Table 6.1 for this particular malware. The table is snipped as there are 42 images in the address space of this malware.

Table 6.1. Linked list of memory images of the malware Keygen with md5: 5fe2aebb2fe4abe503d297c318a37a62.

Memory Image	Image Name	Size in bytes
1	ADVAPI32.dll	1241704
2	CFGMGR32.dll	246345
3	COMCTL32.dll	624878
4	CRYPTBASE.dll	157748
5	DEVOBJ.dll	242715
6	devrtl.dll	243177
7	dwmapi.dll	167310
8	GDI32.dll	1192927
9	heap.csv	67104884
10	imm32.dll	524073
11	kernel32.dll	6307693
12	KERNELBASE.dll	2341679
13	LPK.dll	148552
14	Keygen.exe	24399092
15	mfc42.dll	1947667
16	MSCTF.dll	567270

After the extraction of the memory images, the memory access from malware’s code section is converted into 3d data patterns where the dimensions are the sequence,

instruction pointer, and access pointers. The data is visualized through plotting the 3d data, and the points in the patterns are given colors regarding whether it is a read or write operation. Following Figure 6.3. is the extracted image of the example malware.

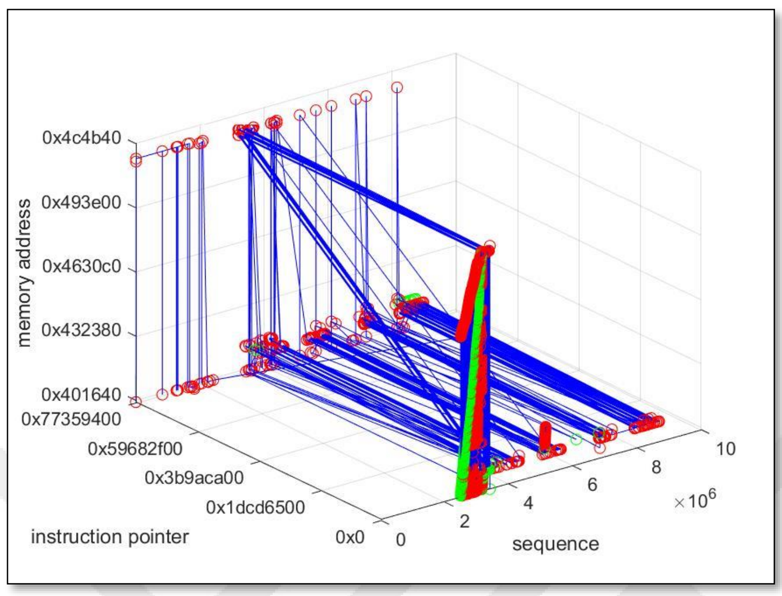


Figure 6.3. The Extracted image of one of the images from the Keygen Trojan family. The md5 of the malware is 5fe2aebb2fe4abe503d297c318a37a62.

6.3. Graph Database Model for Memory Access and a Showcase

In our methodology, the extracted data is saved on a graph database provided by Neo4j. Every memory access is saved as a node in this database for further analysis. The reason for utilizing such data tool is to increase the efficiency of pattern recognition algorithms for future studies. An example analysis of a sample from the Ramnit Worm family is provided in this section for further reference (see Table 6.2).

Table 6.2. Sample Details for Example Analysis.

Malware Sample	Worm.Ramnit.0fe268b9d7eade3a9270e5ab4e54e77d
Memory Access Count	8579
Sha1 Hash Value	a0abbf36a32d22a2e178bb8e2fb82ba6d17c651d
Type	PE32 executable (GUI) Intel, 80386, for MS Windows, Nullsoft Installer self-extracting archive
Section Count	5

Memory access nodes are not connected in the beginning, since there are in total of 134.535.767 memory access entries in our database. However, the selected sample can be connected with a simple relationship called DISTANCE providing the Euclidean distances between nodes. The following Figure 6.4. shows a six hundred of the memory accesses connected with DISTANCE relationship (the result is reduced due to the visibility of the nodes).

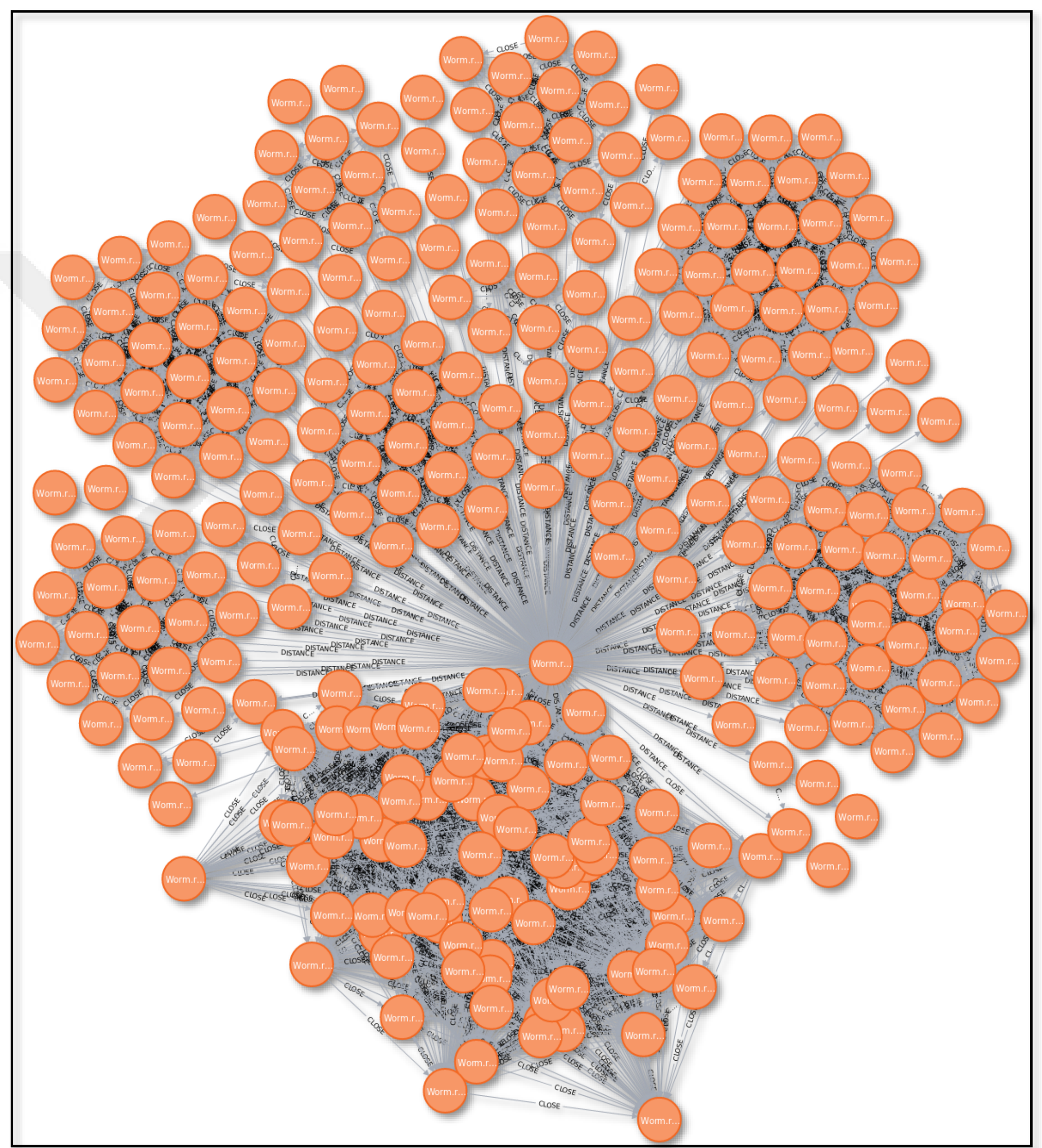


Figure 6.4. Memory Accesses of a Sample connected with Euclidean Distances.

A meaningful query would be looking for consecutive writes on the memory as it is an indication of code injection. Therefore, our demo analysis result is queried for

consecutive writes, which have Euclidean distances between nodes smaller than 2. The shortened query results in Figure 6.5.

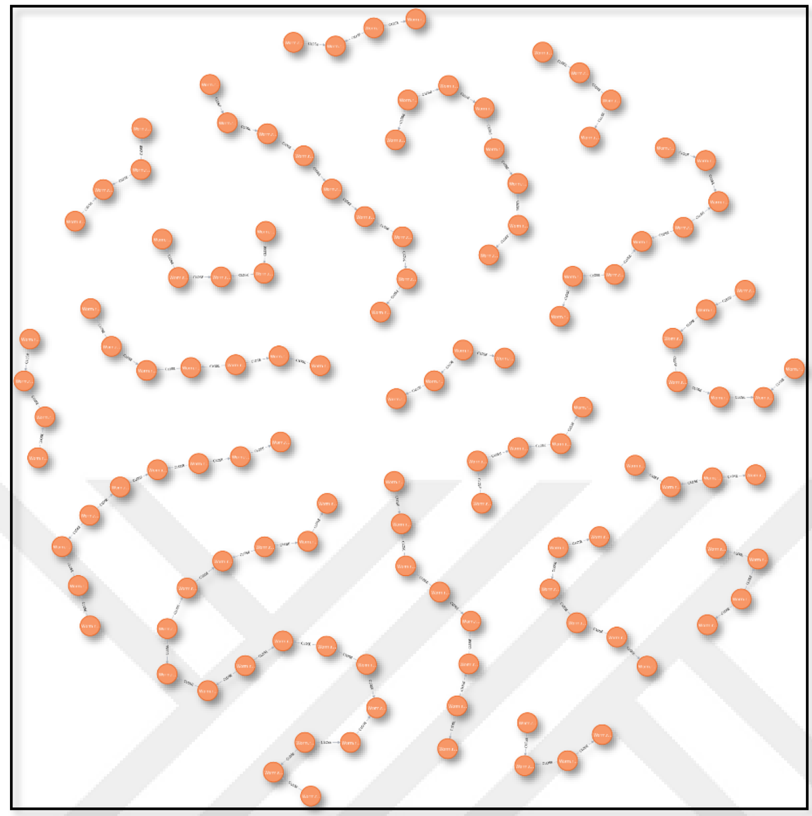


Figure 6.5. Consecutive writes of the malicious sample.

After getting the instruction addresses of the consecutive writes, an analyst can quickly get the image base of the code section and extract the memory exploiting parts. This methodology would significantly reduce the time of the analysis. Another analysis demonstration would be that understanding the memory access clustering of the malware. As memory accesses should be done on memory sections either created by LoadLibrary() or one of the Virtual Allocation functions, there needs to be a clustering of sections. A straightforward analysis would be to merge the nodes with Euclidean distances are smaller than some threshold value. This value should be selected based on the observations on the data. In the following Figure 6.6, the nodes with Euclidean distances are smaller than 100 are shown.

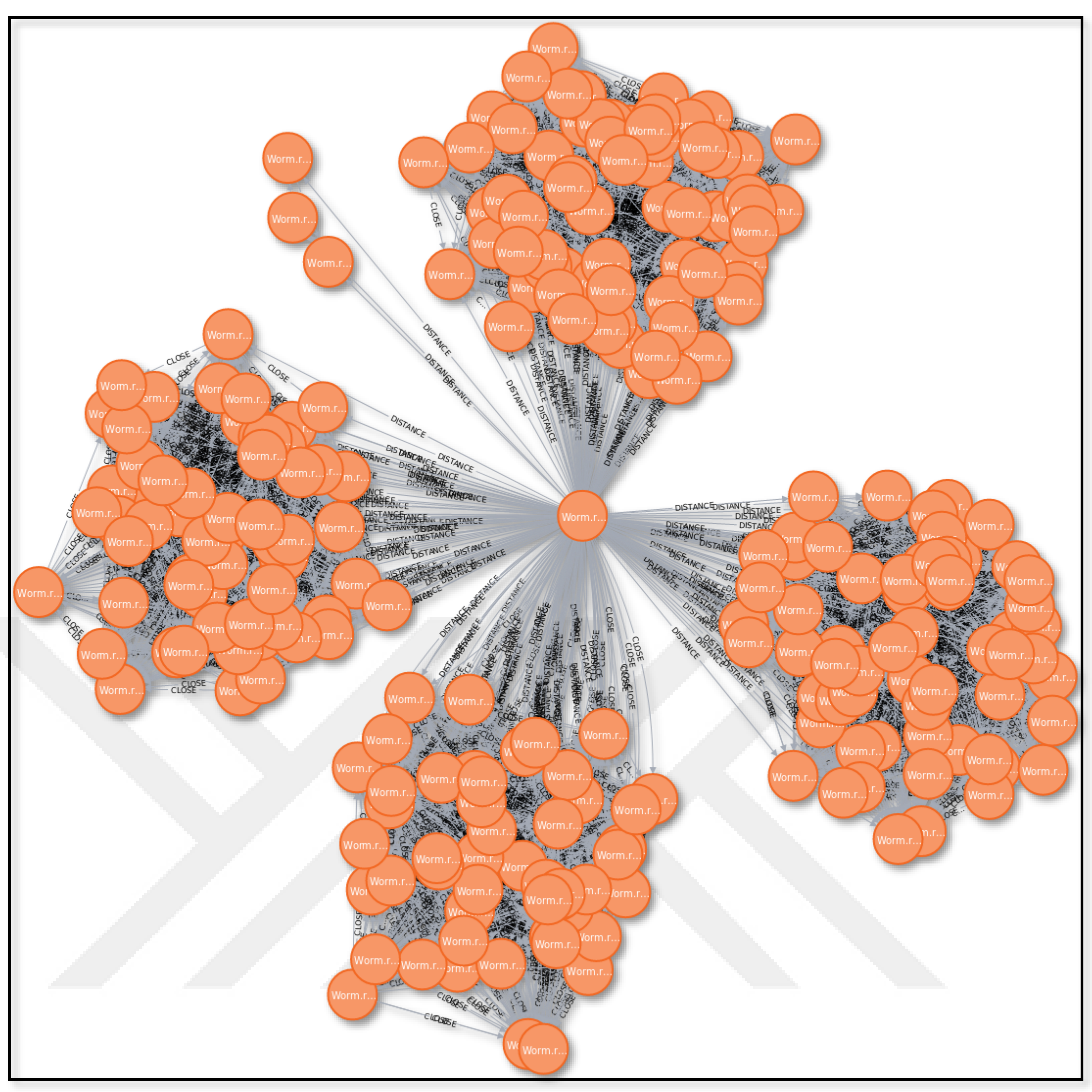


Figure 6.6. The nodes with Euclidean distances are smaller than 100 are merged in this example. An apparent clustering of the memory access can be seen from this figure.

6.4. Structured Similarity Index Measure (SSIM)

The comparison of the memory images is given by the Structural Similarity Index given by the work of Wang et al. (Wang, Bovik, Sheikh, & Simoncelli, 2004). This method is used to measure the similarity between malware families in this research. The method takes two images and calculates their average, variance and cross-correlations of the binary strings of the two images and return the luminance, contrast and structural similarities. These similarities then merged into one similarity metric. The derivation of the metric is as follows:

$$SSIM(x,y) = [l(x,y)]^{\alpha}.[c(x,y)]^{\beta}.[s(x,y)]^{\gamma}$$

where

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}$$

$$\mathrm{c(x,y)} = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}$$

$$s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$$

where $\mu_x, \mu_y, \sigma_x, \sigma_y$ and σ_{xy} are the local means, standard deviations, and cross-covariance for images. The exponents α, β and γ are the weight of the similarities of luminance, contrast and structural similarities. These exponents are selected as 1 which is the default settings of the implementation of the SSIM in MATLAB 2018b which is given by the authors of (Wang et al., 2004).

6.5. Conclusion of Chapter 6

In our methodology, we highlight two contributions to this chapter. The first one is the memory access analysis can be a characteristics feature of a binary. By analyzing the access patterns in 3d dimensions, as explained in this chapter, various patterns for memory operations can be extracted from this data. By instrumenting the memory access instructions in the binary in an isolated environment, both the benefits of the dynamic analysis and memory forensics can be merged as shown in this Chapter.

The second contribution is the demonstration of spatial properties of the 3d data can be used to identify even more signatures and characteristics. However, the investigation of these properties are left as future work, and only a showcase is presented.

In the next chapter, we present our tests on a malware dataset, which led to 134.535.767 memory access. SSIM has been used as a measure of similarity and relevance between similar memory operations and malware families are presented.

CHAPTER 7
TESTS AND RESULTS

This chapter presents the test and results on extracted malware memory images with the methodology provided by the previous chapter. The memory images are tested against their behavioral characteristics, and these characteristics are matched with the memory artifacts and results. Similarities on the memory images are revealed, and the memory images, particularly on the merits of their similarities amongst their malware families are evaluated and discussed.

7.1. Software Support

Several open source projects are utilized in this research. Intel's Pin Tool version 3.7 is used for the dynamic binary instrumentation, as mentioned in the methodology chapter. The malware memory images are inserted in a graph database, and the graph database engine is provided by Neo4j version 3.5. For the implementation of the sandboxing and automation, Oracle's Virtualbox Software version 5.2 and its API are integrated into the research. To compare and contrast our malware for their behavioral analysis, Cuckoo Sandbox version 2.0.6 is combined. MATLAB R2018b and SSIM implementation in this software is used. In the rest of this chapter, similarity ratio refers to the value of the result of SSIM.

7.2. Malware Dataset

A dataset consisting of malicious and benign programs are collected to test our methodology. The malicious samples come from 24 families of different types of malware composed of trojans, ransomware, viruses, and worms in a total of 121 malware samples. In addition to malware families, six benign portable executable files are included in the dataset. The malware families and the number of samples from each can be found in Table 7.1. The malware samples are downloaded from Virusign

website (ViruSign, 2019), and the benign files are downloaded from http://www.portablefreeware.com website (Freeware, 2019).

For demonstrating and testing our methodology, six system internal programs are included in the dataset. These programs although being extremely useful for system analysis, process tracking and memory tracking, their methods show similarities with that of the malicious samples. These programs are downloaded from from http://www.sysinternals.com website (Microsoft, 2019).

Table 7.1. The distribution of the malware samples in our dataset.

Type	Family	Number of Samples
Trojan	Autorun	5
Botnet	BackOrifice	5
Ransomware	Ceber	4
Ransomware	Cryptowall	5
Botnet	Cutwail	5
Ransomware	Jigsaw	4
Trojan	Keygen	9
Botnet	Lethic	2
Ransomware	Locky	5
Botnet	Marina	3
Trojan	Matsnu	1
Botnet	Necurs	6
Trojan	Netbus	3
Cleanware	Portable	10
Worm	Ramnit	10
Virus	Rex	2
Botnet	Sality	6
Botnet	Storm	4
Trojan	Sub7	3
Cleanware	Sysinternals	6
Trojan	Tdss	3
Ransomware	TeslaCrypt	5
Botnet	Torpig	4
Virus	Virut	5
Virus	Xpaj	4
	Total	121

7.3. Malware Memory Patterns

In this section, our observed characteristics for similar patterns are presented. Observed patterns are grouped by the memory operations of malware, as discussed in Chapter 5. The common groups of patterns are;

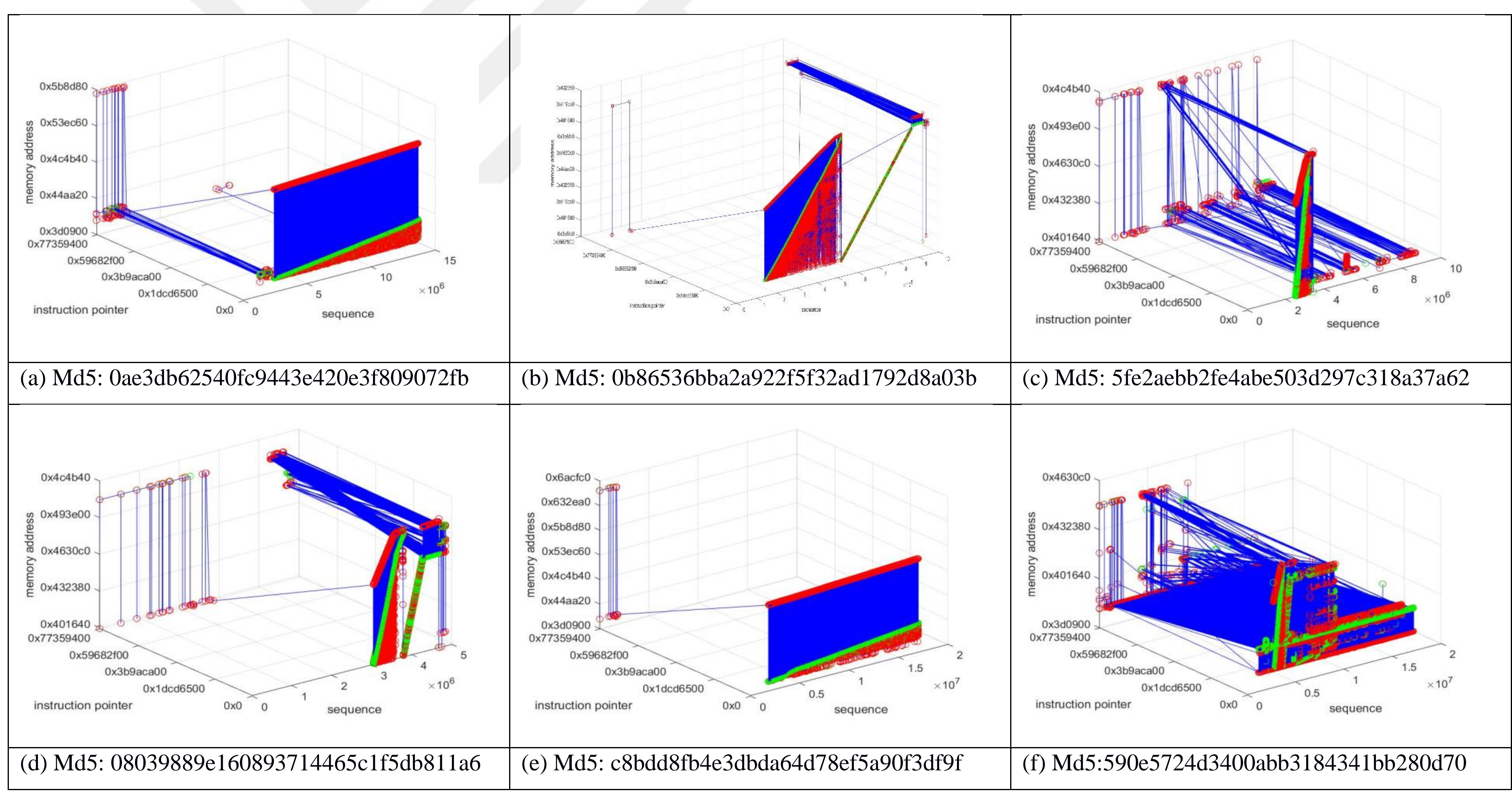
- The Ultimate Packer for Executables (UPX) packing algorithm patterns,
- Packing patterns,
- Process or Code Injection Patterns
- Encryption patterns, particularly ransomware patterns
- Infectors and Virus patterns.

These patterns are evaluated with the Average Similarity Measure built on SSIM method. This is a straightforward approach to apply the SSIM index to a group of pictures where the average value is calculated after the calculation of the pairwise similarities within a group of samples. Although there are ~134 billion memory operations that are investigated for this research, the number of malware samples is moderately low for a kernel-based similarity and clustering approach (Choi, Cha, & Tappert, 2010). Therefore applying an $O(n^2)$ algorithm to calculate all the similarities between the pairs exhaustively suffices for this research.

For the memory images in this section of the dissertation, the red points are for the read operations, and green points are the write operations. The sequence of the access, instruction address, and accessed memory locations are given in x, y, and z-axis, respectively. The lines in between memory accesses construct the blue colored spaces.

UPX Packing Patterns

Our first observation was packed binaries, particularly the ones that are packed with UPX produce similar patterns on the 3d memory images while unpacking themselves on the memory. Figure 7.1. presents two benign software and two malware samples from Keygen family. It has been detected that these four samples show similar patterns of consecutive writing to the memory space incrementally. The similarity ratio for UPX patterns is 0.6724712.



(a) Md5: 0ae3db62540fc9443e420e3f809072fb
(b) Md5: 0b86536bba2a922f5f32ad1792d8a03b
(c) Md5: 5fe2aebb2fe4abe503d297c318a37a62
(d) Md5: 08039889e160893714465c1f5db811a6
(e) Md5: c8bdd8fb4e3dbda64d78ef5a90f3df9f
(f) Md5:590e5724d3400abb3184341bb280d70

Figure 7.1 UPX patterns for the samples respectively, (a), (b) portable benign executables, (c), (d), (e) Trojans from Keygen Family, (f) a trojan from Sub7 family. The average similarity ratio for these patterns is 0.6724712.

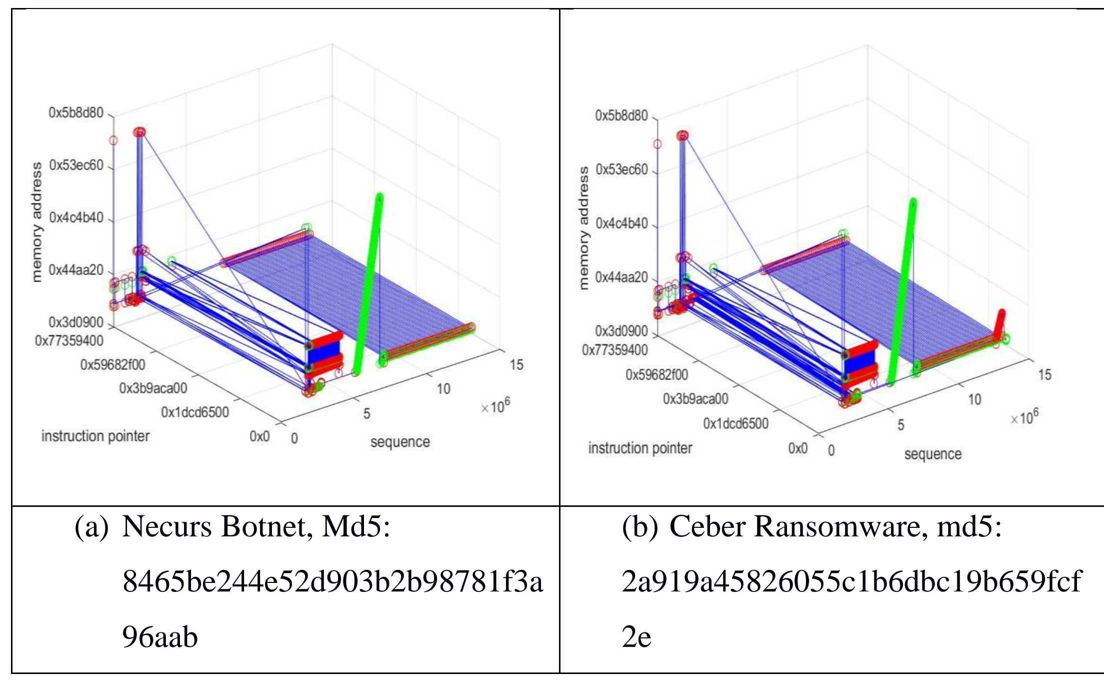
As can be seen from Figure 7.1., there is a clear packing pattern in all the images, and the average similarity ratio is around 0.67. The sizes of the packed executables are varying, and the unpacking operation has resulted in more extended patterns for some executables, as in Figure 1 (e) and Figure 1 (b). The trojan from Sub7 malware family, Figure 1 (f), is relatively different from the other images in this Figure, as it contains injection patterns in its memory image although the unpacking operation can be seen around the sequence 0.5 x 10^{7}.

Packing and Self-Decrypting Patterns

For packing algorithms other than UPX, the patterns from Figure 7.2. are observed. These patterns from different malware families show similar memory operations within their address space.

Another observation in these patterns is the self-decrypting malware result in similar memory patterns with that of the ransomware. In Figure 2, the similarity ratio for Figure 2 (a) and (b) is 0.987532, which shows that their packed with the same algorithm and showing similar memory operations.

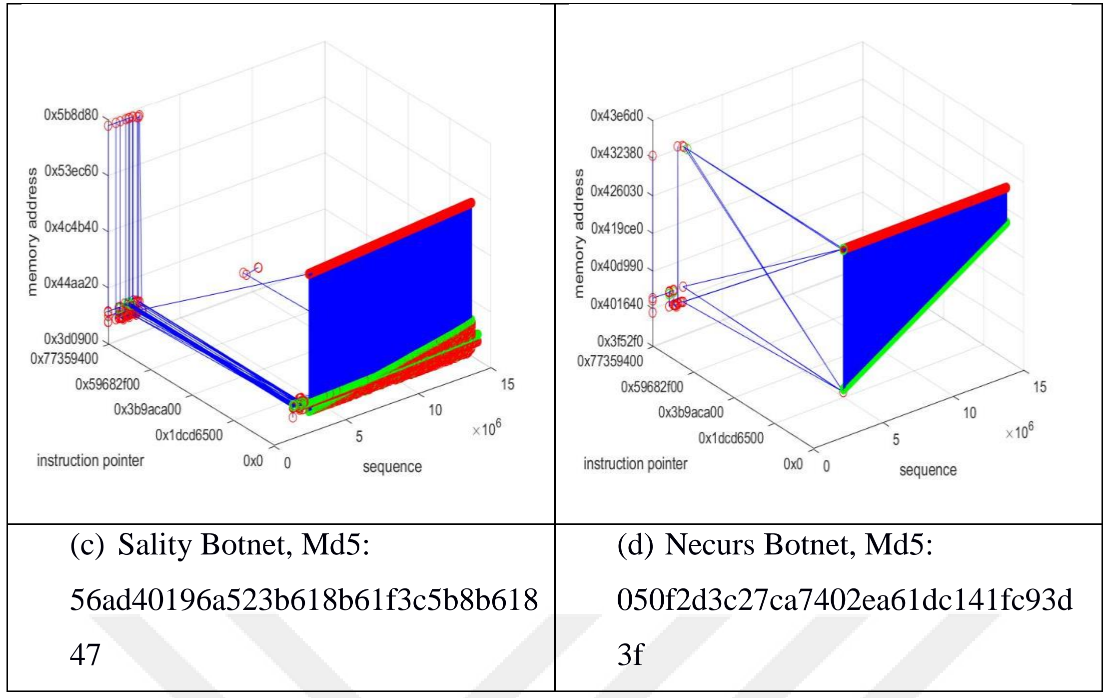
Figure 2 (c) and (d) are revealing another pattern for packing and self-decrypting structure even though two samples are coming from different botnet families.



0x5b8d80
0x53ec60
0x4c4b40
0x44aa20
0x3d0900
memory address
0x77359400
0x59682f00
0x3b9aca00
0x1dcd6500
0x0
instruction pointer
0
5
10
15
×10^6
sequence

(a) Necurs Botnet, Md5: 8465be244e52d903b2b98781f3a96aab

(b) Ceber Ransomware, md5: 2a919a45826055c1b6dbc19b659fcf2e



(c) Sality Botnet, Md5: 56ad40196a523b618b61f3c5b8b61847	(d) Necurs Botnet, Md5: 050f2d3c27ca7402ea61dc141fc93d3f

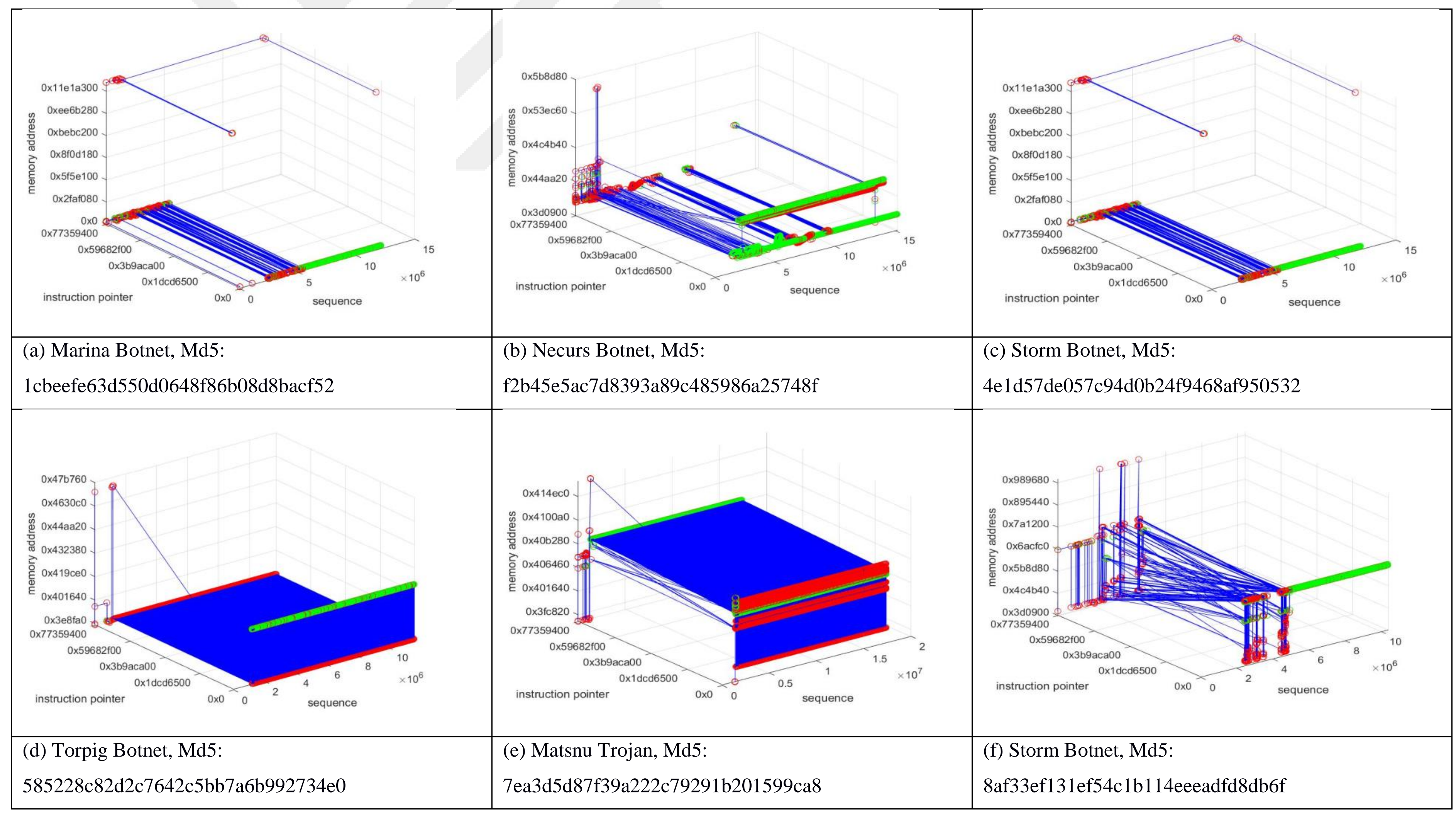
Figure 7.2. Observed Packing and Self-decrypting patterns.

Process or Code Injection Patterns

Code injection is the process of copying malicious executable code payloads or malicious pe executables into another process address space and running the code from there. In the following Figure 7.3., indications of rate code injection are present.

The observed patterns are from several malware families. These families are Marina, Torpig, Storm, Necurs, and Matsnu. The resulting patterns show the visibility of writing to the same memory space, possibly an opened pipe between process address spaces, and therefore forming straight green lines along the sequence axis.

Two botnets in Figure 7.3. (a) and (c), showing very close observations. This is generally the reason for using the same algorithm or code piece for injection into another process's address space. Their similarity ratio is 0.989123. Figure 7.3. (d) and (e), Trojan Matsnu and Botnet Torpig show injection to another module address space after reading either unpacked or decrypted the malicious code. These patterns they construct are unique in other samples, and these patterns occurred only among in their family members in our dataset. This observation can be seen later on this chapter in the results of malware family pairwise comparisons. In Figure 7.3. (b) consecutive reading and writing is done by the Necurs malware, which indicates a calculation on the written data done prior to injecting. This pattern also can be seen in ransomware samples.

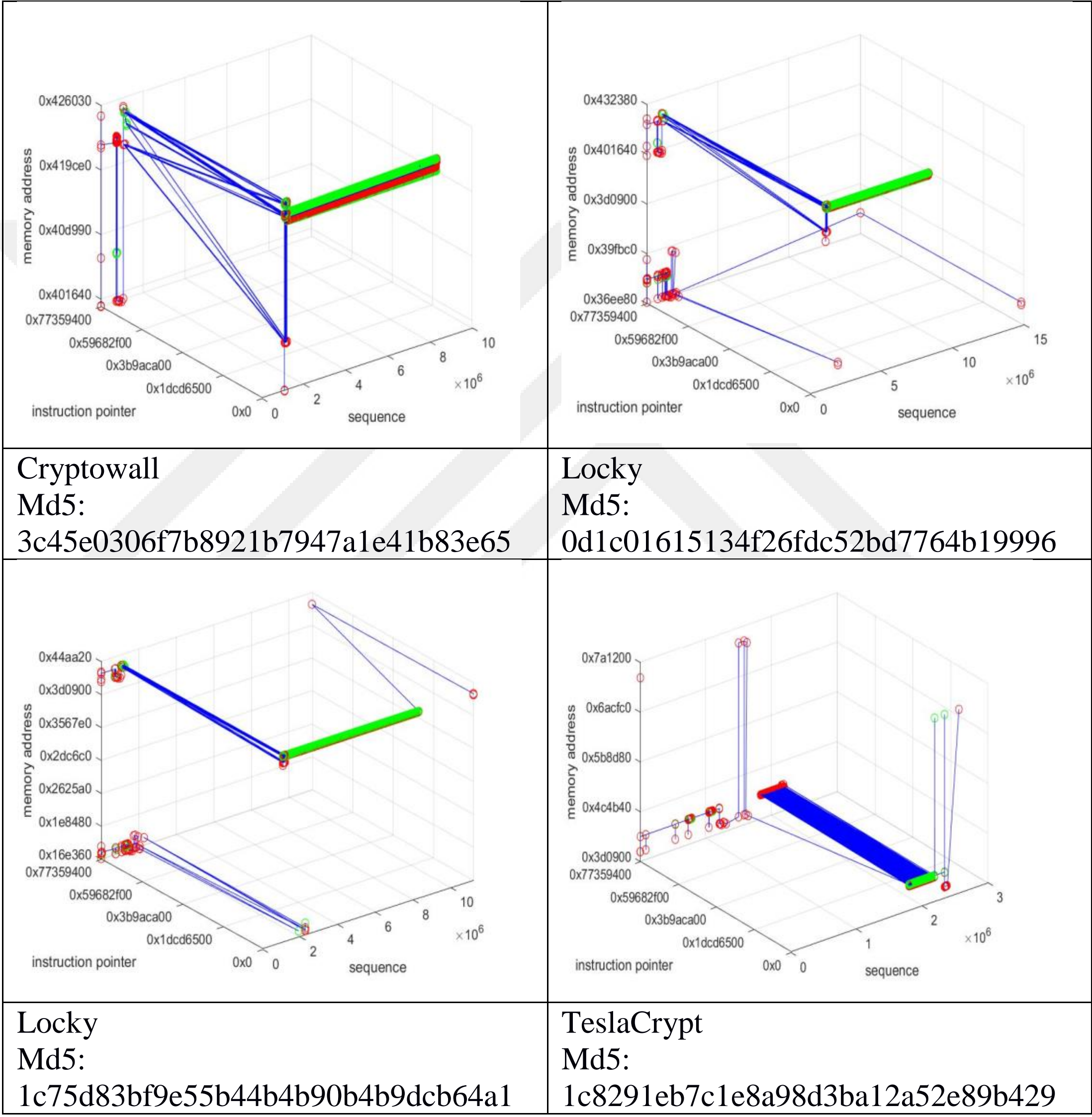


(a) Marina Botnet, Md5: 1cbeefe63d550d0648f86b08d8bacf52	(b) Necurs Botnet, Md5: f2b45e5ac7d8393a89c485986a25748f	(c) Storm Botnet, Md5: 4e1d57de057c94d0b24f9468af950532
(d) Torpig Botnet, Md5: 585228c82d2c7642c5bb7a6b992734e0	(e) Matsnu Trojan, Md5: 7ea3d5d87f39a222c79291b201599ca8	(f) Storm Botnet, Md5: 8af33ef131ef54c1b114eeeadfd8db6f

Figure 7.3. Process Injection Indications on Various Patterns.

Ransomware Patterns

Another important observation in these patterns was from the ransomware samples. It has been seen that ransomware has a very distinctive memory trace that can easily be distinguished from our artifacts. The following figure contains three different families of 4 ransomware samples producing the same output pattern with the similarity ratio of 0.822487.



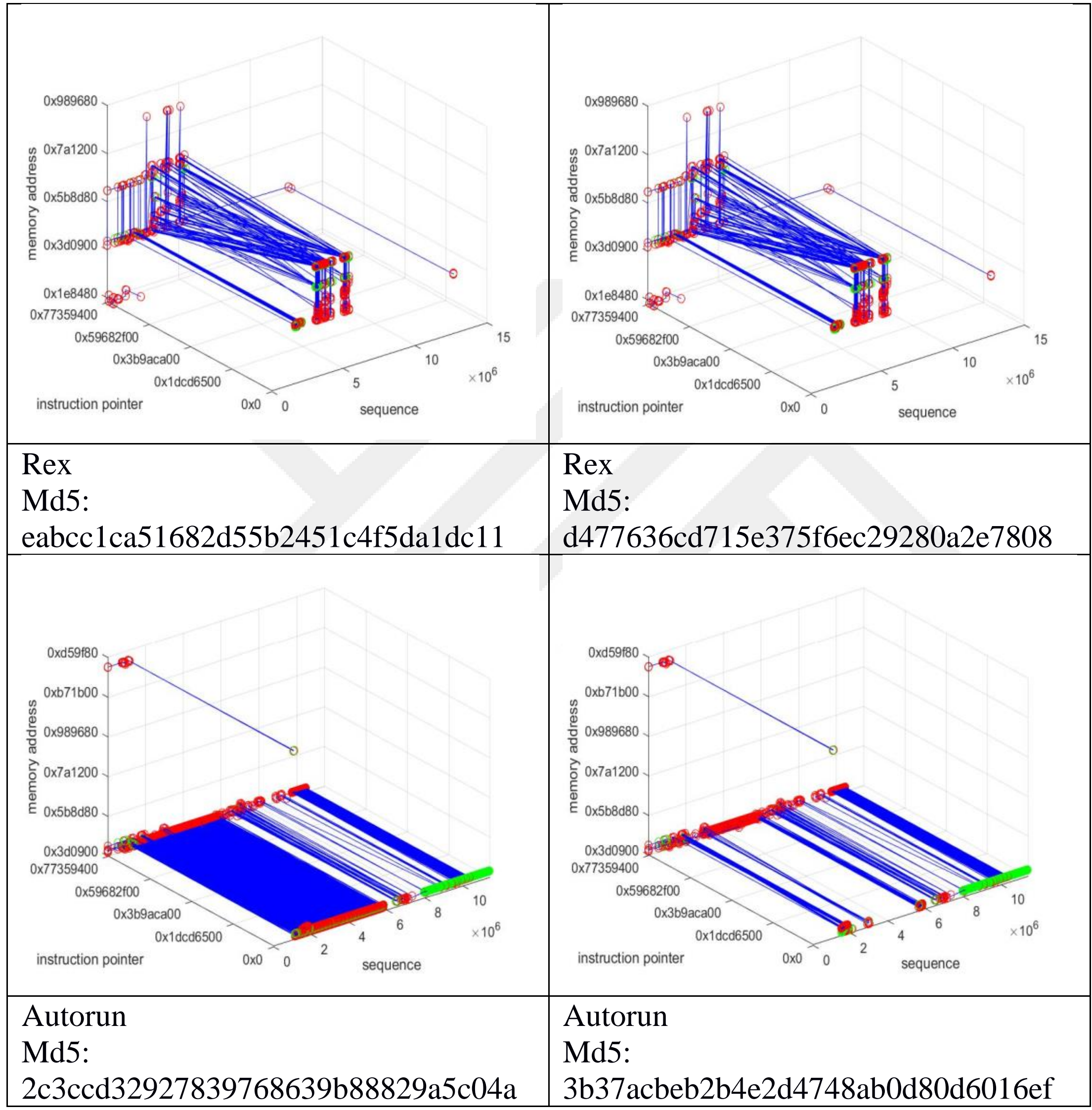
Cryptowall Md5: 3c45e0306f7b8921b7947a1e41b83e65	Locky Md5: 0d1c01615134f26fdc52bd7764b19996
Locky Md5: 1c75d83bf9e55b44b4b90b4b9dcb64a1	TeslaCrypt Md5: 1c8291eb7c1e8a98d3ba12a52e89b429

Figure 7.4 Ransomware Patterns with the pairwise similarity average of 0.822487.

As can be seen from Figure 7.4., these ransomware samples produce red and green straight lines on the same instruction write patterns in their images.

Observed Patterns among Infectors

Similarly, slightly different patterns are also caught from the observations of virus (infector) files. Among the samples, the files that are infecting other files and their images have been extracted. Some of the examples of viruses from the same families are given in pairs in the following Figure 7.5.



Rex Md5: eabcc1ca51682d55b2451c4f5da1dc11	Rex Md5: d477636cd715e375f6ec29280a2e7808
Autorun Md5: 2c3ccd32927839768639b88829a5c04a	Autorun Md5: 3b37acbeb2b4e2d4748ab0d80d6016ef

Figure 7.5. Viruses and Infectors Fingerprints. For the family samples of Rex Virus given above, the similarity rate is 0.999994, and for the Autorun samples, the same ratio is 0.947181.

These results are obtained on a computer with Intel Processor Intel(R) Core (TM) i7-7700HQ CPU @ 2.80GHz, 4 Core(s). In our dataset, nine of the malware could not be mapped into 3d space as the number of points was not feasible to map on our experiment computer. It has been observed in our experimentation that 16 of the

malware from our sample did not run on our sandbox or detected that they are being observed.

7.4. Malware Families

One of the main reasons for this research is to reveal malware family based characteristics in addition to identifying memory operations of malware. For this aim, the average pairwise similarity ratios of the samples coming from the same family are given in the following Figure 7.6. Most of the families in this research resulted in ratios between 0.6 and 0.7, with a minimum similarity ratio of 0.6923085. It is shown that even for a small number of samples in our dataset, our mechanism and the search for memory patterns are evident. This is the main contribution of this dissertation.

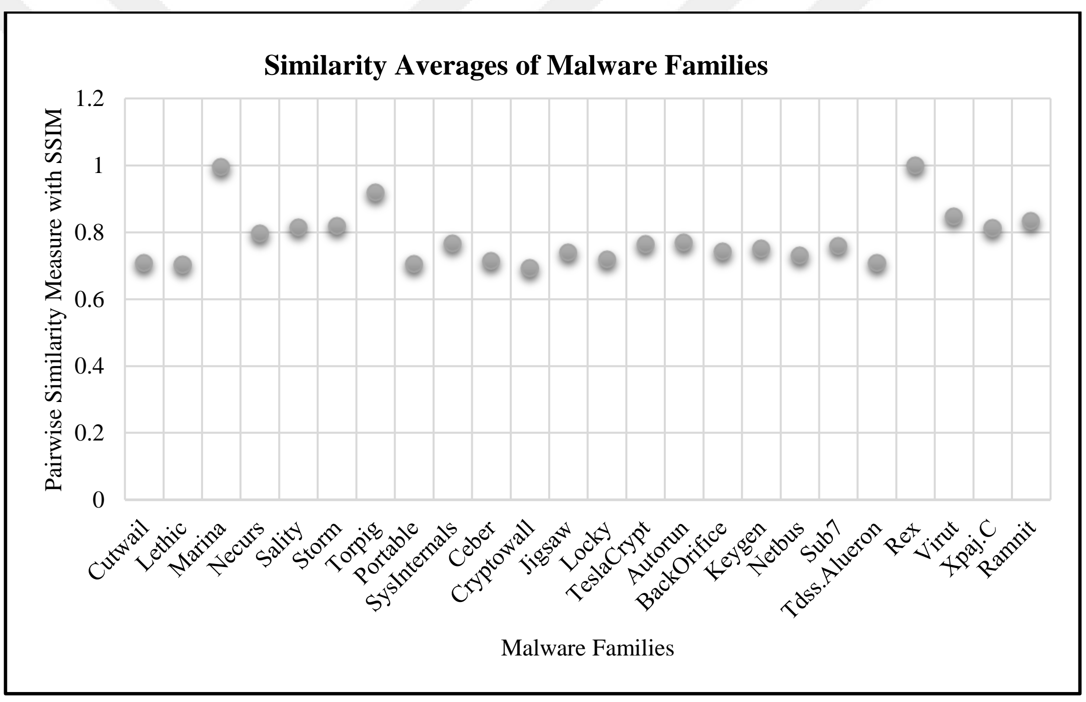


Figure 7.6. Pairwise Similarity ratio average values for all the families in our dataset.

7.5. Discussion

Figure 7.6. shows the average similarity ratio amongst the same malware family. As mentioned in the previous Chapter 7, the average similarity ratio is calculated by comparing all the pairs in a family and getting the arithmetic averages of these pair-wise similarity rates. The lower bound of this measure for our malware dataset is around the value of 0.50. As the SSIM measure is a tool for comparing images and our resulting patterns resides on the same axis structure in each image.

The higher bound is 1.00 which means the resulting patterns are the same.

Figure 7.6. reveals the similarity of the families Rex Virus and Matsnu Trojan produced precisely the same memory images and patterns, although they have employed mechanisms to avoid signature-based detection. Although their cryptographical hash values and behavior analysis differs from sample to sample, their memory usage patterns and representations shows 0.995655667 for Marina Botnet Family and 0.999994 for Rex Virus Family.

Virus Families result in the highest similar images with the average ratio of 0.886347678. The pattern that the virus families produce is very similar to the patterns in Figure 7.5. Botnet Families in our dataset generates the second most similarities amongst their families with an average ratio around 0.82. It is an expected property of a botnet as they are in general designed to tweak their code a little bit at every infection and connect to the same C&C with the same protocol.

There are some striking cross-family similarities in our dataset. It is mainly because of the utilization of the same loader or downloader integrated into several types of malware. As for an example to the cross-family similarities, Figure 7.2 (a) - (b) and Figure 7.3 (a) - (c) can be considered. There are many more examples for the utilization of the same code-piece, and these can be observed from the Github page of our dataset and resulting images. A heatmap of the pairwise similarities of all malware in our dataset is given in Figure 7.7.

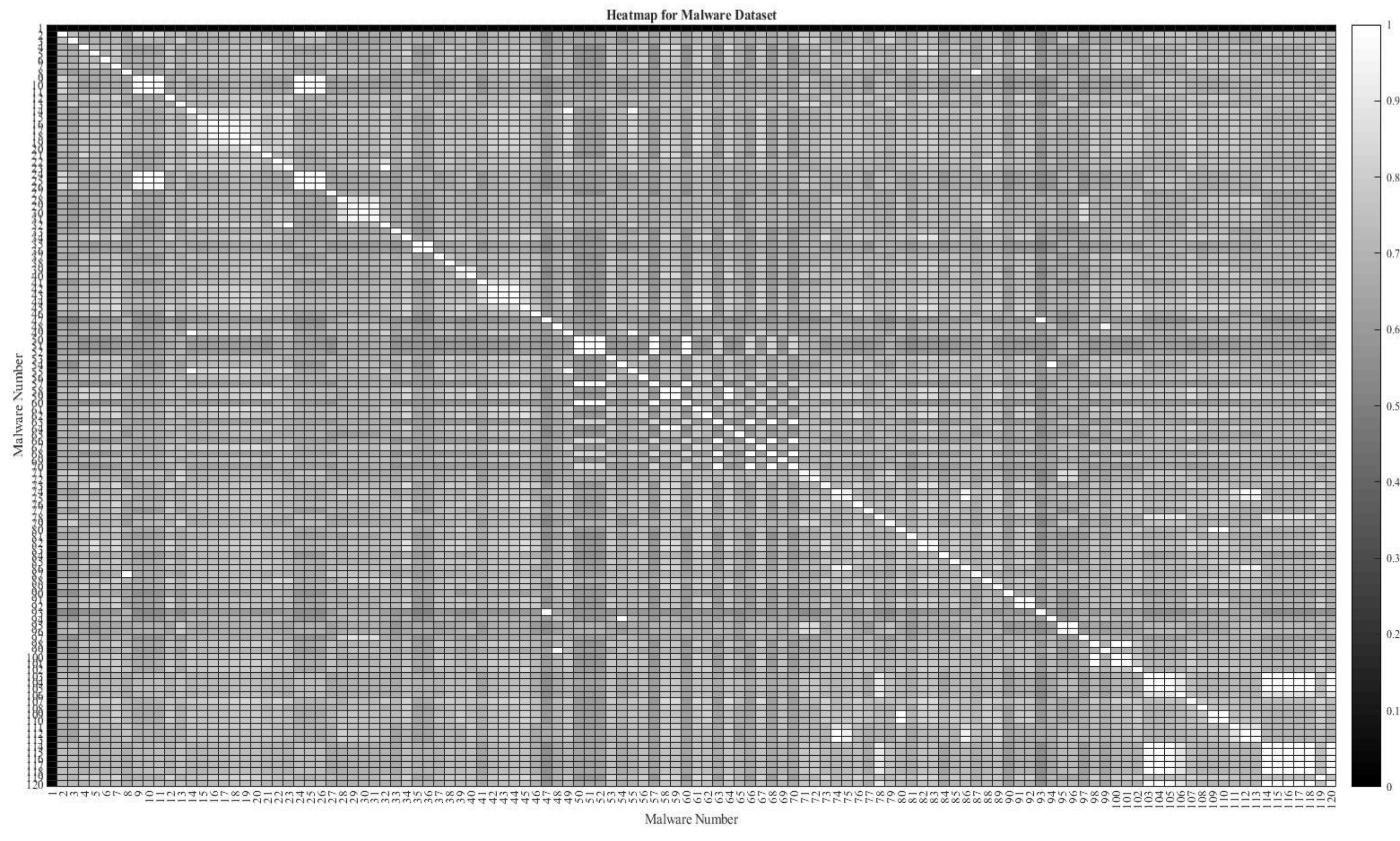


Figure 7.7 - Heatmap of the Malware Dataset Similarities.

On the heatmap in Figure 7.7, the malware are numbered consecutively to their families in the order of Botnets, Cleanware, Ransomware, System Tools, Trojans, Viruses and Worms. The numbers closed to each other represents the malware coming from the same family. Therefore, the tones around the diagonal line are lighter.

Thre rectangular areas with same tones in the heatmap in Figure 7.7 are mostly around the diagonal line. However in these results, our methodlogy shows significant similarities between worms and viruses which can be seen around 103-107 on X-axis and 114-120 on Y-axis. The light scalars distant from the diagonal line typically shows malware coming from different families and have integrated similar piece of codes in their developments. This heatmap and the data that constructs this heatmap can be found in the accompanying GitHub page given in Conclusion Chapter 8.

In this chapter, common indicators of memory operations and similarities of the malicious samples on memory are shown and presented. The next chapter concludes this dissertation with the challenges and future works, as well as emphasizing our contributions in this study.

CHAPTER 8
CONCLUSION

8.1. Problem Definition Re-visited

Contemporary malware detection mechanisms and malware analysis techniques are not able to keep up with the malware authors because of the avoidance techniques employed. Static analysis techniques suffer from obfuscation, packing, and encryption methodologies, and thus, the limitations of static analysis are evident and discussed in Chapter 3.

Dynamic analysis and behavioral detection methods surpass these limitations. However, as the malware authors advance on the malware analysis techniques, new avoidance techniques such as native DLL coding, memory resident, file-less malware came into existence. Because of these reasons, the need for detection and analysis methodologies on the memory is present.

In this dissertation, a memory tracing technique combining dynamic analysis methodologies and memory forensics with dynamic binary instrumentation is employed. A novel method is explained and presented for memory imaging and pattern extraction of the malicious files. It has been shown on Windows PE Executable Malware that similar malware samples coming from the same families show similar memory access patterns in 3d space constructed by the sequence of the access relative to the computation instruction sequence, the instructions address and the memory address that is being accessed.

8.2. Contributions

Samples from a malicious and a benign software dataset experimented with the methodology, and the results are shared in this dissertation. Therefore;

- The main contributions of this research are first providing a novel methodology for extracting memory usage patterns – namely obtaining the digital memory images of the various malware or benign software.

- Secondly demonstrating that the similar malware samples can be detected utilizing these memory images
- Lastly, underlining the fact that the visual patterns of memory utilization of the various software may be utilized in computer forensics research.

The instances, results, and the graph DB access is open to public on, http://github.com/cgyphd/Imaging-and-Evaluating-Memory-Access-for-Malware.

8.3. Discussions and Challenges

The main challenge in this methodology is that the number of malware access entries can grow in the order of $10^{7,}$ therefore to generate and extract patterns in sufficient quantities for machine learning and deep learning becomes unfeasible.

The second challenge is pattern recognition in 3-dimensional data is still a big challenge in the pattern recognition community. Therefore, the technique employed in this dissertation is applied on the 2-d images of 3-d data. Moreover, for 3-dimensional pattern recognition, a graph database is used and integrated into this research for this reason.

The results of the static analysis visualizations are not explored further n this research, as mentioned in Chapter 3. Static analysis has its limitations set by obfuscations, packing and other avoidance schemes that are excessively explained in this dissertation. Therefore, the thesis is directed to and aimed at dynamical analysis methodologies.

This dissertation presents an analysis methodology that is designed for a post-mortem strategy. The analysis and comparisons can be done after the infection has occurred. There is an absolute need for methods and techniques that are capable of identifying and deciding on memory patterns on a live system. It could be achieved through monitoring the state of the process memory either by utilizing external gates to the processors read/write gates by adding hardware components or by adding external software components to the virtual machines and monitoring the behavior of the virtual machine interfaces.

The methodology in this dissertation assumes that the memory patterns reside in 3-dimensional spaces. Another question arises from this research is whether there will be an additional fourth dimension on the memory. This dissertation does not answer

this question. However, the evolution of the malware and observation of this evolution through a collection of malicious patterns of families over years might result in new links between malicious samples and their evolving patterns. These links may reveal new features and new dimensions yet to be discovered. For this aim, the similarity between families can be related to Open Source Intelligence on malware families for the time being. Collected open-source information content will result in better labeling of the patterns, which will, in return, result in better identification of malicious executables.

8.4. Future Work

Spatial properties of the graphs constructed with the memory extraction are demonstrated in Chapter 6. A line of future work is to investigate these properties rigorously and thoroughly. In that sense, the analysis of the malware can be enriched utilizing those properties.

This Ph.D. work hopes that with the advancement of Deep Learning methodologies, especially the state-of-the-art Convolutional Neural Networks (CNN) for image recognition, the cyber intelligence analysts will be able to identify malware in less amount of time than it already takes currently.

The final and maybe the most crucial future work of this Ph.D. is to integrate the results and patterns of this research into big data platforms and live systems to engineering real-time detection systems through memory analysis.

.

REFERENCES

Aghaeikheirabady, M., Farshchi, S. M. R., & Shirazi, H. (2015). A new approach to malware detection by comparative analysis of data structures in a memory image. *2014 International Congress on Technology, Communication and Knowledge, ICTCK 2014*, (Ictck), 1–4. https://doi.org/10.1109/ICTCK.2014.7033519

Asturiano, V. (n.d.). 3d Force Graph Library.

Avoine, G., Oechslin, P., & Junod, P. (2007). *Computer System Security: Basic Concepts and Solved Exercises*. EPFL Press.

Bailey, M., Oberheide, J., Andersen, J., Mao, Z. M., Jahanian, F., & Nazario, J. (2007). Automated Classification and Analysis of Internet Malware. *Recent Advances in Intrusion Detection*, 178–197. https://doi.org/10.1007/978-3-540-74320-0_10

Banin, S., Shalaginov, A., & Franke, K. (2016). Memory access patterns for malware detection. *NISK-2016 Conference*, 12.

Bencsáth, B., Pék, G., Buttyán, L., & Félegyház, M. (2012). *Duqu_Analysis, Detection, and Lessons Learned*.

BitDefender. (2010). Malware History. *BitDefender*, 71. Retrieved from http://download.bitdefender.com/resources/files/Main/file/Malware_History.pdf .

Bletsch, T., Jiang, X., Freeh, V. W., & Liang, Z. (2011). Jump-oriented programming. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11*, 30. https://doi.org/10.1145/1966913.1966919

Brunner, J. (1984). *Shockwave Rider*. Del Rey.

Bruschi, D., Martignoni, L., & Monga, M. (2006). Detecting self-mutating malware using control-flow graph matching. *Informatica*, *4064*, 129–143. https://doi.org/10.1007/11790754_8

Cheng, B., Ming, J., Fu, J., Peng, G., Chen, T., Zhang, X., & Marion, J.-Y. (2018).

Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost. *CCS*. https://doi.org/10.1145/3243734.3243771

Choi, S., Cha, S., & Tappert, C. C. (2010). A Survey of Binary Similarity and Distance Measures. *Group*, *0*(1), 43–48. Retrieved from http://www.iiisci.org/Journal/CV$/sci/pdfs/GS315JG.pdf

Christodorescu, M., & Jha, S. (2003). Static Analysis of Executables to Detect Malicious Patterns. *SSYM'03 Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, 12. https://doi.org/http://doi.org/10.1109/MSP.2010.92

Cuckoo Sandbox. (2019). Cuckoo Sandbox. Retrieved from https://cuckoosandbox.org/

Cymmetria Reseach. (2016). *Unveiling Patchwork – the Copy-Paste Apt*.

Dewdney, A. K. (1989). COMPUTER RECREATIONS. *Scientific American*, *260*(4), 116–119. Retrieved from http://www.jstor.org/stable/24987222

Ding, S. H. H., Fung, B. C. M., & Charland, P. (2016). Kam1n0: MapReduce-based Assembly Clone Search for Reverse Engineering. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*. https://doi.org/10.1145/2939672.2939719

Duan, Y., Fu, X., Luo, B., Wang, Z., Shi, J., & Du, X. (2015). Detective: Automatically identify and analyze malware processes in forensic scenarios via DLLs. *IEEE International Conference on Communications*, *2015-Septe*, 5691–5696. https://doi.org/10.1109/ICC.2015.7249229

Falliere, N., Murchu, L., & Chien, E. (2011). W32.Stuxnet Dossier. *Symantec-Security Response*, *Version 1*.(February 2011), 1–69. https://doi.org/20 September 2015

Farhadi, M. R., Fung, B. C. M., Fung, Y. B., Charland, P., Preda, S., & Debbabi, M. (2015). Scalable code clone search for malware analysis. *Digital Investigation*, *15*, 46–60. https://doi.org/10.1016/j.diin.2015.06.001

FBI. (2018). The Morris Worm 30 Years Since First Major Attack on the Internet. Retrieved April 23, 2019, from https://www.fbi.gov/news/stories/morris-worm-

30-years-since-first-major-attack-on-internet-110218

Freeware, P. (2019). Portable Freeware. Retrieved February 24, 2019, from https://www.portablefreeware.com/

Fujino, A., Murakami, J., & Mori, T. (2015). Discovering similar malware samples using API call topics. *2015 12th Annual IEEE Consumer Communications and Networking Conference, CCNC 2015*, 140–147. https://doi.org/10.1109/CCNC.2015.7157960

Gandotra, E., Bansal, D., & Sofat, S. (2014). Malware Analysis and Classification: A Survey. *Journal of Information Security*, *05*(02), 56–64. https://doi.org/10.4236/jis.2014.52006

GDATAAdvancedAnalytics. (2016). *r2graphity*. Retrieved from https://github.com/GDATAAdvancedAnalytics/r2graphity

Gupta, A., Kuppili, P., Akella, A., & Barford, P. (2009). An Empirical Study ofMalware Evolution. *COMSNETS'09 Proceedings of the First International Conference on COmmunication Systems And Networks Pages 356-365*, 356–365. Retrieved from http://pages.cs.wisc.edu/~pb/comsnets09.pdf

Haq, I., Chica, S., Caballero, J., & Jha, S. (2018). Malware lineage in the wild. *Computers and Security*, *78*, 347–363. https://doi.org/10.1016/j.cose.2018.07.012

Heinricher, A., & Jilcott, S. (2013). *An Evolutionary Trace Algorithm for Constructing Malware Lineages*. 264–269.

Joe Sandbox. (2019). Joe Sandbox. Retrieved June 20, 2019, from https://www.joesecurity.org/

Karbab, E. M. B., Debbabi, M., Derhab, A., & Mouheb, D. (2018). MalDozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*, *24*, S48–S59. https://doi.org/10.1016/j.diin.2018.01.007

Karim, M. E., Walenstein, A., Lakhotia, A., & Parida, L. (2005). Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, *1*(1–2), 13–23. https://doi.org/10.1007/s11416-005-0002-9

Kaspersky. (2019). 1990. Retrieved April 23, 2019, from

https://encyclopedia.kaspersky.com/knowledge/year-1990/

Kawakoya, Y., Iwamura, M., Shioji, E., & Hariu, T. (2013). API Chaser: Anti-analysis resistant malware analyzer. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *8145 LNCS*, 123–143. https://doi.org/10.1007/978-3-642-41284-4_7

Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X., Wang, X., … Antipolis, S. (2009). Effective and Efficient Malware Detection at the End Host. *System*, *4*(1), 351–366. https://doi.org/10.1093/mp/ssq045

Kolosnjaji, B., Zarras, A., Webster, G., & Eckert, C. (2016). Deep Learning for Classification of Malware System Call Sequences. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*: *Vol. 9992 LNAI* (pp. 137–149). https://doi.org/10.1007/978-3-319-50127-7_11

Kolter, J. Z., & Maloof, M. A. (2006). Learning to Detect and Classify Malicious Executables in the Wild. *Journal of Machine Learning Research*, *7*, 2721–2744.

Korczynski, D., & Yin, H. (2017). Capturing Malware Propagations with Code Injections and Code-Reuse Attacks. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, 1691–1708. https://doi.org/10.1145/3133956.3134099

Lanzi, a, Sharif, M., & Lee, W. (2009). K-Tracer: A System for Extracting Kernel Malware Behavior. *Ndss*, 163–169. https://doi.org/http://www.isoc.org/isoc/conferences/ndss/09/pdf/12.pdf

Lecun, Y., & Bengio, Y. (1995). Convolutional Networks for Images, Speech, and Time-Series. In *The Handbook of Brain Theory and Neural Networks* (p. 43).

Li, Y., Sundaramurthy, S. C., Bardas, A. G., Ou, X., Caragea, D., Hu, X., & Jang, J. (2015). Experimental Study of Fuzzy Hashing in Malware Clustering Analysis. *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*. Retrieved from https://www.usenix.org/conference/cset15/workshop-program/presentation/li

Ligh, M. H., Case, A., Levy, J., & Walters, Aa. (2014). The Art of Memory

Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. In *Wiley* (Vol. 1). https://doi.org/10.1007/s13398-014-0173-7.2

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., … Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '05*, *40*(6), 190. https://doi.org/10.1145/1065010.1065034

Microsoft. (2019). Windows System Internals. Retrieved March 2, 2019, from http://www.sysinternals.com

Miles, C. (1986). *Early History of the Computer Virus*. (1984).

Moser, A., Kruegel, C., & Kirda, E. (2007). Limits of static analysis for malware detection. *Proceedings - Annual Computer Security Applications Conference, ACSAC*, 421–430. https://doi.org/10.1109/ACSAC.2007.21

Mosli, R., Li, R., Yuan, B., & Pan, Y. (2016). Automated malware detection using artifacts in forensic memory images. *2016 IEEE Symposium on Technologies for Homeland Security, HST 2016*. https://doi.org/10.1109/THS.2016.7568881

Mosli, R., Li, R., Yuan, B., & Pan, Y. (2017a). A Behavior-Based Approach for Malware Detection. In *IFIP Advances in Information and Communication Technology* (pp. 187–201). https://doi.org/10.1007/978-3-319-67208-3_11

Mosli, R., Li, R., Yuan, B., & Pan, Y. (2017b). A Behavior-Based Approach for Malware Detection. In *Advances in Digital Forensics IV* (pp. 187–201). https://doi.org/10.1007/978-3-319-67208-3_11

Narudin, F. A., Feizollah, A., Anuar, N. B., & Gani, A. (2016). Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, *20*(1), 343–357. https://doi.org/10.1007/s00500-014-1511-6

Neumann, J. Von. (1969). Theory of self-reproducing automata. *Information Storage and Retrieval*, Vol. 5, p. 151. https://doi.org/10.1016/0020-0271(69)90026-6

Newsome, J., & Song, D. (2005). Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *Analysis*, *44*(May 2004), 2–3. https://doi.org/10.1.1.62.8372

Nguyen, M. H., Nguyen, D. Le, Nguyen, X. M., & Quan, T. T. (2018). Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning. *Computers and Security*, *76*, 128–155. https://doi.org/10.1016/j.cose.2018.02.006

Ni, S., Qian, Q., & Zhang, R. (2018). Malware identification using visualization images and deep learning. *Computers and Security*, *77*, 871–885. https://doi.org/10.1016/j.cose.2018.04.005

Ortega, A. (2016). Paranoid Fish. Retrieved February 24, 2019, from https://github.com/a0rtega/pafish

Park, Y., Reeves, D., Mulukutla, V., & Sundaravel, B. (2010). Fast malware classification by automated behavioral graph matching. *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, 1–4. https://doi.org/10.1145/1852666.1852716

Park, Y., Reeves, D. S., & Stamp, M. (2013). Deriving common malware behavior through graph clustering. *Computers and Security*, *39*(PART B), 419–430. https://doi.org/10.1016/j.cose.2013.09.006

Pektaş, A., & Acarman, T. (2017). Classification of malware families based on runtime behaviors. *Journal of Information Security and Applications*, *37*, 91–100. https://doi.org/10.1016/j.jisa.2017.10.005

Pietrek, M. (2011). *Peering Inside the PE : A Tour of the Win32 Portabl ... Peering Inside the PE : A Tour of the Win32 Portable Executable File Format Peering Inside the PE : A Tour of the Win32 Portabl ...* (1), 1–21.

Pirscoveanu, R. S., Hansen, S. S., Larsen, T. M. T., Stevanovic, M., Pedersen, J. M., & Czech, A. (2015). Analysis of malware behavior: Type classification using machine learning. *2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, 1–7. https://doi.org/10.1109/CyberSA.2015.7166128

Risak, V. (1972). Selbstreproduzierende Automaten mit minimaler Informationsübertragung. *Elektrotechnik Und Maschinenbau*, *89*(11), 449–457.

Rughani, V., & Rughani, P. H. (2017). *AUMFOR : Automated Memory Forensics for Malware Analysis*. *6*(2), 36–39.

Santos, I., Brezo, F., Ugarte-Pedrero, X., & Bringas, P. G. (2013). Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, *231*, 64–82. https://doi.org/10.1016/j.ins.2011.08.020

Sarantinos, N., Benzaïd, C., Arabiat, O., & Al-Nemrat, A. (2016). Forensic malware analysis: The value of fuzzy hashing algorithms in identifying similarities. *Proceedings - 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 10th IEEE International Conference on Big Data Science and Engineering and 14th IEEE International Symposium on Parallel and Distributed Proce*, (July 2018), 1782–1787. https://doi.org/10.1109/TrustCom.2016.0274

Schultz, M. G., Eskin, E., Zadok, E., & Stolfo, S. J. (2001). Data mining methods for detection of new malicious executables. *Proceedings. 2001 IEEE Symposium on Security and Privacy, 2001. S&P 2001.*, 38–49. https://doi.org/10.1109/SECPRI.2001.924286

Sergi Alvarez. (2006). Radare2 Reverse Engineering Framework. Retrieved December 7, 2018, from https://rada.re/r/

Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., & Elovici, Y. (2012). Detecting unknown malicious code by applying classification techniques on OpCode patterns. *Security Informatics*, 1–22.

Sharp, R. (2007). An Introduction to Malware. In *Lecture Notes of DTU02233 Network Security*. https://doi.org/10.1002/9781119183433.ch9

Shijo, P. V., & Salim, A. (2015). Integrated static and dynamic analysis for malware detection. *Procedia Computer Science*, *46*(Icict 2014), 804–811. https://doi.org/10.1016/j.procs.2015.02.149

Smith, C., & Matrawy, A. (2008). Computer worms: Architectures, evasion strategies, and detection mechanisms. *Journal of Information …*, *4*, 69–83. Retrieved from http://www.softcomputing.net/jias/smith.pdf

Stevens, R. M., & Casey, E. (2010). Extracting Windows command line details from physical memory. *Digital Investigation*, *7*(SUPPL.). https://doi.org/10.1016/j.diin.2010.05.008

Symantec. (2015). *Dyre : Emerging threat on financial fraud landscape*.

Szor, P. (2005). *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional.

Teller, T., & Hayon, A. (2014). *Enhancing Automated Malware Analysis Machines with Memory Analysis*. 1–5.

Tenebro, G. (2009). W32.Waledac Threat Analysis. In *Security Response Articles*.

Thompson, K. (1984). Reflections on trusting trust. *Communications of the ACM*, *27*(8), 761–763.

Touchette, F. (2015). The Evolution Of Malware. Retrieved May 5, 2019, from https://www.darkreading.com/risk/the-evolution-of-malware/a/d-id/1322461

Uppal, D., Sinha, R., Mehra, V., & Jain, V. (2014). Malware detection and classification based on extraction of API sequences. *Proceedings of the 2014 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2014*, 2337–2342. https://doi.org/10.1109/ICACCI.2014.6968547

ViruSign. (2019). Virusign. Retrieved February 24, 2019, from https://www.virusign.com/

Walenstein, A., & Lakhotia, A. (2007). The software similarity problem in malware analysis. *Proceedings Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software*, 1–10. Retrieved from http://drops.dagstuhl.de/opus/volltexte/2007/964/

Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004). Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, *13*(4), 600–612. https://doi.org/10.1109/TIP.2003.819861

Wei, Y., Zheng, Z., & Ansari, N. (2008). Revealing packed malware. *IEEE Security and Privacy*, *6*(5), 65–69. https://doi.org/10.1109/MSP.2008.126

Xin, B., & Zhang, X. (2007). Efficient online detection of dynamic control dependence. *Proceedings of the 2007 International Symposium on Software Testing and Analysis - ISSTA '07*, 185. https://doi.org/10.1145/1273463.1273489

Yadegari, B., Stephens, J., & Debray, S. (2017). Analysis of Exception-Based Control Transfers. *CODASPY*, 205–216. https://doi.org/10.1145/3029806.3029826

Ye, Y., Wang, D., Li, T., & Ye, D. (2007). IMDS: Intelligent Malware Detection System. *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1043–1047. https://doi.org/10.1145/1281192.1281308

Yin, H., Song, D., Egele, M., Kruegel, C., & Kirda, E. (2007). Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, 116–127. https://doi.org/http://doi.acm.org/10.1145/1315245.1315261

You, I., & Yim, K. (2010). Malware obfuscation techniques: A brief survey. *Proceedings - 2010 International Conference on Broadband, Wireless Computing Communication and Applications, BWCCA 2010*, 297–300. https://doi.org/10.1109/BWCCA.2010.85

Zaki, A., & Humphrey, B. (2014). Unveiling the kernel : Rootkit discovery using selective automated kernel memory differencing. *Virus Bulletin*, (September), 239–256.

Zakorzhevsky, V. (2010). Analysis Article on Virut.ce. Retrieved July 6, 2019, from Securelist.com website: https://securelist.com/review-of-the-virus-win32-virut-ce-malware-sample/36305/

Zerowine. (2019). Zerowine. Retrieved June 21, 2019, from http://zerowine.sourceforge.net/

Zhu, Y., Gladyshev, P., & James, J. (2009). Temporal Analysis of Windows MRU Registry Keys. In *Advances in Digital Forensics IV* (pp. 83–93). https://doi.org/10.1007/978-3-642-04155-6_6