A

Ph.D Thesis

on

Behavior-based Dynamic Malware Detection Techniques

Submitted for partial fulfillment for the degree of

Doctor of Philosophy

(Computer Science and Engineering)

in

Department of Computer Science and Engineering

(2014 – 2015)

Supervisors:
Vijay Laxmi
Muttukrishnan Rajarajan

Submitted by:
Smita Naval
(2012RCP9013)

MALAVIYA NATIONAL INSTITUTE OF TECHNOLOGY, JAIPUR

Declaration

I, Smita Naval, declare that this thesis titled, "Behavior-based Dynamic Malware Detection Techniques" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a Ph.D. degree at MNIT.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at MNIT or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this Dissertation is entirely my own work.
- I have acknowledged all main sources of help.

Signed: _____

Date: _____

Abstract

Advances in digital technology have made malware program an easy source of financial gains by selling personal and private information stolen from infected host systems. A Malware is a software program that is capable of hampering our system's integrity, availability and confidentiality. The malware authors keep on enhancing the anti-detection capabiliites of malware to maximize their monetary benifits and to defeat anti-malware solutions. Malware samples embedded with these anti-detection features are termed as next-generation malware, a new class of threats that exploit the limitations of existing anti-malware techniques to evade detection. Unfortunately, current anti-malware technologies are inadequate to face modern malware. Therefore, in this thesis we propose novel malware detection techniques that complement exsiting security solutions.

Static approaches for malware detection are vulnerable to obfuscation techniques such as packing, code obfuscation, polymorphism and metamorphism. To nullify the impact of obfuscation, security researchers started to explore dynamic approaches. The dynamic malware detection approaches enable us to understand the run-time behavior of program binaries. We also, aim to identify malware on the basis of their behavior. For this, we utilize system-call sequences since system-calls provide a non-bypassable interface between user applications and OS. We propose a dynamic host-based approach for categorizing malware samples on the basis of their run-time behavior. Formed categories indicate that within malware families, the samples constitute different behavior due to their infection and anti-detection behavior. We compute a distance matrix using Dynamic Time Warping (DTW) algorithm to form these groups. In conjunction to that, the proposed approach also discriminates malware and benign executables. Moreover, to improve the performance of proposed approach, we develop a parallel-version of DTW algorithm as we observe that to align large system-call sequences DTW is computationally expensive.

Majority of dynamic behavior detectors do not account for two important anti-detection features of modern malware *i.e.*, 1) system-call injection attack and 2) Environment-aware malware behavior. The former category of malware samples inject irrelevant and independent system-calls during the program execution thus defeating the existing system-call based detection approaches. To address this problem, we propose an evasion-proof solution which is not vulnerable to system-call injection attacks. Our proposed approach precisely characterizes the program semantics using Asymptotic Equipartition Property (AEP) mainly applied in information theoretic domain. The AEP allows us to extract the information-rich call sequences which are further quantified to detect the malicious binaries. In latter category, malware binaries sense the presence of synthetic (non-real/virtual) environment and do not deliver actual malicious payload. We propose an approach that detects and categorizes malware on the basis of their environment-reactive behavior. We design a decision model that labels a program binary either clean, malicious or having environment-reactive behavior.

We use three performance measures *i.e.*, detection accuracy, evaluation with other datasets, and system overhead to evaluate our proposed approaches. Our evaluation indicates that the proposed approaches are effective in identifying real instances of malware binaries.

Dedications

This thesis is dedicated to my husband, Kamal, who has been constant support and source of motivation during the challenges of Phd. I am truly thankful to him for encouraging me to work hard for things that I aspire to achieve.

Acknowledgements

This doctoral thesis would have been impossible without the support, reviews and constructive criticisms of many persons. It is a pleasure to thank all of them here. First of all, I offer my sincere gratitude to my supervisors. First and foremost, I especially want to thank my principal supervisor, ***Dr. Vijay Laxmi***, for her valuable guidance and suggestions in my research work. She broadened my perspectives in research and provided invaluable comments to overcome the challenges that I faced. Her sincere consideration of my family life during my PhD was greatly appreciated. In addition, the progress I achieved in writing was mostly due to her thorough and critical reviews of my manuscripts and thesis. Secondly, I would like to express my gratitude to my external supervisor ***Dr. Muttukrishnan Rajarajan***, City University of London, who helped through his profound feedback and comments that polished my ideas. He helped me to clarify and organize my research, which was an essential step to preparing for my thesis work.

My special thanks to ***Dr. Manoj Singh Gaur*** for his practical directions and discerning knowledge about my research work. He taught me the real meaning of research and always motivated to research the solution until it reaches its full potential. Despite his incredibly busy schedule, Prof. Gaur was always there to help me to resolve my research problems.

I would like to express my gratitude to ***Dr. Mauro Conti*** for reviewing my research papers and providing his valuable suggestions. His guidance at each and every step was instrumental in systematic and successful completion of my work. Also, ***Dr. Girdhari Singh*** and ***Dr. Vijay Janyani*** deserve special thanks as my research committee members and advisors. I thank them for sparing time in providing me with valuable comments to give the required direction to my work.

I was fortunate to meet wonderful friends and fellow researchers, Rimpy, Gaurav Singal, Sonal Yadav, Anil Saini, Manoj Bohra, Chagan Lal, and Lokesh Sharma. I would like to thank all of them. In particular, my special thanks to Rimpy for helping me by her positive and important suggestions during my low phases in both research and personal life. Also, my thanks to Gaurav Singal for his selfless and generous support.

Last but not the least, I could not have finished my study without the enduring support of my family. I deeply appreciated the love and support of my husband, kamal, who supported me in every possible way. A special thanks to my mother, father, my mother-in-law, and father-in-law for their direct and indirect support and love. The same love I feel for my sisters, my brother, my sisters-in-law, my nieces and nephews.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Malware is a software program that is a persistent threat to any computer system's integrity, confidentiality, and availability [1]. The malware programs have become chronic problem for today's web connected systems. This is the consequence of malware's evolution into a more covert and cultivated malicious behavior.

The malware programs came into existence in the year 1981 when floppy disks were infected by a virus named Elk Cloner. Prior to that, malware programs were developed to show the technical and security skills of the developer. In 1988, Morris worm became first malware that caught the attention of mainstream computer users as it infected most of the Internet of that time [2]. In following years, malware became a severe threat to Internet and computer community as everytime the damage caused by the malware was more intense than the previous one. In such way, malware programs are continuously spreading on the Internet with staggering speed and causing billion dollars of financial damage.

Malware writers have strong monetary, political and anti-business motives behind such evil creations. They create platform-specific malicious codes. According to *virustotal* [3] statistics, malware attacks against Windows binaries have the largest share as compared to other binaries. Now, two questions arise from this. First, do we still need PCs (laptops and Desktops platforms) when the current scenario belongs to mobile platforms? Second, do we require Windows OS? Responses will be

"yes" in both the cases. Though mobiles are equipped with network connectivity and can store a good amount of data, these do not support software development to a great extent.

1. PCs are enterprise friendly compared to mobile platforms in terms of using sophisticated applications. The processor chips (Exynos, Qualcomm, Nvidia' Tegra and Apple's A6X) in mobiles, are still in their nascent phase as compared to Intel and AMD chips.
2. Windows is the most popular operating system for desktop and laptop systems. This OS is likely to remain in existence for as long as desktop and laptop systems are in use.

The malicious Windows binaries infect connected nodes of the network. The compromised infected nodes thus become the source of infection and through IP scanning and vulnerability snooping, new victims of malware attacks are finalized [4]. Risks from malware such as data loss, data tampering, data breach and spying are increasing day by day. These malware attacks [5] are categorized as, 1) Hit-Run attacks and 2) Supply-chain attacks.

Former attacks include pay per click, ad pop-ups and high-cost dialing. These attacks are limited to the group of randomly selected victims. Latter attacks are specific and target the chain of victims that belong to various organizations such as security agencies, software and IT firms, defence organizations, gas and energy industries, etc.

According to Kaspersky report [6] new malware is focusing on diplomatic and governmental agencies of various countries across the world. These next-generation malware programs are highly complex and capable of fighting against the available security solutions. Flame [7] and Stuxnet [8] are the examples of such kind of malcodes. These programs are created to carry out cyber espionage for gathering sensitive data from different organizations. Due to these attacks, security and privacy of our data and system become the key issues to be looked upon.

To deal with malware programs, researchers have explored all the domains of malware analysis. These domains include detection, prevention, real-time response,

damage-repair, reduction of false-positives (negatives), and many others. Out of all the domains, detection is the first line of defense against malware programs. According to the advanced malware survey conducted by McAFee at BlackHat 2014, the malware detection is the first choice of the security researcher. Figure 1.1 presents a pie-chart of domains of research interests as assessed from BlackHat conference.

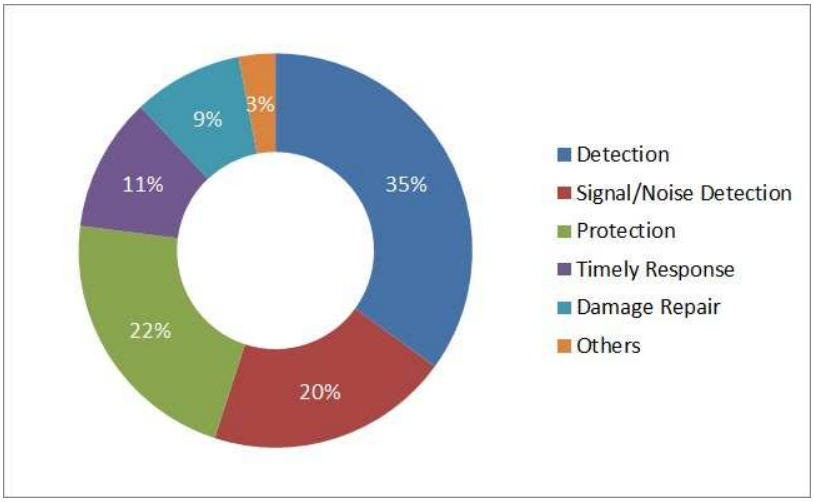


FIGURE 1.1: Advanced malware survey at BlackHat 2014.

1.1 Motivation

In order to avoid any infection or damage to our computer systems, various malware detection solutions have been developed. Unfortunately, the increasing amount and diversity of malware render various malware detection techniques, ineffective [9]. Existing anti-malware solutions are either *static* or *dynamic*. The static malware detection approaches rely on the signature databases of known malware samples. A malware signature is a sequence of bits, bytes or patterns that uniquely identify any malware. The malware signatures are used as the fingerprint of a particular malware. The size of signature databases grows as the number of new malware samples increases exponentially. These databases need to be updated and distributed frequently to detect malware samples.

In past few years, signature-based methods proved to be ineffective as malware authors are also continuously searching the limiting constraints of malware detectors to avoid detection. Polymorphism [10], metamorphism [11], packing [12], and many other code-obfuscation [13] techniques are the examples of such constraints. Malware armed with these techniques make static signature based detection, an NP-complete problem [14]. However, some non-signature based approaches [15, 16] have shown the effectiveness of their approaches against polymorphic and metamorphic malware detection. But, these approaches are vulnerable to zero-day attacks and unknown variants of known malware [17]. Therefore, the static malware detection approaches produce high false alarms.

To nullify the effects of obfuscation, packing, polymorphism, and metamorphism on malware executables, researchers have given preference to dynamic malware detection approaches. In particular, the dynamic behavior-based malware detection approaches utilize the semantics of a malware program by examining its runtime interaction with system objects, resources, and services [18] as these approaches are well suited for capturing new and unseen malware variants that are semantically similar malware variants.

Factors motivating work proposed in this thesis are summarized as follows:

1. The exponential increase of malicious threats: In 2013, AV-test Institute discovered a total of ∼100 million new malicious files and this number has reached ∼140 million in 2014 [19]. This explosion of completely new malware threats and variants of existing malicious programs causes substantial damage.

2. Static malware detection approaches look for syntactic markers to identify malware. These markers are rendered ineffectual while dealing with current malicious threats equipped with various obfuscation techniques.

3. Existing AV solutions depend mostly on signature databases that need to be updated frequently. In order to detect malware programs, the malware signatures need to be distributed to user systems. Signature database size is directly proportional to the number of malware samples and are becoming

very large. For instance, ClamAV distributes more than 120 TB of signatures [20]. Keeping such a large database in main memory is not feasible. Comparing a sample with large database slows down AV performance making users stop scans prematurely and exposing their systems.

4. Dynamic approaches for malware detection rely on the actual behavior of binaries. These approaches are not vulnerable to obfuscation techniques as the program behavior is observed by executing it in a safe virtualized environment. As a result, these approaches provide better detection solution for capturing unseen variants of known malware programs.

5. Non-signature based approaches are effective in identifying detection-aware malware threats [21].

Due to aforementioned reasons, we present behavior-based dynamic malware detection solutions that rely on program semantics instead of syntactic markers. Figure 1.2 shows our research plan.

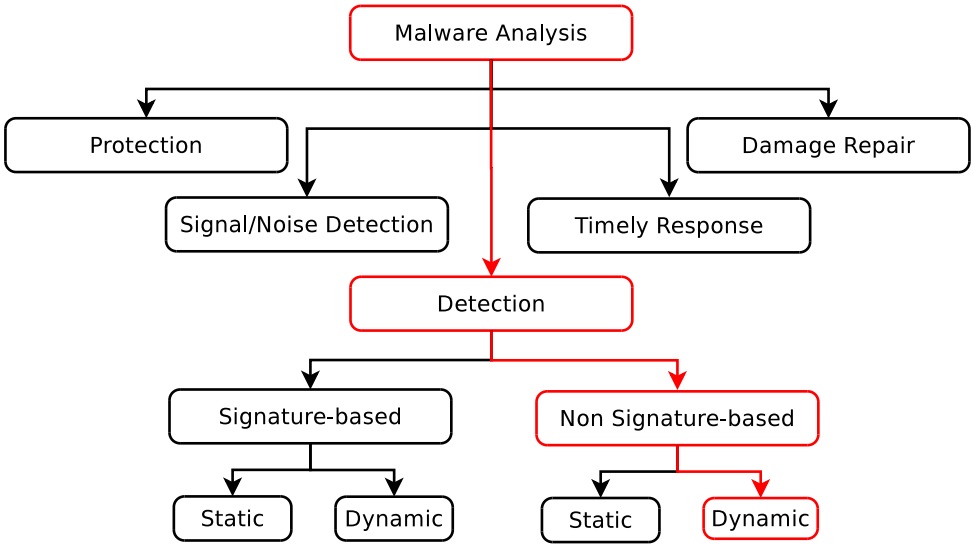


FIGURE 1.2: Research plan

1.2 Objectives

As discussed earlier, dynamic malware detection approaches are preferred by researchers. But these also have certain limitations. Malware programs equipped

with run-time anti-detection features can evade dynamic approaches as well. Due to these anti-detection features malware programs show multiple behaviors during run-time. These behaviors depend on presence of an AV scanner and/or virtualized/emulated environment that may monitor malware's behavior. In this thesis, we first validate our heuristic that samples of same malware family constitute different behaviors under same virtualized environment. We, then, propose approaches that address two anti-detection features of the modern malware. Following are the prime objectives of our research work.

1. To develop non-signature based dynamic malware detection approaches that cater to differentiate malware and benign programs.

2. To develop a technique that can be used to cluster together different behavior groups within a malware family by exploiting runtime behaviors of malware binaries. Malware families we have considered are virus and worms. Even though different samples of virus (worm) groups form different clusters, these clusters are still distinct from cluster formed by benign programs. This technique can be used for eliminating samples that are not too close to benign cluster prior to in-depth analysis for the samples not properly classified for overall speedup of classification.

3. To develop a detection solution that aims to identify malware programs with environment-reactive behavior. The malware programs with environment-reactive behavior do not have normal execution in virtualized environment. In this work, we have tested our method for a total of four behaviors.

4. To construct a detection technique that employs the program semantics of binaries. In addition to this, the proposed approach is also resilient to system-call injection attack (runtime anti-detection feature of malware programs).

To validate the effectiveness of proposed approaches, we need to evaluate their overall performance. This requires a dataset encompassing a reasonable number of samples from all classes and with possibly every expected behavior. The performance of proposed detection solutions is evaluated through real malware and benign samples. The malware samples considered belong to different datasets.

The performance metrics considered for evaluation are – 1) detection accuracy, 2) evaluation with other datasets, and 3) system overhead. Also, a comparison of proposed solutions with existing state-of-the-art dynamic malware detection approaches is also provided.

1.3 Contributions

This section provides an overview of our contributions resulting from the work presented in this thesis. We have proposed various dynamic malware detection approaches, which aim to address anti-detection features of modern malware. It basically first explores that during runtime malware samples exhibit multiple behaviors in spite of belonging to same malware family. The discrepancies in behaviors arise due to the malware samples embedded with various anti-detection features. Further, we focus on two anti-detection features *i.e.*, environment-reactive and system-call injection. To impart generality to our solutions, we have used more than one malware dataset and evaluated the performance of our approaches.

Figure 1.3 illustrates the work flow of the thesis. The dataset preparation, behavior monitoring and behavior modelling are the steps common to all the proposed approaches of the thesis. The major contributions from our research work can be summarized as follows.

1. We have carried out an extensive survey on the existing behavior-based malware detection approaches. Based on our literature review, we have observed that most of the proposed approaches detect the malware on the basis of its run-time behavior. These approaches fail to detect new malware samples embedded with anti-detection features. We provide solutions that address these new and improved class of malware.

2. We have proposed a detection and categorization approach [C-1] that is based on the heuristic, which says that the samples of a malware family exhibit different behaviors. These behaviors are due to the anti-detection features

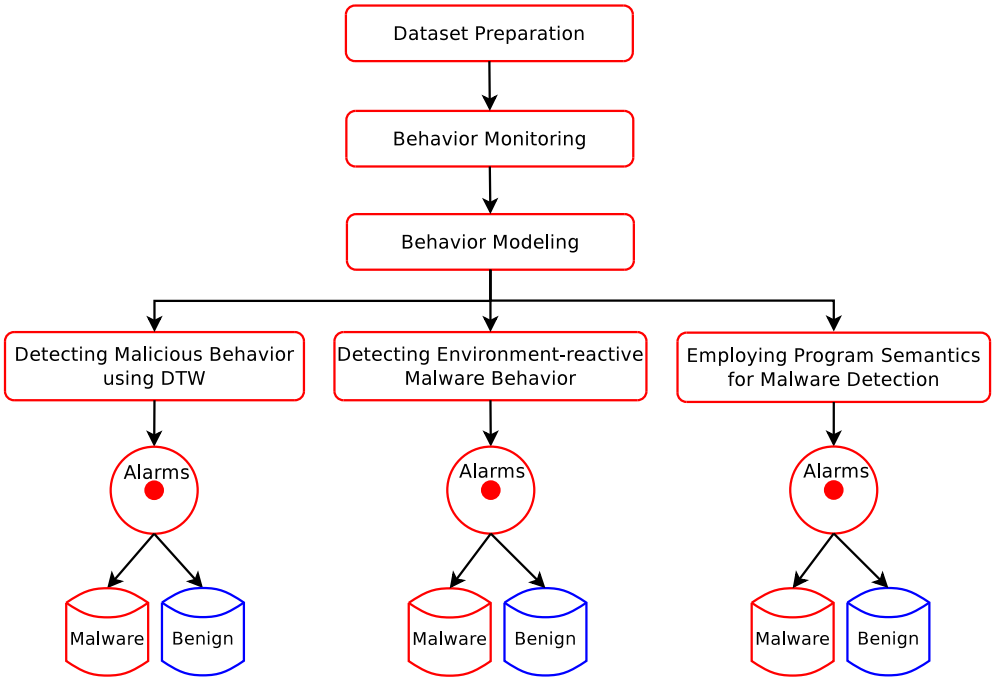


FIGURE 1.3: Work-flow

present in the malware samples. The proposed approach exploits the non-uniformity in execution traces of malware samples. For this, we utilize Dynamic Time Warping (DTW) algorithm that generate distance score between two samples. To reduce the computational complexity of DTW, we propose and implement the parallel algorithm of DTW (P-DTW). The working and results of proposed approach are presented in Chapter 3.

3. We have proposed an approach [C-2] that identifies and categorizes the malware samples exhibiting environment-reactive behavior. The proposed approach utilizes malware's tactics of evading detection for predicting its reactive and malicious behavior. For this, a multi-class model is prepared that makes use of the multi-layer perceptron learning algorithm with error back propagation. The experimental results (discussed in Chapter 4) indicate that the proposed model is capable of finding the known and unknown instances of malware binaries, which are environment-reactive.

4. We have also proposed a malware detection approach [J-1] that is resilient against system-call injection attacks. The proposed approach characterizes the program behavior in terms of semantically-relevant paths employed to build and train our detection model. For this, we apply the asymptotic

equipartition property (AEP-a concept adopted from information theory). The concept of AEP is used to extract semantically-relevant paths that depict program behavior. Further, we employ Average Logarithmic Branching Factor (ALBF) to construct our feature space. The detailed discussion of the proposed approach is presented in Chapter 5.

1.4 Thesis Structure

The remaining of the thesis is structured as follows. In Chapter 2, we present a brief introduction to modern malware and infection strategies adopted by them. Chapter 2 also includes a detailed survey of existing dynamic malware detection techniques. Chapter 3 discusses our first malware detection and categorization approach through application of DTW on malware behavior captured through trace of system-call during runtime. The malware detection technique on the basis of its environment-reactive behavior has been presented in Chapter 4. Using an information theoretic approach, Chapter 5 presents a malware detection technique by employing program semantics. In addition to this, it also shows that the proposed approach is resilient to system-call injection attacks. Finally, Chapter 6 offers the conclusions of this thesis drawn based on the presented work and provides pointers for future research.

Chapter 2

Malware Detection Techniques: A Review

Before exploring the proposed approaches of detecting malware, this chapter provides the background information on contemporary malware and its infection strategy as well as the terms used in this thesis. After introducing the recurring concepts of the thesis, we present a comprehensive review of the published research work in the domain of dynamic malware detection.

2.1 Contemporary Malware

In past four decades, malware has been transformed into its present complex form by learning and adapting to new tactics of spreading infection and evading detection. These modern malware programs are coupled with many capabilities that are used to protect them from available security solutions. These vicious codes with similar behavior, propagation mechanism, the payload carried, and used infection mediums are grouped and labeled with the same name. In Table 2.1, a brief summary [22–29] of malware families such as virus, worms, trojans, rootkits, bots, and spyware/adware has been presented. This table summarizes typical attacks, characteristics, sub-divisions, and examples of each family. These malicious

TABLE 2.1: Malware summary

Malware	**Characteristics**	**Sub-type**	**Examples**
Viruses	1) Self-replicating. 2) Rely on other host programs for propagation and infection. 3) Can infect local and system files.	File viruses Boot-sector viruses Email Viruses Macro viruses	Sunday, cascade Parity boot, Disk killer Melissa, ILoveYou Concept
Worms	1) Self-replicating. 2) Do not rely on other program for infection. 3) Target vulnerable machines in the network. 4) Exhaust system resources and network bandwidth.	Internet Worms p2p Worms Email Worms IM Worms	Code Red I&II Benjamin MyDoom, Love Letter Choke, Serflog
Trojans	1) Do not replicate. 2) Parasitic in nature. 3) Pretend to be benign in the system. 4) Perform a malicious task in the background.	Remote Access (RATs) Data-Sending Destructive Denial of Service Security S/W Disablers	Back Oriface Badtrans.B Goner RFpoison, W32/Trinoo Bugbear
Rootkits	1) Apply self-hiding mechanism. 2) Have root privileges. 3) Can modify system objects like SSDT, IDT, GDT, etc.	User Mode Kernel Mode Firmware Hypervisor	Qoolaid, lkr, trOn Da Ios
Bots	1) Provide remote access to its creator (Bot Master). 2) Can communicate with other bots using Botnets. 3) Replicate themselves using botnets.	Click Frauds Denial of Service Spamming Phishing Distributing Malware Data Stealing	Clickbot.A Hameq, Waledac Spambot Pushbot Rimecud Rbot, Zbot
Spywares & Adware	1) Do not replicate. 2) Spy on user's sensitive information using ads/pop-ups. 3) Collect and transmit information to the attacker.	Cookies Browser hijacker Games Ad popups Keyloggers	BrilliantDigital BonzaiBuddy Elf Bowling Weatherbug, CoolWebSearch Zlob

binaries can appear in three forms: 1) Known malware, 2) Unseen variants of known malware and 3) New malware instances. All three forms undergo phases of malware life-cycle. Figure 2.1 describes the journey of a typical malware from penetration to activation stage. Throughout its journey of infecting a machine, a malicious program locates a medium to enter into the system. Second, it applies covert launching methods to initiate the execution. Third, it delivers the malicious payload. Finally, once a machine is compromised, malware is ready to infect a new computer. Figure 2.1 shows that difference in infection mechanism of old and new malware. In this figure, "white circle" is employed only by contemporary malware while the "black circle" is employed with the both obsolete and new malware.

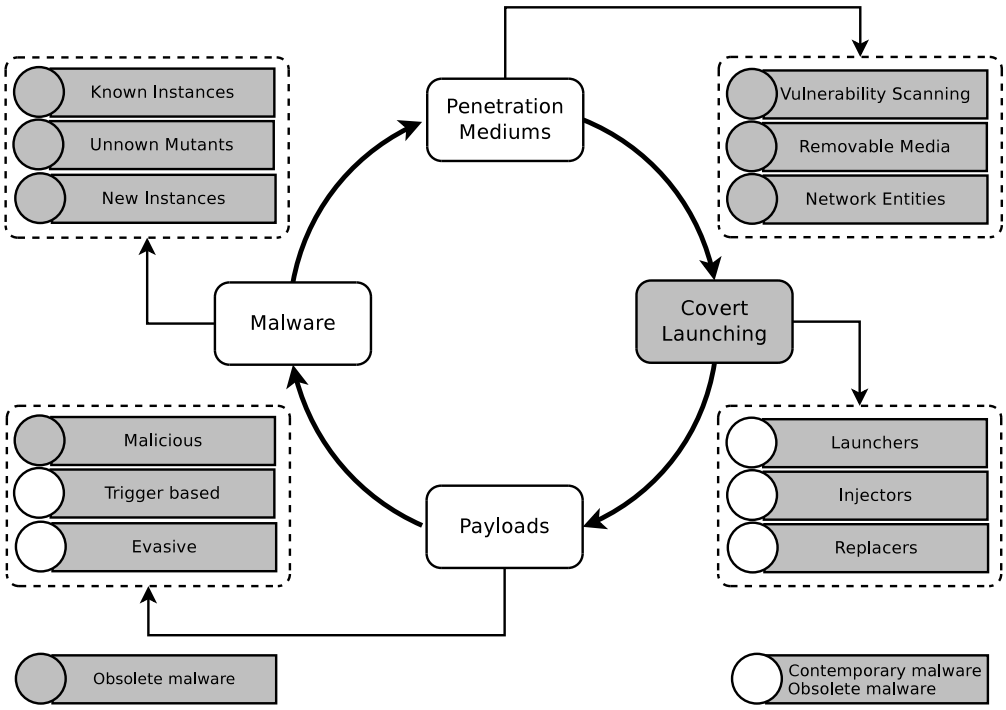


FIGURE 2.1: Malware life-cycle

2.1.1 Penetration Mediums

Malware makes use of various sources to launch its attacks. The sources used by malware to gain access to a computer are known as infection mediums. The malware utilizes following mediums for infection.

- ***Vulnerability Scanning:*** Vulnerability is a security bug within a software allowing malware to gain entrance into the system. Malware scans applications and exploits their flaws. The buffer overflow [30], dangling pointers [31], privilege escalation [32], cross-site scripting (XSS) [33], and clickjacking [34] are few vulnerabilities exploited by malware.

- ***Removable Media:*** Removable storage devices such as flash-drives, CDs, DVDs, USBs are considered as the old methods of spreading infections. But, these infection methods are still effective and cannot be neglected. Stuxnet [8] in 2010, infected millions of machines across the world and was delivered through a USB flash drive. These infected devices, when plugged into the systems, start autorun applications by which malware enters into our systems.

- ***Network Entity:*** High-speed communication network is a powerful weapon to spread malware attacks. Networking entities such as p2p, instant messaging, emails, shared networks, downloaders, and many more are responsible for malware infection. Internet plays a major role in spreading malicious attacks world-wide as it provides a way of connecting and communicating the clean and infected computers. Remote access and anonymity are added advantages for the malware writer when Internet is used as a medium.

2.1.2 Covert Launching

Malware authors are continuously sharpening and evolving the tools and techniques to attack our systems. In this quest, these malicious software programs are bundled with the techniques that covertly launch malware. Once the malware enters into victim machines, it tries to launch its malicious payload. But, in the presence of firewalls and security software, it could be detected. To hide its presence, malware is equipped with code to detect anti-malware solution(s) on infected device. When such malware are alerted of presence of AV, these do not execute malicious code and appear benign. Michael Sikorski *et al.* [35] presented covert techniques incorporated by malware authors for evading AV detectors. Malware embedded with these techniques are broadly classified into three categories *viz.* Launchers, Injectors, and Replacers.

1. ***Launchers:*** Launchers are executable programs developed to load the malicious payload into the system. The actual payload of these programs is completely benign in nature, and the malcode is stored inside `.resource` section of an executable. The `.resource` section is a non-executable section to store logos, strings, and images. This non-executable feature of `.resource` section makes it a choice to store the malicious code to evade the detection. On execution, these programs load the malicious payload in memory and, then, execute their clean code.

2. ***Injectors:*** These types of malware samples are very common. These programs inject malicious code into an executable binary that, on execution,

unknowingly executes the malicious payload. Injectors make use of code injection techniques. These techniques are as follows.

a) *DLL-injection*: *DLL*-injection loads a malicious DLL using `LoadLibrary` function into the context of the running process.

b) *direct-injection*: Direct injection is done by directly loading the malicious payload into the memory area of a running process.

c) *Hook-injection*: Hook-injection based malware leverages the Windows hook capability. Hooks are used for message inception. Malicious payload is released as and when that particular message is intercepted.

d) *APC-injection*. APC (Asynchronous Procedure Call) injection is implemented to direct a thread to modify its execution path. Threads of the running process queue APCs. APC is a function that executes asynchronously in the context of a particular thread. These APCs are processed when the thread is in the alertable state and invokes `WaitForSingleObjectEx, WaitForMultipleObjectsEx`, and `Sleep` function.

3. ***Replacers:*** Replacers are the malicious programs that overwrite the memory space of a benign running process with malicious payload when the process is in a suspended state. Replacers invoke `CreateProcess` to create a new process, which replaces the current suspended one. Then, these programs replace the victim process's memory with the malicious code. Hence, execution path of the running process is modified according to the malicious payload.

2.1.3 Payloads

Payload is the main component of a malware executable as it decides the functionality of the malware. Modern malware contains multiple payloads. Once the malware comes in a running state, it executes its malicious payload. The payload typifies a malware's behavior. In addition to that, this malware may also be loaded with other payloads to check that the running binary is not being monitored. If

so, the actual malicious payload is not executed. In subsequent paragraphs, these payloads are described in brief.

- ***Malicious:*** These payloads include the code responsible for data stealing, data manipulation, data removal, data corruption, unauthorized uploading/downloading, disarming firewalls/AV Scanners, logging passwords/bank transaction ID's, spying, phishing to name few.
- ***Trigger-based:*** Trigger-based payloads decide whether to deliver malicious payload based on trigger-conditions. These triggers include timestamps, system events and network inputs [36]. Many viruses and worms attack the systems on specific dates (specific weekdays or date) or a particular time of the day. Some of the malware programs with this payload wait for certain system or network commands or specific keywords to launch the malicious payload. These types of malware programs remain dormant till the advent of the trigger.
- ***Evasive Payloads:*** Present corpus of malicious binaries is detection-aware employing many evasive techniques to defeat the anti-malware solutions. These techniques are carried out with the help of payload that checks environment settings of running malware. Malware with evasive payloads may employ obfuscation, modifying system objects, *anti-debugging, anti-sandboxing, anti-vm* and *anti-emulation* techniques. In the presence of emulated or virtualized system settings, malware either stops its execution or mimic a benign behavior to avoid the detection.
  - *Obfuscation:* Polymorphic, metamorphic and packed malware programs come under this category. These types of malicious codes evade static signature-based malware detection. Variants with different syntactic structures are generated while their functional behavior remains invariant.
  - *Modifying System Objects:* There exists some malware that modify the system objects to hide their presence in the system. For instance, modifying `EPROCESS` linked list, the malware makes itself invisible in the process list of the task manager.

- *Anti-Debugging:* Malware authors apply anti-debugging to prevent reverse engineering of malware binaries. They use win32 APIs to check the presence of a debugger in the infected system. For instance, `DbgUIConnectToDbg` is used to connect with the debugger present and `IsDebuggerPresent` is a boolean function that returns a nonzero value if the calling process is being debugged by a debugger.
- *Anti-Emulation:* Emulators provide a safe environment required to capture the execution sequence of a malware binary. These emulators imitate a real system. Malware binaries incorporated with anti-emulation apply timing difference checks, CPU semantics checks, and hardware characteristics checks to evade the detection [37].
- *Anti-Sandboxing and Anti-VM:* Majority of existing approaches make use of sandboxes and virtual machine to acquire the execution sequence of malware binary. To escape from such detection, anti-sandbox and anti-VM payloads are inserted into malware binaries. The interrupt descriptor table (IDT), local descriptor table (LDT), global descriptor table (GDT) values in virtual machines differ from real machines. By checking these values, an application can determine where it is being executed.

2.2 Dynamic Malware Detection

Malware detection approaches can be broadly classified into (1) static and (2) dynamic. The static approaches to malware detection are susceptible to code obfuscation, polymorphism, and metamorphism. Moser et al. [38] have explored the limitations of static approaches in view of code obfuscation. They claimed that static analysis alone was not sufficient to identify malware binaries. The dynamic analysis is needed to complement static approaches especially for obfuscated/encrypted malware, gadgets and code injection attacks. Therefore, in recent years security researchers have focused more on dynamic detection techniques to determine if an unknown binary is benign or malware.

Dynamic approaches to malware detection utilize behavior features (dynamically extracted) to build detection model. Dynamic analysis emphasizes on gaining information about running executables and their interactions with system. Monitoring the behavior of running binary enables to collect a profile, which is less vulnerable to code obfuscation, packing, polymorphism and metamorphism. Behavior profile is a set of activities performed by a binary program during execution. These activities denote the action or reaction of malware under certain internal/external input or environment states. To capture these activities, a standard sequence of operations as shown in Figure 2.2 are carried out. The entire process is completed in three steps: 1) Capture program behavior 2) Model captured behavior and 3) Design a detection and categorization model.

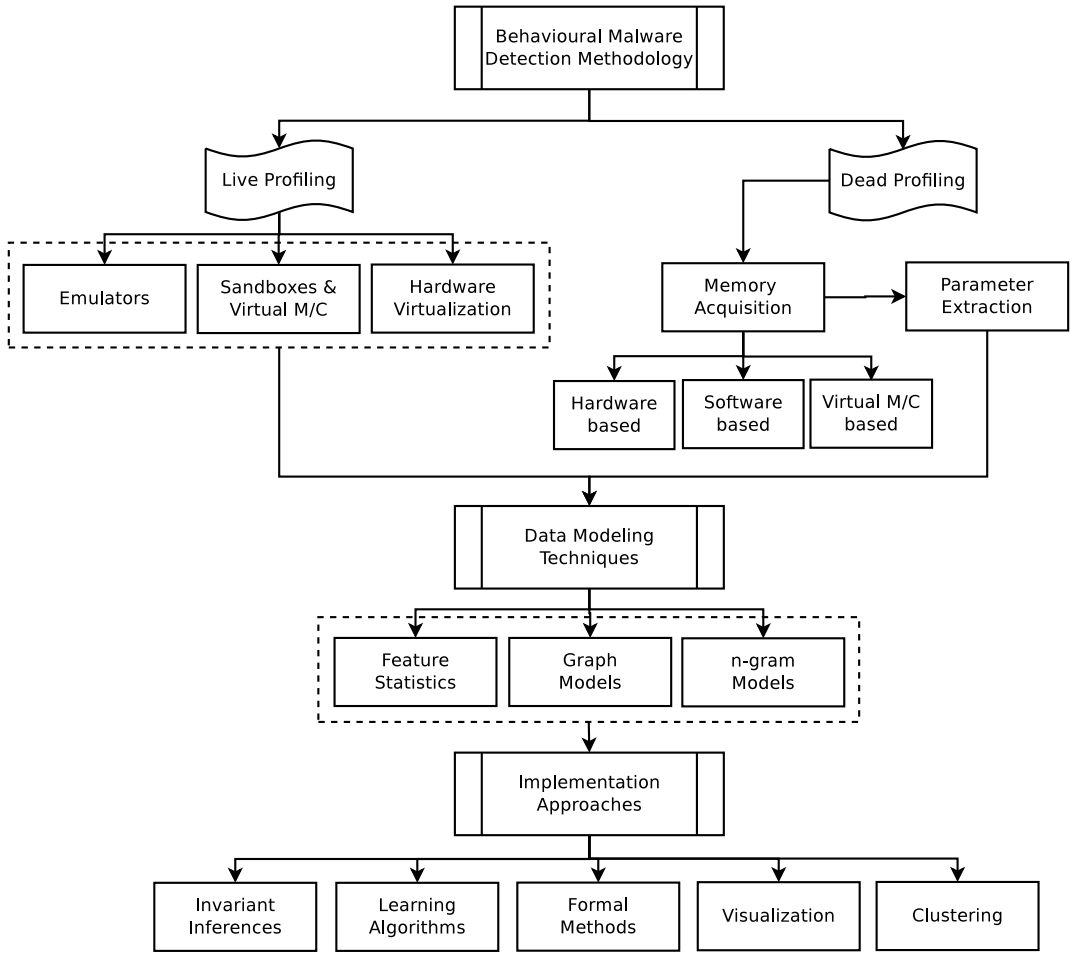


FIGURE 2.2: Dynamic malware detection steps.

2.3 Behavioral Profiling: Analysis Frameworks

Dynamic malware detection approaches rely on the execution of malware binaries. As malware execution on physical machines/host can corrupt them, a safe analysis environment supporting malware execution is the first step in analyzing malware behavior. It provides isolation between host and guest operating systems. Any side effects generated by the malware execution cannot harm the host machine, and the modified and/or infected guests can be reverted to their original clean state. Analysis environment should be immune to *anti-debugging*, *anti-emulation* and *anti-virtualization* techniques employed by malware binaries. Behavior profiling can be incorporated by live and dead analysis of running executable. Former approach captures behavior during the malware execution while the later approach investigates the raw memory dumps to examine the malware footprints in memory. Tools like Vtrace, Procmon [39] and Responder-Pro [40] capture specific activities (temporal and frequency attributes) of a process within a guest environment.

2.3.1 Live Behavior Profiling

Live profiling enables analysis of the malware behavior during run-time. It gives an overview of running program by capturing the activities that transpired at runtime of binaries. To capture these live profiles, following in-box and out-of-the-box mechanisms are used.

1. ***Emulators:*** Emulators such as ANUBIS (**An**alyzing **U**nknown **Bi**narieS) [41] and Panorama [42] provide the full system emulation for capturing malware behavior. The emulated environment in ANUBIS and Panorama is controlled by Qemu [43] that runs Windows XP as the guest. These emulators log process, file, registry activities, loaded DLLs, API calls, and system calls. The Emulator-based monitoring remains unaffected by the anti-debugging and obfuscation techniques employed by malware authors to defeat the detection. But, these systems are vulnerable to anti-emulation

techniques [44] because these systems cannot correctly emulate physical real CISC computers [45] in toto.

2. ***Sandboxes:*** Sandboxing techniques are also called as in-box-monitoring. CWSandbox (Now, GFISandbox), Cuckoo and NormanSandbox are few techniques to monitor an executable. CwSandbox and NormanSandbox collect process-specific activities by API hooking. Hooking mechanism monitors each function prior to its execution and collects activities related to file system, registry modification, system calls and network traffic [26]. In addition, Cuckoo includes the memory dumps of the windows for verifying the malware presence. These approaches share the same privilege level as the monitored application, therefore, the malware acting on kernel-level can interfere and manipulate the logs of sandboxes.

3. ***Virtual Machines:*** Virtual machines such as VMware and VirtualBox provide a guest OS environment that is a look-alike to real OS. This environment is used for tracing the logs of running malware. There are tools such as procmon, strace, tshark, PIN and Windbg used to gather dynamic traces in virtual machine environments. These tools are installed in guest OS and collect the behavior patterns.

4. ***Hardware Virtualizers:*** Ether [46] and V2E [45] provide a transparent analysis platform by incorporating hardware virtualization. Ether makes use of XEN hypervisor to provide the same. It logs instructions, system-call traces and unpacked dumps of the executables by intercepting EFLAG register. It only provides single-step instruction tracing because in-depth tracing will slowdown its performance. V2E, on the other hand also include the software emulation for extensive binary analysis. It relies on TEMU for emulation, therefore, its replay mechanism lacks in efficiency. Figure 2.3 shows the tracing architecture of Ether and V2E.

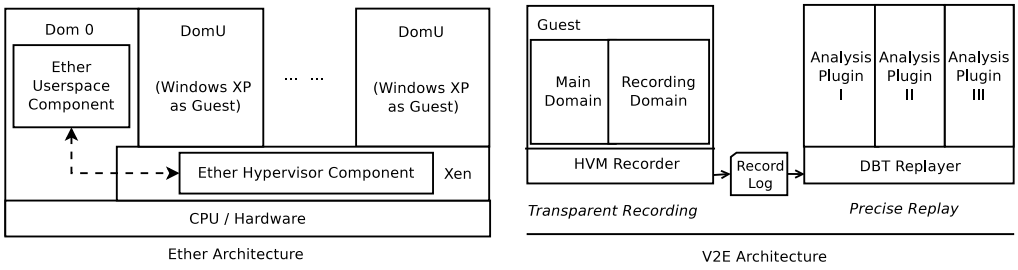


FIGURE 2.3: Live-tracing: hardware virtualization [45, 46]

2.3.2 Dead Behavior Profiling: Memory Forensics

The role of memory forensics in behavioral malware analysis has become vital due to the presence of detection-aware malware being executed in memory. These malware programs can be captured by acquiring and analyzing digital artifacts from memory. These digital artifacts include network activities, file activities, registry activities, currently running process, loaded kernel modules, and other critical information. The memory forensics can be implemented in two steps (a) *Acquisition of memory* and (b) *analysis of acquired memory dumps for incidence responses.*

2.3.2.1 Memory Acquisition

Modern malware has reached to a high level of sophistication. It utilizes system resources and stores its data like decryption/deobfuscation keys, botnet command and control information and routing tables in memory [47]. Volatile data can be captured by following techniques as reported in literature [48, 49].

- *Hardware-based:* Hardware-based acquisition techniques of volatile data are proposed to acquire a reliable memory image of the infected system. Carrier *et al.* [48] have described that rootkits and trojan horse attacks against applications and operating system kernels can modify the system data, therefore, the applications running on such machine for creating memory dumps cannot be trusted. WindowsSCOPE [50] and Tribble [48] make use of PCI card and PCI express bus respectively to store volatile information. The overhead of installing PCI cards with hardware-based approaches put a question mark on such solutions.

TABLE 2.2: Behavior profiling techniques

Profiling technique	**Category**	**Characteristics and Limitations**	**tools**
Emulators	Live	Remain unaffected by the anti-debugging and obfuscation techniques. Vulnerable to anti-emulation techniques by timing difference.	ANUBIS, TEMU Panaroma, Qemu
Sandboxing & VMs	Live	Hide all system objects that could reveal the presence of the analysis framework. Potential malware residing on higher privileged level, interferes with sandboxing and virtual framework	CWSandbox, Cuckoo, NormanSandbox, VirtualBox, VMWare
H/W Virtualizers	Live	Execution is monitored from bare metal hardware. In-guest changes are made hidden and intercepted to provide transparency. Performance degradation due to heavy instruction-level tracing.	Ether, V2E
H/W-based	Dead	Makes use of PCI card and PCI express bus. PCI card and express bus must be installed prior to its use.	Tribble, WindowsScope
Software-based	Dead	Does not require any dedicated hardware to acquire the memory image. Requires process and kernel memory to perform acquisition and will overwrite possible evidences of running malware	PMDUMP, PD, NotMyFault.exe
Virtual-machine based	Dead	Provides a facility to snapshot the memory image. Contains own set of virtual processor, memory, graphics cards, input/output interfaces. Prone to anti-virtualization as for instance VMWare uses `vmware` string in its internal processing.	VMWare, VirtualBox
Memory Analysis	Dead	Extracts digital artifact from raw dump of memory. Unable to log the modified system objects, which are put back to their initial state during execution.	Volatility, Memparser, Memoryze

- *Software-based:* The current approaches for software-based memory acquisition rely on Windows debugging services and third party applications to access physical memory. Microsoft has incorporated various debugging options that are used to get memory image into its versions of Windows in the form of crash dumps. To generate these dumps of user memory and kernel memory, `CrashOnCtrlScroll` and `NotMyFault.exe` developed by sysinternals [51] can be used. The former approach is having limitation of usable only with PS/2 keyboards. The tools such as PMDUMP and Process Dumper (PD) are capable of dumping a particular process's state, code, data and its environment

stack. These tools make use of process status application programming interface (PSAPI) to get the information about running processes which can affect the acquired image. The software-based solutions require process and kernel memory to perform acquisition and will overwrite possible evidences of running malware [48].

- *Virtual Machines-based:* The virtual machine environments, such as VMWare and VirtualBox offer a facility to snapshot their current state. These virtual machines are equipped with respective set of virtual processor, memory, graphics cards, input/output interfaces [49]. The memory dump is saved in file with formats such as `.vmem, .img, .vdi, .sav` based on virtual machine used. To capture the execution traces of malware, these environments are used and memory dump is achieved for analysis. Virtual machine aware malware [52] can alter its behavior by knowing the fact that it is being monitored. In such specific cases, the virtual machine based memory acquisition is not a good idea to investigate malware behavior.

2.3.2.2 Analysis of Memory Dumps

The acquired memory dumps are sent for in-depth memory analysis for malware footprints. Analyzing malware requires tools such as volatility, memparser, and memoryze. These tools help us in extracting digital artifacts from memory dump of a particular sample that needs to be analyzed. A raw memory dump can contain information like hooked processes, kernel modules, system calls, data structures, network traffic, loaded DLLs, files, APIs and registry hives.

2.4 Data Modeling Techniques

Abstraction of behavior features also play a vital role as appropriate data modeling determines what kind of approach can be applied [53]. Acquired traces must be transformed into certain forms to detect the malicious behavior as malware authors try to add irrelevant and independent features to avoid the detection. These data

modeling techniques include Feature statistics-based, graph-based, and n-gram based.

2.4.1 Feature Statistics-based

The approaches [54–58] that rely on some statistics of their extracted feature, are put into this category. These statistics include the count, probability, data–value, entropy, and information gain. These features with their statistics are further refined to select the prominent feature.

2.4.2 Graph-based

Graph-based modeling is deployed to encode the relative information of a behavior parameter. To detect malware, graph or subgraph with aggregated feature attributes is formed. This modelling represents the dependency structure of the sequences. This dependency model is constructed using program modules, control-flow and program instructions. Shun *et al.* [59] have developed a dependency structure matrix (DSM) abstracting the task/module dependencies to detect the module–based co-working malware. Babic *et al.* [60] and Fredrikson *et al.* [61] have modelled their system call traces into a data-flow dependency graph. Graph-based modelling suffers from the limitations that these methods are computationally expensive and also graph matching is ambiguous due to graph isomorphism.

2.4.3 n-Gram based

n-gram based models have been widely used for static and dynamic malware detection [62–64]. Christian *et al.* [65] have presented the suitability criteria (perbutation, density, variability) for n-gram models in malware detection. n-grams is the overlapping subsequence of length n. In n-gram technique feature space depends on two parameters n and L [66]. Here, L is the total number of features (instruction, system calls, APIs, etc.) collected from execution traces. The larger the value of n and L, higher is the computational complexity as larger sequences need to be processed. Accuracy of the model may reduce as each n-gram may have

redundant information shared with other features. Model may be too generalize to be of any use. On the other hand, small values of n and L results into false alarms. When n is small, each feature may carry information too small to be of much use.

2.5 Approaches

After applying the data modeling on acquired traces, various implementation techniques as shown in Figure 2.2 can be applied. We have segregated these techniques into following categories and discuss these along with their pros and cons.

2.5.1 Invariant Inferences

A predefined set of invariants is recorded from the execution of normal programs. Any modification in the invariance during the program execution raises a flag of suspiciousness. This technique is employed to detect the normal and abnormal behavior of a binary under consideration. Invariant inferences in malware detections are mainly used to verify the presence of a highly-privileged malicious sample. The system objects are precisely selected for inferring the invariance. Here, we are presenting those techniques that make use of kernel data structures, kernel modules, kernel code and data and kernel memory to detect the presence of malware in the system.

Shosha *et al.* [67] have developed a Signature-Generation tool (SigGene) for malware detection. The signature was formed by profiling kernel data structure (`EPROCESS`) objects extracted from malware execution traces. `EPROCESS` is a kernel data structure used to represent the running processes in the operating system. This data structure is manipulated by malware to evade detection (Figure 2.4). `Flink` and `Blink` in this figure denote the forward and backward links of `EPROCESS` list. Malicious code is identified by invariant values of features in execution traces.

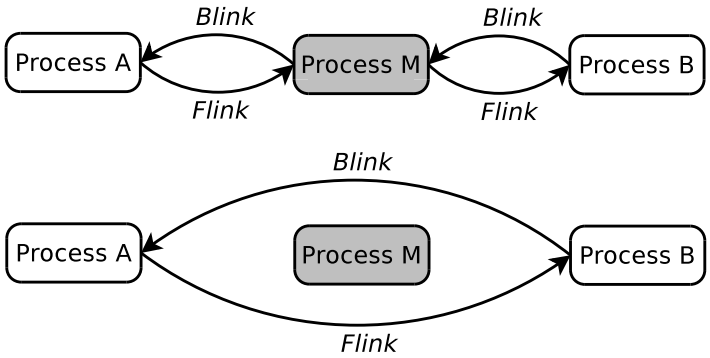


FIGURE 2.4: DKOM attack: manipulating *EPROCESS* kernel data structure

A quite similar approach for rootkit detection was introduced by Dolan–Gavitt *et al.* [54], but the signature formation was different than [67]. The features, those cannot be altered by malware, were used for signature construction. Any alteration in those features will either cause a system crash or an inoperable system state. In training phase signatures were constructed in two steps – profiling and fuzzing – using `EPROCESS` structure. The authors have claimed that if OS is functioning properly for 30 seconds then the application is a legitimate one otherwise it is malicious.

Riley *et al.* in [57], have developed a system called NICKLE which is a rootkit prevention tool. NICKLE guarantees lifetime kernel code integrity. They have implemented Virtual Machine Monitoring (VMM) to preserve a memory area (named as shadow memory) of a guest OS. NICKLE works on the assumption that the application in guest OS cannot alter the highly privileged shadow memory. Hence, any effort towards illegitimate code execution can be hampered.

Rhee *et al.* [68] proposed an approach that incorporate kernel data-object mapping. The authors have used data allocation and deallocation mechanism to find out the illegitimate code. They have constructed malware signature based on their data access patterns. They have considered Linux platform for their approach. But they have also claimed that their approach is also effective in detecting a Windows-specific malware.

Xuan *et al.* [69] have introduced Rkprofiler that is sandbox-based malware tracking system that monitors the malware behavior. They have employed a memory tagging technique named as Aggressive Memory Tagging (AMT). This

tagging technique tracks the kernel objects visited by a malware. The Rkprofiler is capable of tracing instruction and function calls made by malware. They have shown that their method is efficient in capturing rootkit behavior.

- ***Invariant inferences: pros and cons***

  1. *Merits:*
     - Provides a way to find out the vulnerable or modifiable system level objects.
     - A promising mechanism to detect the presence of high-privileged malware into our systems.
  2. *Limitations:*
     - These techniques require dedicated hardware and software systems with in-depth knowledge of each OS objects.
     - A single kernel object can only target a fraction of malware domains. And, handling all the objects is a computationally expensive.

2.5.2 Visualization

Visualization technique is gaining prominence in identifying malicious attacks. It provides a way to find out the regions of malicious functionality, therefore, can be used as a base for feature selection in malware detection. It presents a view of a suspicious activity in the form of information flow, network flow, or similar instances. These approaches completely rely on the tools used for analyzing anomalies. There are many tools for visual representation of malware behavior. These include Treemaps, Graphviz, NFlowViz, Skyrails, Dotplots, Grid layouts etc.

Trinius *et al.* [58] have proposed an approach based on treemaps and threaded graphs to visualize the individual functional domain and temporal behavior of malware executables and PDFs respectively. This approach mainly aims at detecting malware and classifying it according to its behavior. Using treemaps,

authors try to visualize activities such as network related, file system changes and process interaction of a sample according to API call sequences. In addition to that, they have also made use of threaded graphs. Threaded graphs represent the temporal order of these activities. Malware behavior is summarized in a colored map where each color represents a different activity.

Mansmann *et al.* [70] presented a system for network flow visualization (NFlowVis) and compared it with treemaps and graph representation for intrusion detection in network traffic. The NFlowVis provides the IDS alerts, home-centric view, graphical view and network flow view. Network monitoring gives an overview of the vulnerable and unwanted host in the network by visualizing the connectivity and communication patterns between the local and external hosts. The authors have visualized three case studies using NFlowVis *i.e.* service monitoring, SSH attack distribution, and blacklisted hosts investigation.

Yongzheng *et al.* [71] have presented an approach to find out the similar malware instances using DotPlots. They make use of Hash-based content sampling to reduce the n-gram based feature space. They have used the byte opcodes gathered from memory images of the infected and clean system. Authors have pointed out that their visualization approach is affected by Address Space Layout Randomization (ASLR) strategy of Windows to relocate the independent codes.

Saxe *et al.* [72] try to visualize the various behavioral characteristics of malware samples. They have presented a clustering based method that relies on the functional blocks of system call sequences to measure the similarity. Quist *et al.* [73] have applied visualization to capture the overall flow of a binary to investigate the source code areas of malware samples. The approach can only be applied to select the feature domain of a malicious binary.

- ***Visualization: pros and cons***

  1. *Merits:*
     - Identification of the source of malicious attacks in network traffic.
     - Provides a better understanding of malicious code layout and similar malware instances.

2. *Limitations:*
   - Requires better graphics cards, visualization tools and a system with high computing capability.
   - Requires expert human intervention to visualize behavior patterns.

2.5.3 Machine Learning Techniques

The machine learning techniques for malware detection and classification aim at developing discriminatory model from patterns formed by features through supervised/unsupervised methods. In supervised learning, detection model is trained using known behavior instances while in unsupervised learning, the detection model is prepared without the prior knowledge of binary. Here, we will discuss both the forms of learning techniques used in malware detection.

Tian *et al.* [74] and Islam *et al.* [55] have presented an approach of malware detection and classification using API calls. In the former approach, the authors have applied five classifiers (SVM, RF, adaboost, IB, DT) in WEKA to achieve their aim. They achieved 97.3% and 97.4% of detection accuracy in malware detection and categorization. In the latter approach, authors combined API calls with two static features (function length and printable strings). With this integrated feature set, they used four classifiers – SVM, DT, RF and IB1 – to classify a sample. They claimed 97.3% of detection accuracy.

Moskovitch *et al.* [21] have presented a technique to detect known and unknown worms in the different environment. The authors have collected eight different datasets containing system activity in the execution duration (20 minutes) of the worm and normal application. These systems have diverse hardware and software configurations. In addition to this, impact of background activity and user activities was also monitored. They have constructed a feature set of 323 from categories such as protocols, process, threads, network interface and memory. This feature vector size is reduced to 20 after employing three feature selection methods (Chi-Square, Gain Ratio and Relief-F). The authors have applied DT, NB, BN and ANN with reduced feature vector and shown that their method is effective enough to detect known and unknown worms with low false positive rate.

Ahmadi *et al.* [75] have discussed an approach using API calls to detect a malicious sample. They have also applied two feature selection methods (Fisher score and CFSSUBSETEVAL) to remove the redundant and irrelevance features. The authors have applied SVM and RF to classify the labeled data.

Another SVM-based learning approach was employed by Rieck *et al.* [76] and Anderson *et al.* [66, 77]. The authors in former approach [76], have used API calls to built a weighted behavior learning model. The weights are assigned to behavior patterns according to their contributions to malware families. Benign samples are also used to build a strong discriminative model. In the latter approach [66], authors combined static and dynamic features that are representative of the program's intent to classify malware families. They have incorporated multiple kernel learning to construct a weighted combination of combined feature set (opcodes, basic blocks and system calls). The authors have claimed that formed feature vector resulted into detection accuracy of 98%.

- ***Supervised Learning Algorithms: pros and cons***

  1. *Merits:*
     - These techniques provide a simple approach of classifying labeled malware instances irrespective of any infrastructure and architecture.
     - A correctly learned discrimination model results into a high detection accuracy as compared to any other technique.
  2. *Limitations:*
     - Approach will fail if malware authors manipulate applied features.
     - As, time passes the false positive rate tends to increase due to zero-day malware evolution.

Ulrich *et al.* [78] have proposed a method to characterize a malware into its specific class making use of emails transferred, HTTP downloads and IRC conversations. Execution traces (system calls + APIs) are collected and transformed into behavior profiles. The authors have applied hierarchical clustering using system calls and APIs. They have applied a local hashing mapping (LSH) to reduce the clustering

overhead. For similarity measure, Jaccard index has been employed. Finally, precision and recall are used to evaluate the proposed method.

Perdisci *et al.* [79] have presented a malware classification approach by investigating similarities in HTTP URLs requested by malware. Coarse-grained (BIRCH), fine-grained (Hierarchical) and cluster merging have been applied to form clusters. The first two clustering are employed for the refinement of clusters being formed while the cluster merging is used to generate network signatures. To form the clusters, authors have used a total number of HTTP requests, a total number of GET-POST requests and average URL length and response length. Levenshtein and Jaccard indices are used in forming clusters.

Kheir [56] has proposed a malware detection approach by investigating abnormality in HTTP user agents. The execution traces are generated using `tshark` in sandboxed environment. The authors have employed three steps of clustering (High-level, fine-grained, incremental k-means) to generate the signature. They have also evaluated their cluster forming technique and signature quality. As these approaches [56, 79] are signature-based, therefore, fail to detect unseen malware samples.

Fredrikson *et al.* [61] and Park *et al.* [80] have presented a graph clustering approach. In the former approach, authors have developed a tool named as HOLMES. It works in two steps: 1) Extraction of significant malicious behaviors and 2) Creation of a discriminative specification of malware behavior. A dependency graph is constructed in which the graph vertices (system calls) are connected by the dependency in their arguments. The authors have also applied simulated annealing to find the cluster subsets. A behavior is specified by applying information gain, Structural Leap Mining (SLM). Further, one common synthesized malware behavior is extracted for a malware dataset. The authors have shown the detection rate of 86% with 0% false positives. The latter approach constructed a HOT path that represents an overall malware family behavior. The authors have constructed a Kernel Object Behavior Graph (KOBG) from system call traces. These KOBGs are the set of kernel objects (vertices) and their dependencies (edges). They have generated a Weighted Common Behavioral Graph

to compute the common behavior of a family. Finally, they have also shown that their approach is vulnerable to system-call injection attack.

Jacob *et al.* [81] have proposed a graph-based technique that specifically identifies bot-initiated Command & Control (C&C) communication. They have constructed a behavior graph of system call traces. C&C templates are created with known and unknown C&C communications. These templates share a similarity in behavior graph. The authors have applied subgraph matching to find out the subgraph that is not in the benign dataset. Then, they formed clusters of subgraphs having homogeneous C&C activities.

- ***Unsupervised Learning: pros and cons***

  1. *Merits:*
     - Clustering provides a way of labeling unlabeled data by grouping them according to their similar/dissimilar behaviors.
     - Every attribute of the dataset participate in the clustering process, therefore, intra-cluster and inter-cluster homogeneity and heterogeneity remain consistent.
  2. *Limitations:*
     - Unsupervised nature of clustering does not allow any external valuable information that can help in forming clusters.
     - The number of clusters and the size of clusters is undecidable.

2.5.4 Formal Methods

Formal methods provide a mechanism of modeling real activities in mathematical proofs and models. These models must be validated and include a specification for achieving a high degree of formalism. Formal methods of malware behavior detection offer a way to express the program semantics. These models provide a formal semantics-based framework that can be proved beneficial in evaluating maliciousness of a target program. These techniques make use of model-checking (LTL, CTL), finite-state-automata and push-down automata.

Babic *et al.* [60] have proposed a malware detection and classification approach based on tree-automata inferences. Tree-automata represents a state-machine in which tree structures are used. The authors have extracted system call traces using PIN (A dynamic binary instrumentation tool) and constructed data flow dependency graphs. They have implemented their approach in two steps: 1) Inference algorithm construction, 2) Detection and classification mechanism using inferred algorithm. In step-1, k-Testable in Strict Sense (k-TSS) language is defined for constructing inference algorithm. It consists of k-level tree patterns. These patterns and k root hash divide the graph into a finite number of equivalence classes. In step-2, learning phase offers the fine tuning of inferred automata with benign and malware samples. Each sample is executed against the inferred automata and tree height is measured to calculate a score $S \quad S \in [0, 1]$. Higher score value indicates a higher likelihood of the sample to be malicious. For classification purpose, authors developed family specific tree automata.

Tree-automata provide the abstraction of the malicious patterns by inducing a minimal state automata. The inferred tree automata is refined every time thus it increases the inference complexity ($O(kpn)$). Here, k is the number of levels, p is the number of the patterns and n is the size of input supplied.

Kinder *et al.* [82] developed a Computational Tree Predicate Logic (CTPL) to identify malware behavior. They used instruction opcodes which were extracted statically. They have shown that their behavior specification can be used to model the real-world worms.

Song and Touili in their works [83–85] have proposed a a method of modeling malicious behavior using pushdown systems (PDS) that track the program stacks. They have extended the CTPL to SCTPL for representing stack operations [83]. Further, authors expanded their work and produced SCTPL formulas that take into account the values of registry and memory locations instead of their names [84]. In [85], they developed Linear Temporal Logic (LTPL) with predicates. These LTPLs were then applied with PDS for SLTPL (for stack semantics). Similar work was proposed by Beaucamps *et al.* [86]. They have incorporated First-Order Linear Temporal Logic (FOLTL) to transform static malware traces

into high-level behaviors. Their developed logic is used to identify the malicious and benign samples.

Beaucamps *et al.* [87] have used Finite State Automata (FSA) to identify the anomalous malware behavior. They have created a notion of traces and behaviors using abstract machine modeling. In order to achieve their goal, they have used library call extracted from PIN. These traces are mapped with their designed FSA model. Their approach was independent of implementation details. The authors have identified the programs with keylogging or similar behavior.

- ***Formal methods: pros and cons***

  1. *Merits:*
     - Formal methods provide a full coverage of activities incorporated to compute program behavior.
     - These have the potential of designing a behavior detection model that is resilient to malware side effects.
  2. *Limitations:*
     - These methods tend to have high learning rate due to vast diversity of malicious files.
     - To build, a precise model, ample amount of human efforts are needed.

2.6 Summary

In this chapter, we have reviewed existing dynamic malware detection techniques. We have discussed infection and attack vectors of contemporary and old malware. Additionally, we have presented behavior monitoring and data modelling methods applied prior to malware detection. Based on our literature survey, we have observed that present malicious threats are equipped with anti-detection features. The existing solutions of malware detection are vulnerable to these anti-detection features. Due to anti-detection behaviors, modern malware exhibit multiple behaviors at run-time. In next chapter, we present that malware samples belonging

to same family depict different behaviors. Further, we address two anti-detection measures – 1) environment-reactive and 2) system-call injection – employed by modern malware in subsequent chapters. Considering these measures, we will propose malware detection with comparative detection accuracy.

Chapter 3

Detecting Malicious Behavior using Dynamic Time Warping

In past few years, the growth in malicious codes has been increased exponentially. Also, these codes are bundled with anti-detection features that enhance their severity of infecting our network and systems. Analyzing a malware sample for identifying its nature of infection, is the first step. But, analysts are facing millions of samples everyday and focusing on an individual sample is not practically possible. Therefore, there is a need to cluster malware samples that exhibit similar run-time behavior.

In this chapter, we present an approach that clusters malware samples within a malware family. Our notion of clustering is based on the fact that inspite of belonging to same malware family, the samples constitute different behaviors. These behaviors indicate that samples are embedded with multiple payloads for evading detection mechanisms. These payloads are delivered according to the security measures and execution environment present in the system. Therefore, samples belonging to a malware family, show different behavior at run-time. We, in this approach, present a malware behavior clustering and detection approach.

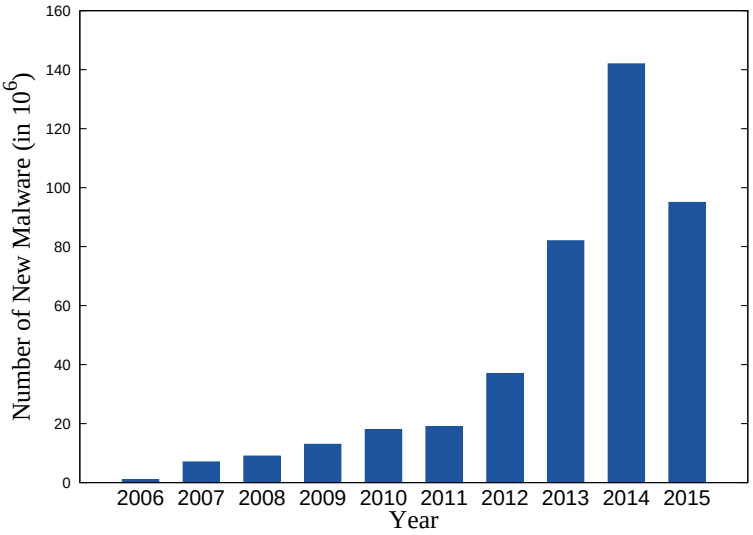


FIGURE 3.1: AV-Test report [19]

3.1 Introduction

Malware programs have a disruptive impact on our applications, service providers, storage, servers, and networks. In 2013, AV-test Institute discovered a total of ∼100 million new malicious files and this number has reached ∼140 million in 2014 [19]. This explosion of completely new malware threats and variants of existing malicious threats cause substantial damage in terms of financial losses. Figure 3.1 illustrates how number new malware samples received at AV-Test are increasing. In this figure, we can see that in first half of year 2015 this figure has reached to ∼95 million. By the end of 2015, this figure will be ∼200 millions.

The explosion of thousands of new malware samples everyday has increased the workload of Anti-malware (AM) researchers. Each of these samples requires security experts to analyze their threat and severity levels in-depth. Therefore, there is a need to categorize each incoming sample into its respective behavior to accelerate this process. Clustering malware samples having similar behavior traits is beneficial for following two reasons [78]:

1. Every time a new sample arrives, analysts can determine whether it is a completely new instance or a variant of existing malware class.

2. It becomes easy to derive generalized removal procedures and to create new mitigation strategies that work for the whole class.

Clustering malware samples is not new. In past, many approaches [56, 61, 78, 81, 88–90] have been developed to cluster and classify malware samples. Existing approaches of clustering malware can be divided into two broad categories- 1) network-level, and 2) system-level.

The network-level [78, 89, 90] clustering approaches utilize network traffic. The network-traces are acquired by scanning network during malware execution. These approaches distinguish between the normal and abnormal traffic in terms of HTTP downloads, application-level protocols (SMTP, FTP, IRC), web-based command & control, HTTP-URL etc. However, modelling normal and abnormal network traffic is hard because of the diversity in the nature of the traffic from email systems, instant messenger and peer-to-peer applications resulting in high false alarm rate [91].

The host-level approaches [88, 92] cluster malware samples on the basis of their payloads responsible for carrying out malicious attacks. Capturing malware samples on the basis of their actual attacking behavior allowed exploring the unknown and new malware samples irrespective of their propagation mechanisms [93]. Therefore, in this chapter we focus on host-based identification and categorization of real malware samples.

For behavior-clustering, similarity measures play an important role. The majority of existing approaches [78, 89, 90, 92] make use of Jaccard Index [94]. The Jaccard measure is a set-intersection based method. This type of similarity measure suffers from following limitations.

1. It cannot capture the ordered relationship within the execution traces.
2. It cannot capture the similarity when data is sparse [95].
3. It fails to capture the lack of similarity among activities that differ between two samples [95]. For instance, if two execution traces show high registry activities, but only one of the trace contains some file activities, Jaccard measure gives high similarity score, thus not able to capture the lack of similarity between file activities.

To overcome above-mentioned limitations, we use Dynamic Time Warping (DTW) as our similarity measure. DTW can capture the intra-relationship and dissimilarity among execution traces. Also, it captures the similarity when sequences are sparse. DTW is a dynamic programming algorithm that is widely used to align two time-varying or variable length sequences. Our malware sequences are of variable lengths and also vary with time during execution. Therefore, in this work, we utilize DTW to cluster malware samples and differentiate malware from benign programs.

The main objective of our approach is to identify multiple behaviors within a malware family. Initially, we have selected worm malware family to justify our heuristic of "multiple behaviors within a family". The selection of worms is made as these samples show the highest threat to our computer systems (discussed later). The proposed approach exploits non-uniformity in execution traces of worms. We perform a cluster validity assessment to quantify exact number of clusters. In addition to that, we design and implement the parallel algorithm of DTW (P-DTW) to reduce the computational complexity of DTW. Our proposed approach is evaluated with known and unknown instances of virus family samples also.

3.2 Worm Behavior

The Worm is an illegitimate program that exploits the vulnerability of connected systems and applications within a network for spreading itself injecting systems encountered en-route. Besides infection through network, worms have the capability to carry out attacks such as DDoS, privacy-breaching, phishing and data-loss [93, 96]. In past few years, worm attacks have caused substantial financial losses. One single worm can infect thousands of machines connected in a network. For instances, "Code Red", "Stuxnet"and "Slammer" are a few known worms that induced significant damages costing billions of US dollars [97].

According to KasperSky Labs [98], the threat level of worm samples is the highest among all the other family samples. Figure 3.2 shows that threat level of each

malware family in a descending order. Therefore, we also, have considered worm samples to evaluate our DTW-based clustering method.

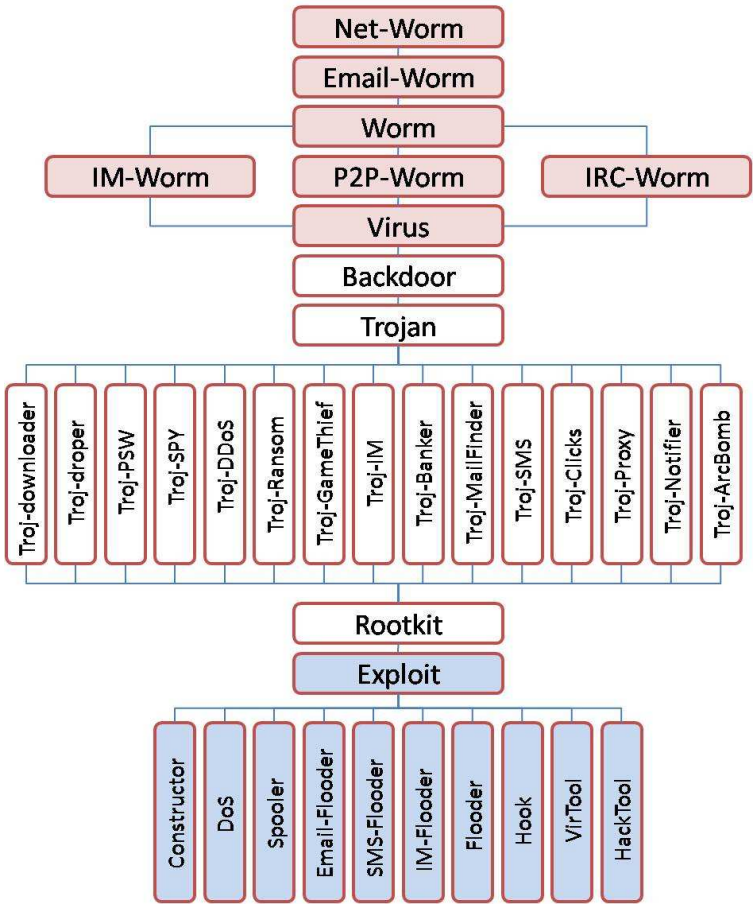


FIGURE 3.2: Malware family tree showing threat level (in descending order) [98]

3.3 Approach Overview

The prime purpose of our model is to construct groups that are functionally similar within worm family so that further analysis of worms becomes a convenient task for security analysts. In conjunction to that we also aim to discriminate worm and benign executables. To carry out these objectives, we divide our worm and benign datasets into Dataset1 (70%) and Dataset2 (30%) sets. Figure 3.3 outlines our proposed detection and categorization model. This figure illustrates the proposed approach. System calls are used as a feature for behavioral modelling. DTW is used for assessing similarity measure. Blue line indicates processing steps for

training and red path shows how samples are tested. Each sample of both the sets are traversed through three main phases discussed in following paragraphs.

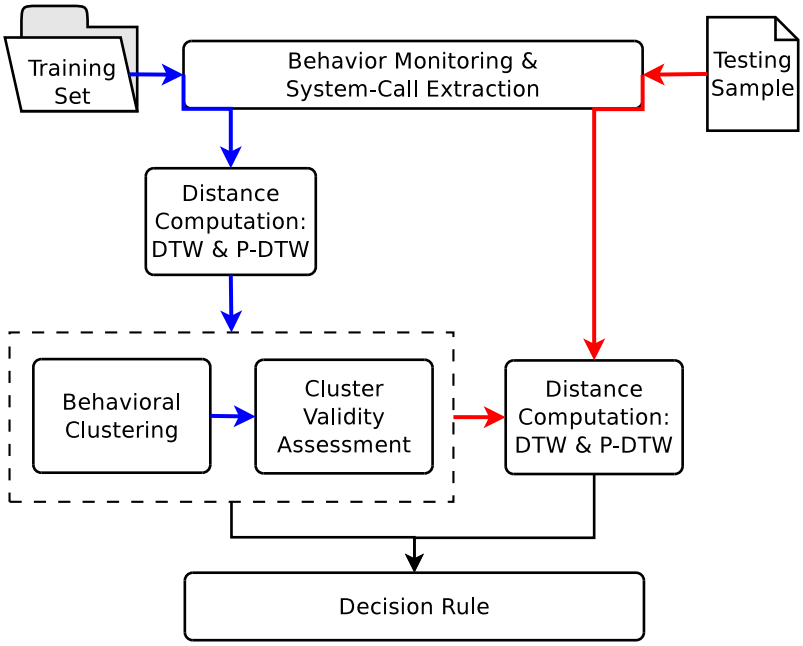


FIGURE 3.3: Proposed approach

3.3.1 Behavior Monitoring

Behavior monitoring is the first step in any dynamic malware detection problem. We utilized Ether [46] for behavior monitoring of executables. According to [46, 66], Ether provides a safe and transparent environmental setup that does not relay any side-effects to host systems and also not visible to running binaries respectively. We select Ether over other tools (CWSandbox, ANUBIS, NormanSandbox, etc.) because it is more resilient to anti-analysis techniques (anti-debugging, anti-virtualization, anti-emulation) deployed in worms as compared to aforementioned tools. Although, worms equipped with timing and CPU semantics attacks can detect the presence of Ether [99]. To hide their malicious payload, these worms depict unpredictable behavior that includes crashing the guest machine, executing beyond the time-out limit and not generating any log. Our proposed approach can also capture this category of worms as these samples will have diversity in their execution traces. Due to incurred diversity, existing techniques may not work properly.

We use system-call traces for detecting worms and grouping them according to their behavior. System-calls are invoked by a running program to request kernel for accessing system services. A worm (or any malicious program) tries to infect the host machine by altering the system state [100, 101]. For this purpose, it makes use of system-calls as these are the non-bypassable interface for modifying OS state. Therefore, to capture a program behavior or functionality, system-calls are the valid choice.

In our method, we consider Windows XP (SP2) as our guest operating system to monitor the execution traces of binaries that are in PE (Portable Executable) file format. According to [102], there are 284 unique system-calls that can be invoked by any running application in Windows XP. To collect the system-call traces, we limit execution time for each sample to 10 minutes. Anderson *et al.* [66] have mentioned that 5 minutes of time-out is sufficient for malware binaries. We doubled this execution-time to capture the worms encapsulated with time-out attacks. During behavior monitoring, we observe that worm samples exhibit non-uniformity in their running time that results into the variable length of traces. After collecting the logs for each sample, we extract the list of system-calls invoked. This list will be used as features for subsequent phases.

3.3.2 Distance Computation using DTW

In this approach, we exploit non-uniformity in execution time and trace lengths of worms to identify their behavior. We compute distance score between the pairs of samples using DTW as it can efficiently align the sequences that constitute variability in their length and execution-time. The DTW [103] is a dynamic programming algorithm that yields an optimal alignment path of two sequences which is further used to determine distance score.

For example, consider source sequence $\mathbb{S} = \{S_1, S_2, \cdots, S_N\}$ of length N and target sequence $\mathbb{T} = \{T_1, T_2, \cdots, T_M\}$ of length M for score computation. In following discussion, each S_n and T_m denote the n^{th} and m^{th} system-calls invoked by source (worm) and target (worm or benign) binaries during execution. We map each call in both the sequences with an ASCII character as we observed that

only 160 calls out of 284 were invoked during behavior monitoring. This mapping minimizes the time and space complexity of alignment process in DTW. Score computation in DTW is carried out using following two steps.

Step** 1**: Score Matrix Construction

To find the optimal alignment between $\mathbb{S}$ and $\mathbb{T}$, we build an $N \times M$ matrix ζ. The matrix ζ is *Accumulated Cost Matrix* constructed during the alignment process. The value $\zeta[n, m]$ is determined using Equation 3.1 where n ranges from 1 to N and m ranges from 1 to M.

$$\zeta[n, m] = \phi(S_n, T_m) + min \left\{ \begin{array}{c} \zeta[n-1, m-1] \\ \zeta[n-1, m] \\ \zeta[n, m-1] \end{array} \right\} \tag{3.1}$$

In Equation 3.1, the first addend called as local cost measure $\phi(S_n, T_m)$ indicates the match/mismatch value for S_n and T_m. This value will be either 0 or 1 depending upon match or mismatch of S_n and T_m. The second addend is the minimum value of diagonal ($\langle n-1, m-1 \rangle$), top ($\langle n-1, m \rangle$), and left($\langle n, m-1 \rangle$) cells. The value of cells (first row and first column) in matrix ζ is decided with following initial conditions (Equation 3.2):

$$\zeta[n, 1] = \sum_{k=1}^{n} \phi(S_k, T_1) \text{ and } \zeta[1, m] = \sum_{k=1}^{m} \phi(S_1, T_k) \tag{3.2}$$

During the construction of ζ, we simultaneously create a traceback matrix R that stores the directions 'D' (Diagonal), 'L' (Lower) and 'U' (Upper). We put 'D' in $R[n, m]$ when $\zeta[n-1, m-1]$ is minimum out of $\zeta[n-1, m-1], \zeta[n-1, m]$ and $\zeta[n, m-1]$. Similarly 'L' and 'U' are also stored.

Step** 2**: Score Computation

The distance score between sequences $\mathbb{S}$ and $\mathbb{T}$ is determined by finding optimal warping (alignment) path in matrix ζ. An optimal warping path between sequences $\mathbb{S}$ and $\mathbb{T}$ is a sequence $\mathbb{P} = (P_1, P_2 \cdots P_L)$ of length L where each P_l represents the cell (n_l, m_l) in ζ. This optimal path is the path having minimum

total cost

$$\Psi_{\mathbb{P}} = \sum_{l=1}^{L} \zeta[S_{n_l}, T_{m_l}] \tag{3.3}$$

subject to following three conditions as mentioned in [103, 104].

(*i*) *Boundary Condition:* This condition enforces that the end points of the sequences must be aligned means $P_1 = (1,1)$ *and* $P_L = (N,M)$.

(*ii*) *Monotonicity Condition:* This condition reflects the requirement of faithful timing. If an element in $\mathbb{S}$ precedes a second one this should also hold for the corresponding elements in $\mathbb{T}$, and vice versa.
$n_1 \leq n_2 \leq \cdots \leq n_L$ and $m_1 \leq m_2 \leq \cdots \leq m_L$

(*iii*) *Unit-Step Size Condition:* This condition ensures that the path $\mathbb{P}$ should not skip any alignment information of system-calls in both the sequences and there must be no replications in the alignment.

$$P_l - P_{l-1} \in \{(0,1),(1,0),(1,1)\} \text{ for } l \in [1,L]$$

Figure 3.4 illustrates the three conditions. In Figure 3.4(a), the warping path satisfies all the three conditions. Figures 3.4(b),(c),(d) illustrate violation of one of the conditions. In Figure 3.4(b), the path neither starts from index $(1,1)$ nor ends at index $(9,7)$, thus boundary condition is violated. In Figure 3.4(c), the path from index $(4,5)$ goes next to index $(5,4)$ *i.e.*, monotonicity condition is violated and lastly Figure 3.4(d) shows the violation of step size condition because in the path, there are two consecutive indices as $(4,5)$ and $(6,5)$ and difference between these two is $(2,0)$.

Now, The optimal warping path between $\mathbb{S}$ and $\mathbb{T}$ is a warping path $\mathbb{P}^*$ having minimal total cost among all possible warping paths. The total cost of path $\mathbb{P}$ is calculated using Equation 3.3. The path $\mathbb{P}^*$ is determined by traversing matrix ζ starting from index (N,M) and backtracking along the minimum $\zeta[n,m]$ until we reach cell $(1,1)$.

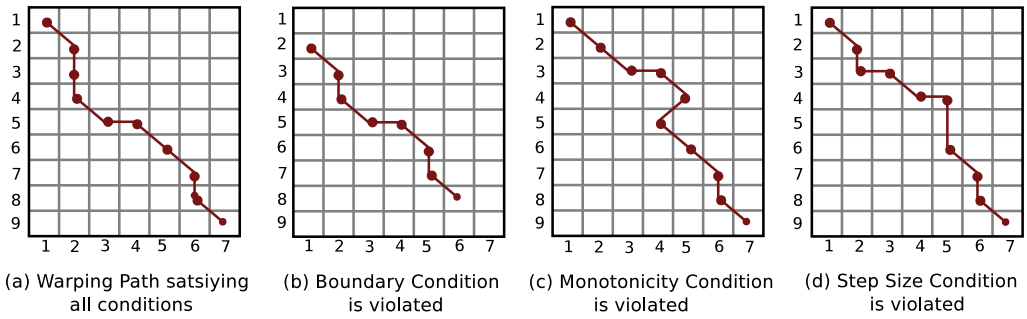


FIGURE 3.4: Illustration of conditions of warping path with sequences $\mathbb{S}$ of length 9 and sequence $\mathbb{T}$ of length 7

The whole process of DTW is illustrated in Figure 3.5. Here, we have considered two sequences with $\mathbb{S} = ABDFIOHYZ$ of length 9 and $\mathbb{T} = ABDEXCG$ of length 7. Formation of matrix ζ of size 9×7 is explained in Figure 3.5(a). The first row and first column of the matrix are computed using Equation 3.2. The rest of the matrix cells are filled using Equation 3.1. For instance, consider cell at index $(3, 3)$ in Figure 3.5(a). This cell value is calculated by adding local cost measure $\phi(3, 3) = 1$ into the minimum of $(2, 2)$, $(3, 2)$ and $(2, 3)$ cell values which is 0. So the final value of $\zeta[3, 3]$ is 1. In similar manner, entire matrix is filled.

After construction of ζ, we trace back in the matrix to get the optimal path. For tracing the matrix, we start from index $(8, 6)$ as shown in Figure 3.5(b). We check three indices *i.e.*, $(7, 5)$, $(8, 5)$ and $(7, 6)$. Minimum of these three cells is at index $(7, 5)$ so we include the index in optimal warping path and ζ value at this cell is added in $\Psi_{\mathbb{P}}$. Similarly, we trace the matrix until we reach at index $(1, 1)$.

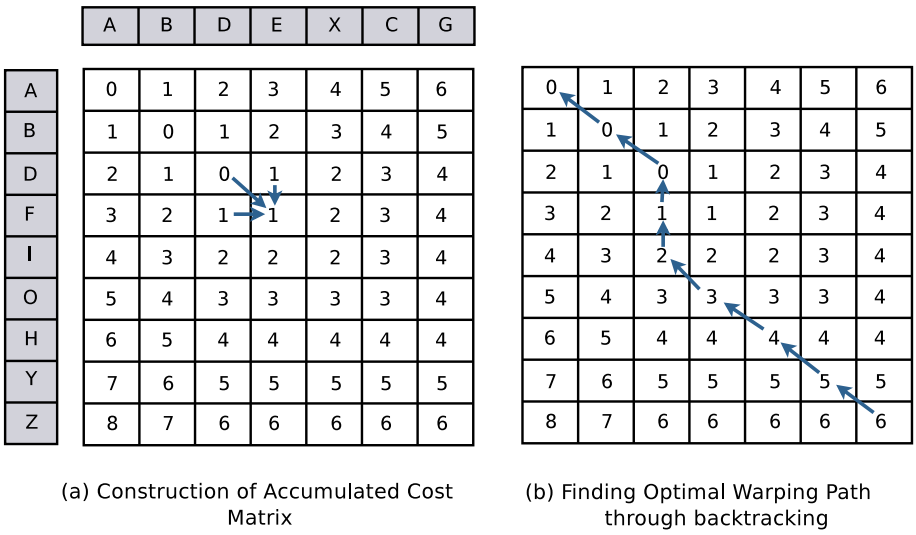


FIGURE 3.5: DTW: an example

Finally, we get optimal aligned path $\mathbb{P}$ of two sequences $\mathbb{S}$ and $\mathbb{T}$ and its cost $\Psi_{\mathbb{P}}$. The value of $\Psi_{\mathbb{P}}$ for our example is $\Psi_{\mathbb{P}} = 0 + 0 + 0 + 1 + 2 + 3 + 4 + 5 + 6 = 21$ which is termed as DTW score.

3.3.3 Behavior Clustering

To construct similar groups, we utilize acquired DTW scores of all pair of samples. These pair-wise scores are stored in a distance matrix D. We form two distance matrices D_w and D_b representing two cases 1) worm versus worm and 2) worm versus benign respectively. The first case is to determine whether all worms exhibit same behavior or not. And the second case is to assure that using the distance scores worm and benign samples are differentiated or not. After forming score matrix, we compute the mean score of a worm sample with all the other samples of dataset (worm and benign). In such manner, the mean scores of all worm samples is computed. These mean scores are then used to categorize and detect the worms.

Visualizing DTW scores gives us different behavior clusters. For better categorization, we need to apply a clustering algorithm that also automates the process. Also with larger datasets, effective visualization becomes non-trivial. To make our approach scalable for large datasets, we applied distance based hierarchical clustering. This clustering method allowed us to identify the most concise and well dispersed clusters.

We incorporated single linkage hierarchical clustering on acquired distance matrix D. Each pair D_{ij} in D denotes the distance score between i^{th} and j^{th} sample. This clustering method takes D as input and generates a tree like structure named as *dendrogram*. The leaves and edge length in dendrogram represent the D_{ij} values and distance between clusters respectively. To identify the number of clusters, we need to cut the dendrogram at a certain height h. To decide appropriate value of h, we applied a standard cluster validity criteria Davies-Bouldin index (DBI) [90, 105]

that is computed using Equation 3.4.

$$DBI = \frac{1}{K}\sum_{i=1}^{K} \max_{0\leq j\leq K, j\neq i}\left\{\frac{\Delta_i + \Delta_j}{\Delta_{ij}}\right\} \tag{3.4}$$

In Equation 3.4, Δ_i represents the average distance of all points in cluster C_i to its centroid. The value Δ_{ij} represents the distance between the centroids of C_i and C_j. It is clear that DBI is average similarity between each cluster C_i *where* $i = 1, 2, \cdots, K$. For clustering, the similarity between clusters should be minimum and it can be achieved by cutting the dendrogram at height h such that number of clusters (K) obtained minimize the DBI index [105].

Figure 3.6 illustrates the cluster formation by generating dendrogram using mean distance scores. The matrix in this figure shows the mean distance scores plotted of worm samples. Each point in the matrix is considered an individual point for dendrogram construction. In dendrogram, point 4 and 5 are clustered first, then point 1 and 3 are grouped in one cluster. Point 2 is more close to cluster – $\langle 4, 5\rangle$. Therefore, it is clustered into $\langle 4, 5\rangle$. At last all the clusters are merged into one cluster. As discussed earlier, we have used DBI index to cut the dendrogram at appropriate height so that we can determine exact number of clusters formed.

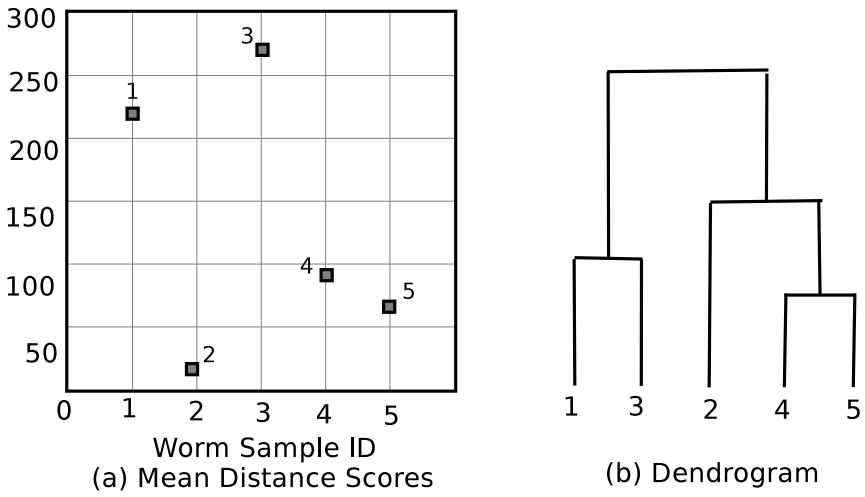


FIGURE 3.6: Cluster formation: an example

3.4 Experiment Setup

We have used Windows32 PE (portable executable) binaries as input. According to the statistics of *virustotal.com*, this file format is the first choice of hacker community for spreading infection. A GPU (Nvidia Tesla C2075) with 448 streaming processors (575MHz core clock, 1150MHz shader clock) is employed. To improve the performance of DTW algorithm, we have also implemented the parallel version of DTW that has been discussed later. For this, we have used a CUDA system that utilizes Nvidia Driver version 304.54 and CUDA toolkit 5.0.

3.4.1 Dataset Preparation

In our approach, we use system-call traces of worm and benign executables. These executable are executed in Windows XP platform. Although, Microsoft has abandoned its support to Windows XP yet this will not affect our proposed approach as 1) our target worm binaries infect all Windows platforms and 2) the system-calls used in Windows XP is the subset of those used in Windows 7 [102].

Table 3.1 shows the sample distribution of our datasets. The benign dataset contains total 1415 executables that are gathered from Windows/system32 directory of freshly installed Windows system. Our worm dataset consists of 1458 samples collected from on-line sources and user agencies and labeled as worm using three different AV scanners (Norton, Quick Heal, AVG). The worm dataset includes samples from *agent, ailis, aimven, alphx, anilogo, antinny, apart, autoit*, and *autorun* subfamilies. Both the datasets are divided into two sets such that Dataset 1 is used for training and Dataset 2 is used for testing phase.

TABLE 3.1: Sample distribution

	Dataset1 (70%)	**Dataset2 (30%)**	**Total**
Worm	1020	438	1458
Benign	990	425	1415

3.4.2 Worm Categorization and Detection

As discussed earlier, we construct distance matrices D_w and D_b that store pairwise alignment scores of worm samples with worms and benign dataset respectively. We plot mean distance scores of all worm samples as shown in Figure 3.7 for both the experiments. Equation 3.5 illustrates our mean distance score M_i of i^{th} worm sample. Here, d_{ik} denotes the DTW score of i^{th} worm sample to k^{th} sample (worm or benign). n is the size of dataset (worm or benign).

$$M_i = \frac{1}{n}\sum_{k=1}^{n} d_{ik} \tag{3.5}$$

We obtain high mean score for all the samples. Therefore, we normalize each score dividing it by 10^6 to better visualize the formed groups. From this visualization shown in Figure 3.7, it can be seen that there are 4 major groups formed within the worm family, and benign samples are also separated from worms. The worm groups are labeled as ω_1, ω_2, ω_3, and ω_4. The benign group is labeled as β_1. These initial results with DTW score indicate that 1) our approach can segregate worms into different groups and 2) using DTW, we can differentiate worm and benign samples as well.

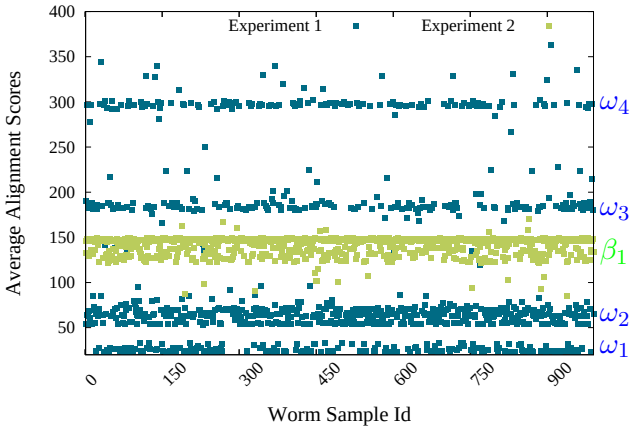


FIGURE 3.7: Alignment scores of worms with worms (experiment 1) and benign (experiment 2)

But in case of large datasets, effective visualization becomes non-trivial. To make our approach scalable for the large datasets we apply distance-based hierarchical clustering [106, 107]. In conjunction to that, this clustering also enables us to

quantify exact number of clusters and number of samples in each cluster. Bayer *et al.* [78] have applied hierarchical clustering approach to form malware clusters considering similarity scores generated after applying Jaccard index. On the other hand, we utilize DTW distance scores as points in hierarchical clustering as our system-call traces are dynamically generated and to capture the similarity within these traces a dynamic algorithm is required.

We have incorporated single linkage hierarchical clustering on acquired distance matrix D. To obtain the exact number of clusters and their sizes, we have applied Davies-Bouldin index (DBI). Figure 3.8 shows the plot of value of DBI versus the number of clusters. The minimum value of DBI is achieved for $K = 14$. We selected $K = 14$ because the DBI value is stabilized from this point onwards.

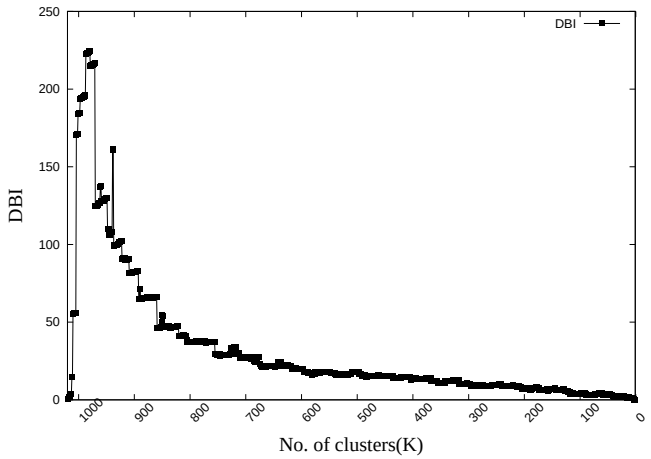


FIGURE 3.8: Davies-Bouldin index (DBI) vs number of clusters

Table 3.2 presents 14 clusters with size of each. From table, we can clearly infer that we find four major clusters namely C_1, C_2, C_3, C_4 with size 477, 173, 126, 192 respectively. All other clusters are of size less than or equal to 15 which is very small with respect to total size of data. We are not denying the fact that other clusters also represent a different behavior. Our results indicate there are four major behavior clusters, samples of which constitute higher intra-cluster similarity than inter-cluster similarity.

Our proposed model can distinguish worms and benign samples as shown in Figure 3.7. From dataset1 samples, we compute worm detection accuracy. From

TABLE 3.2: Worm categorization

Cluster	**Size**	**Cluster**	**Size**
C_1	**477**	C_2	**173**
C_3	**126**	C_4	**192**
C_5	15	C_6	13
C_7	12	C_8	4
C_9	2	C_{10}	1
C_{11}	2	C_{12}	1
C_{13}	1	C_{14}	1

DTW scores the number of correctly and incorrectly instances can be easily calculated. We evaluate the detection capability using four evaluation parameters *viz.* TPR (True Positive Rate), FPR (False Positive Rate), TNR (True Negative Rate) and FNR (False Negative Rate). For any worm detection approach, it is mandatory to have high TPR and low FPR value. We achieve TPR as 98.72%, FNR as 1.27%, TNR as 99.09% and FPR as 0.91%. The obtained evaluation parameters indicate the accuracy of our approach in differentiating worms and benign samples.

3.5 Accelerating Malware Detection: P-DTW

The quadratic time complexity of DTW creates the need for methods to speedup dynamic time warping. The methods used to make DTW faster fall into three categories.

1. *Constraints:* Limit the number of cells that are evaluated in the cost matrix.
2. *Data Abstraction:* Perform DTW on a reduced representation of the data.
3. *Indexing:* Use lower bounding functions to reduce the number of times DTW must be run during time series classification or clustering.

Constraints are widely used to speed up DTW. Two of the most commonly used constraints are Sakoe-Chuba [108] and Itakura Parallelogram [109]. We, in this approach, to reduce the computational cost apply parallel version of DTW as applying any constraint or abstraction mechanism may lead towards information loss.

Talking about the time complexity of both steps in DTW, Step 1 employs dynamic programming approach and creates $N \times M$ matrix and each cell is iterated one time so the time complexity of Step 1 is $O(NM)$ and Step 2 is backtracking step in the matrix D from index (N, M) to index $(1, 1)$. Since the maximum length of path in the matrix will be $N + M$ so the time complexity of this step is $O(N + M)$. Also, our system-call sequences are very long especially in malware samples. Therefore, high computational cost due to size of malware sequences makes the method a slow approach. Since Step 2 does not contain any data independent computations, we expect to reduce complexity of Step 1 by using a parallel high performance computing platform (GPU). It is clear that DTW is computationally intensive algorithm for the large size of input sets (malware are of very large size). So we reduce this complication by implementing the parallel version of DTW named P-DTW using CUDA.

3.5.1 Parallelization Strategy

Figure 3.9 shows the data independence in construction of matrix D. From the Equation 3.1, we can infer that the computation of cell $D[i, j]$ is dependent on three cells $D[i-1, j]$, $D[i, j-1]$ and $D[i-1, j-1]$. In Figure 3.9 we show that $D[3, 3]$ is dependent on $D[2, 2]$, $D[2, 3]$ and $D[3, 2]$ and not dependent on any cells of its anti-diagonal (shown by colored cells in Figure 3.9). This dependence implies that cells of an anti-diagonal can be computed in parallel using CUDA threads. Also, cells of an anti-diagonal are dependent of previous two anti-diagonals so parallelism is limited to only one anti-diagonal. Matrix D of size $N \times M$ contains $(N+M-1)$ anti-diagonals and each anti-diagonal can be computed in one iteration so the time complexity of construction of matrix D reduces to $O(N+M-1)$ from $O(NM)$. For N=M, quadratic complexity on sequential machines becomes linear complexity on parallel machines with a gain factor of $O(N)$.

3.5.2 Memory Assignment Scheme

In our method, matrix D is stored to be used in Step 2 *i.e.*, for finding optimal warping path cost. As total cost is found by tracing back the matrix so whole

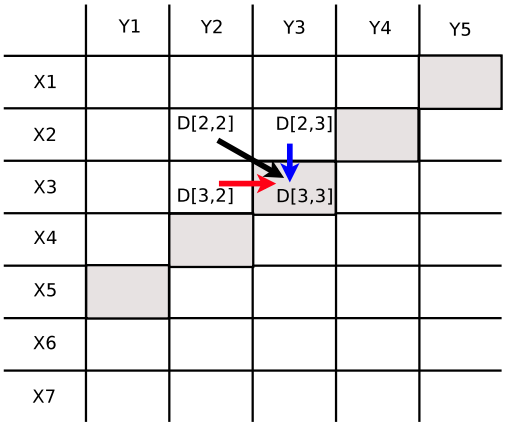


FIGURE 3.9: Data independence in D matrix computation. Cells of an anti-diagonal are independent.

matrix is needed in the computation of total cost. Therefore, our memory requirement for parallel algorithm increases due to size of matrix D. Also, matrix D is needed in each anti-diagonal iteration so we store it in global memory. Similarly, input sequences X and Y are copied to global memory of GPU.

Algorithm 1 shows the pseudocode of our proposed kernel for P-DTW. Let τ denotes the `thread-id`. This kernel is invoked for every thread τ where $1 \leq \tau \leq l$ and l is the length of an anti-diagonal. Each thread computes a matrix cell of an anti-diagonal. Initially, first column and first row of the cost matrix D is filled at CPU (Lines 8-15). After copying input sequences and matrix D to GPU memory, kernel function (Lines 26-36) is invoked. In this function, each thread computes $D[i, j]$ entry according to the Equation 3.1 and $\phi()$ denotes the match/mismatch cost computed using Equation 3.2. Once all the anti-diagonals are completed, `for` loop of Line 10 terminates and we obtain cost matrix D.

3.5.3 P-DTW Constrains

Memory requirement for matrix D is quite large as input malware sequences are quite long. This limitation makes our parallel approach P-DTW applicable for fixed range of input sequences. But as the size limitation does apply in a similar fashion to sequential algorithm of DTW so need to modify it. Sequences above a specified length, are run through the modified sequential algorithm. For each pairwise alignment, the maximum size of D matrix in parallel approach is restricted

Algorithm 1 Parallel Construction of Matrix (Phase 1)

Data Structures
$\mathbb{S}$: Source sequence of length N
$\mathbb{T}$: Target sequence of length M
ζ : Accumulated cost matrix of length $N \times M$
$\mathbb{C}$: Total Cost of path
c : Cost of match/mismatch (0,1) of $\mathbb{S}$ and $\mathbb{T}$
τ : Thread–id of a thread in CUDA kernel

Initialization
$\zeta[1] = c(\mathbb{S}[1], \mathbb{T}[1])$
for $i = 2 : N$ **do**
  //Fill First Column of Matrix ζ.
  $\zeta[i + M] = \zeta[i - 1 + M] + \phi(\mathbb{S}[i], \mathbb{T}[1])$
end for
for $i = 2 : M$ **do**
  //Fill First Row of Matrix ζ.
  $\zeta[i] = \zeta[i - 1] + c(\mathbb{S}[1], \mathbb{T}[i])$
end for

Copy $\mathbb{S}$ and $\mathbb{T}$ array to GPU
Copy ζ array to GPU
$i = 2$
$j = 2$
for $k = 1, N + M - 1$ **do**
  //for all anti–diagonals
  Call kernel function CAL-COST($\mathbb{S}$,$\mathbb{T}$,ζ,i,j,k)
  $i = i + 1$
  $j = j + 1$
end for
return ζ //Cost Matrix

function CAL-COST($\mathbb{S}, \mathbb{T}, \zeta, i, j, k$) //Kernel Function
if $\tau > 0$ and $\tau < i + j - k$ **then**
  $idx = i - \tau$
  $var = (\tau * M) + (i + 1) * (k - i + 1)$
  $diag = \zeta[var - M - 1]$
  $upper = \zeta[var - M]$
  $lower = \zeta[var - 1]$
  Min= min(diag,lower,upper)
  $\zeta[var] = Min + z(\mathbb{S}[k - idx], \mathbb{T}[i])$
end if
=0

to $1200MB$. Beyond that limit, parallel approach will not work for our DTW approach. Given w worm samples, we calculate a total of $w * (w - 1)/2$ pairwise alignments using DTW. Since we have $w = 1226$ so number of alignments we calculate is 750925. After applying size restriction to parallel algorithm, we find that the parallel algorithm works for 738535 alignments, which is 98.3% of total alignments. This way we reduce the execution time of the algorithm at some extent by applying mixed approach of parallel and serial DTW versions.

3.5.4 Comparison with Traditional Clustering Algorithms

Table 3.3 shows the time complexity of traditional clustering algorithms. The computational cost of these algorithms is not less than O(n^2). Our proposed algorithm P-DTW outperforms all the other algorithms as it has time-complexity of O(n).

TABLE 3.3: Comparison of DTW and P-DTW with traditional clustering algorithms

S.No.	Algorithms	Complexity	S.No.	Algorithms	Complexity
1.	Agglomerative	O(n^3)	2.	Divisive	O(2^{n-1})
3.	K-means	O(nki)	4.	CURE	O(n^2)
5.	DTW	O(nm)	6.	P-DTW	O(n)

3.6 Performance Evaluation

The effectiveness of our proposed approach is evaluated using dataset2 (test dataset) and virus dataset.

3.6.1 Evaluation with Other Dataset

This metric refers to the ability of proposed model to perform similarly with different set of malware samples. Also, the proposed approach must not be biased to training samples only, it should work well with test samples as well. We used two datasets- 1) worm test dataset, and 2) virus dataset. Our virus dataset contains a total of 674 number of samples. We applied our approach on test samples and found that with four major groups were formed as with training samples. This indicates that our approach works well with training and testing samples.

Figure 3.10 shows the behavior groups formed within virus samples. It is clearly visible that there are five different groups formed within the virus family. These formed groups also confirm our heuristic that samples of a malware family constitute different behavior during execution. Also, the benign samples are clearly differentiated from virus samples. The overall detection accuracy achieved with virus samples is $\sim$93%.

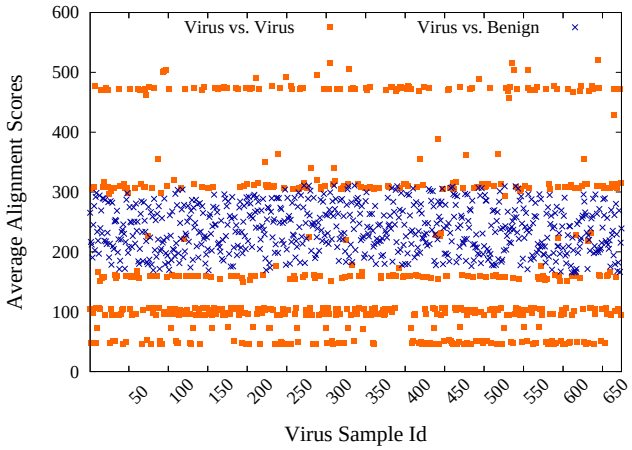


FIGURE 3.10: Alignment scores of virus with virus (experiment 1) and benign (experiment 2) samples

3.6.2 Detection Accuracy

We achieved overall testing accuracy as 98.27% that is nearly same to the overall accuracy 98.90% of dataset1. This indicates that our proposed model can detect worm with same detection accuracy with different samples as well. As discussed earlier, the low FPR and high TPR are desirable in any worm detection approach. With test dataset, we obtain TPR as 97.87% and FPR as 1.33%. As far as virus dataset is concerned, the overall accuracy obtained is 92.95%.

3.6.3 System Overhead

We have discussed about memory constraints of P-DTW. Since P-DTW can be utilized for 99.24% pairs of alignments of total worm samples. We have evaluated the overhead of these pairs only with DTW and P-DTW. The performance can be seen as speed up ratio of P-DTW over DTW for different pairs of samples with varying sizes. Figure 3.11 shows the comparison between P-DTW and DTW for variable size of pairs.

This figure shows that the parallel algorithm P-DTW works faster as we increase size of samples. There is a mandatory communication overhead involved in data transfer between CPU and GPU. This becomes smaller percentage of overall running time with bigger sequence. Therefore, in small sequences we have achieved

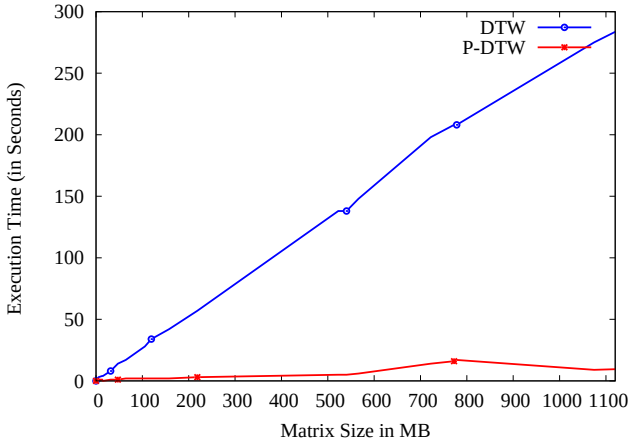


FIGURE 3.11: Comparison of P-DTW with DTW

low amount of speedup in comparison with larger sequences. The maximum speed up which is obtained is 30.55 times with matrix ζ size of $772380KB$.

3.7 Summary

Present malicious threats are causing substantial financial damage to the users connected through network infrastructure. Analysis of malware attack vector becomes the prime necessity of security researchers to mitigate the monetary losses. Malware analysis is a time-consuming task as there are billions of unseen and new malware programs. Therefore, a categorization within a malware family is required to reduce analysis time of a security researcher.

In this chapter, we have presented an approach that categorizes and detects malware samples on the basis of their run-time behavior. Our experiments validate the heuristic that samples belonging to same malware family constitute different behaviors. We have applied DTW algorithm that captures the behavior similarity/dissimilarity among worm and virus samples. In addition to that, we have created P-DTW algorithm to improve the performance of sequential DTW.

The diversity in behavior indicates the presence of anti-detection payloads in malware samples. To address the anti-detection features of modern malware, in next chapter, we proposed a malware detection and categorization approach. We aim at identifying and classifying malware samples with environment-reactive behavior.

Chapter 4

Environment-Reactive Malware Behavior: Detection and Classification

Present malicious threats have been consolidated in past few years by incorporating diverse stealthy techniques. So, proactive malware detection is not always possible [110]. Proactivity refers to a technique's ability of detecting new and unseen malicious codes. Existing anti-malware techniques have advanced in recent past; still there are some sophisticated malware programs that evade security mechanisms and contaminate our systems. In such situations, the dynamic malware detection approaches are preferred to detect current detection-aware malware programs. But, these approaches suffer from various anti-detection measures embedded within malware programs.

In the previous chapter, we have shown that within a malware family, different behaviors from different samples cluster into finite set of behaviors during run-time. These behaviors give an indication that malware binaries carry multiple payloads that are delivered after certain conditions are met. One such anti-detection measure is to apply various analysis-aware (environment-aware) checks to bypass dynamic malware detection methods. In this chapter, we detect and categorize the

environment-aware malware. Malware bundled with environment-aware payload result in degraded detection accuracy of existing detection approaches. These malicious programs detect the presence of execution environment and not revealing their malicious payload they mimic a benign behavior to avoid detection. In this approach, we make use of system-calls to identify malware on the basis of their malignant and environment-reactive behavior. The proposed approach offers an automated screening mechanism to segregate malware samples on the basis of aforementioned behaviors.

4.1 Introduction

The most of the dynamic malware detection approaches adopt a standard procedure to monitor the run-time behavior of malware. During the monitoring process, a safe and isolated environment set-up is established to protect the host from any malware generated side-effects. A variety of sandboxing, virtualization and emulator-based tools are available to create such set-ups. These controlled environments imitate a real runtime environment. But a full imitation of real system cannot be achieved as these environments differ in CPU semantics (CPU semantic attacks) and in instruction execution time (timing attacks) as compared to uninstrumented host [66].

Malware writers have embedded various checks to detect the presence of analysis environment. Rutkowska developed a technique Red Pill [111], which shows that incorporating CPU semantic attack using SIDT (Store Interrupt Descriptor Table) is an effortless task. He demonstrated that the value of non-privileged SIDT (Store Interrupt Descriptor Table) instruction in virtual machine was not similar to the value in real machine. Also, by fingerprinting of system artifacts of the analysis environment, one can easily detect the presence of instrumented environments. These artifacts include processes running in the background, registry keys, system functions, and IP addresses [95].

To resolve the problem of analysis-aware malware, researchers have developed detection techniques [37, 95, 112–114]. These techniques capture malware by noting

its behavior deviation in different environments. The main heuristic behind these techniques is that an analysis-aware malware will have different behavior profiles when executed in different analysis frameworks. This behavior diversion occurs due to fingerprinting of system artifacts. Existing approaches [37, 95, 113, 114] have utilized virtualized or emulated frameworks for marking behavior deviations. Unfortunately, these approaches suffer from following limitations.

1. Overhead of running a sample in multiple analysis frameworks.
2. If malware program detects these instrumented frameworks then it is possible that all the behavior profiles are same.
3. According to [112] and our observations, executing a sample multiple times in same environment results into different behavior profiles.

Due to reasons listed above, using behavior deviation in multiple frameworks, for capturing analysis-aware malware, is not a reliable indicator. Balzarotti *et al.* [112] have applied a record and replay mechanism to identify malware analysis-aware checks. The authors have recorded the malware execution in a reference host system and then replayed the recorded system-call traces in an emulator to observe any behavior deviation. Any deviation in recorded and replayed execution assumed to be an indication of anti-analysis checks. The replay mechanism adopted by authors in [112] lacks some implementation issues such as 1) incorrect reply of applications that rely on random number to decide certain execution path, and 2) their replay mechanism does not work with applications composed of multiple processes. Thus, for detecting analysis-aware malware we require an approach that is not

- *Resource-hungry*, means it does not require multiple frameworks.
- *low Monitoring-overhead*, means it does not execute a sample multiple times.

In this chapter, we propose an approach that is not resource-hungry as well as free from monitoring-overhead of multiple frameworks. The analysis-aware malware can detect analysis environment and react by terminating, crashing or stalling the

execution to avoid performing any malicious activity [95, 112, 114]. The proposed approach exploits the reaction (crashing the guest OS and stalling the execution) of malware to evade any analysis mechanism. We term such malware as "environment-reactive".

In literature, different terms have been utilized to address this category of malware such as environment-sensitive, environment-resistant, environment-aware, analysis-aware and split-personality. We have used the term 'environment-reactive' for the same. Malware with environment-reactive behavior incorporates two activities 1) sensing the presence of virtual environment and 2) responding to this environment sensing by showing an unusual behavior such as crashing the guest OS and stalling the execution to evade time-outs. However, malware programs that do not manifest such reactive behavior inspite of embedded with analysis-aware checks cannot be termed as environment-reactive. We, in this approach target only those malware binaries that react to environment-sensing.

In this chapter, we have devised a mechanism that transforms the manual behavior screening into the automated one. Our proposed approach exploits malware's tactics of evading detection and predicts its reactive and malicious behavior under a host-based virtualized environment (Ether [46]). Though the proposed approach is specific to a given monitoring environment but can be generalized by applying same methodology with other environments also. We construct an input vector that is best suited for our learning model. The input vector consists of transition probabilities of two consecutive system-calls. This input vector is fed into the multi-class decision model. The decision model is based on multi-layer perceptron learning algorithm with back propagation. Each network is trained and tested in parallel to reduce the performance overhead. Our experimental results indicate that the proposed model is capable of 1) finding the known and unknown instances of malware binaries which are environment-reactive 2) improving the monitoring mechanism of Ether by analyzing the detected binaries.

4.2 Approach Overview

The main objective of our approach is to identify the malware's environment-reactive behaviors. In addition to that, our approach also discriminates the benign and malware (Non environment-reactive) executables. To achieve these objectives, we executed binaries in a virtualized environment using Ether and noted the interaction of binary with Kernel in terms of system-calls. On the basis of noted behaviors, we have classified samples in four categories. Figure 4.1 outlines proposed classification framework. As can be seen in the figure, three major components of proposed method are 1) Analysis Framework, 2) Behavior Representation, and 3) Decision Model. In our proposal, we consider four behavior – 1) Clean, 2) Malignant, 3) Guest-crashing, and 4) Infinite-running.

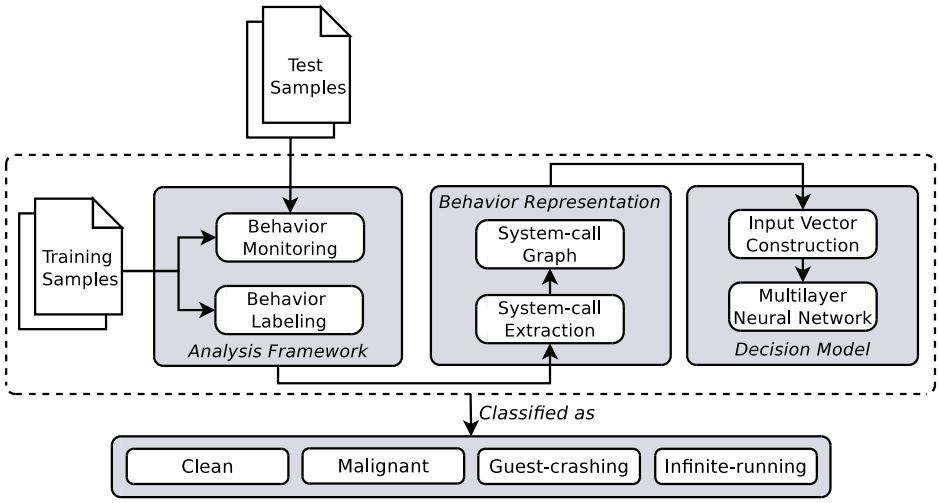


FIGURE 4.1: Proposed classification framework

4.2.1 Analysis Framework

The analysis framework plays an important role in analyzing the behavior of malicious binaries. It must provide an isolated and transparent environment setup that does not relay any side-effects to host. We have used Ether as our dynamic analysis framework. Our approach relies on Ether for behavior monitoring and labeling of benign and malware.

Behavior Monitoring: Behavior monitoring of binaries using Ether is performed in similar fashion as discussed in Chapter 3 and Appendix A.

Behavior Labeling: According to [46] Ether is capable of capturing execution traces of all malware and remains invisible to the target application being monitored. We have noticed that there are samples for which no logs are generated. It indicates that Ether is not completely transparent. Also, Pek *et. al.* [99] have shown in their approach (nEther) that Ether is prone to timing attack. In such situations, when Ether is detected by an application the acquired logs cannot be trusted. So, we applied a close manual monitoring of malware behavior of training samples and utilized these noted behaviors to detect the maliciousness of an unknown sample. In the training phase, we adapted following definitions of clean and malicious behavior of benign and malware binaries.

$$\mathbb{P} = \begin{cases} \text{Suspicious} & \{\beta \mid \beta \, \epsilon \{M,\ G,\ I\}\} \\ \text{Clean} & \text{Otherwise} \end{cases} \tag{4.1}$$

A program $\mathbb{P}$ is said to be suspicious or clean if it depicts a behavior β in one of the four forms 1) Malignant (M), 2) Guest-crashing (G), 3) Infinite-running (I) and 4) Clean (C). All four are discussed as follows.

1. *Malignant (Non-Environment-Reactive):* This class of malware depicts the non-environment-reactive behavior of malware binary. These malicious programs generate system-call logs within our specified time-frame and do not constitute any visible abnormal activity that alters the state of guest OS.

2. *Guest-crashing (Environment-Reactive):* Guest-crashing behavior is marked for those samples that crash the guest OS to terminate the execution-monitoring process. To incorporate such a mechanism in the malware source code is not a difficult task. The system can be crashed by causing a page fault, exception and access violation [115].

3. *Infinite-running (Environment-Reactive):* We labeled a malware sample as having infinite running behavior if the target sample does not terminate in 10 minutes and respective logs show repetitions. Most of dynamic analysis techniques run a sample for some time and check for malicious activity. To circumvent such detection techniques, malware employ a loop with many iterations. Once monitoring is over, malware binaries eagerly wait just for

a single chance of execution and if granted it will try to infect the machine as soon as possible. The infinite running behavior indicates that malware is running in an infinite loop and try to add non-malicious sequence in generated logs.

4. *Clean (Non Environment-Reactive):* We marked each benign application with this behavior. The benign executables do not reflect any system crash or infinite running behavior. Our benign dataset does not include any large setup and installation files having execution time more than 10 minutes.

4.2.2 Behavior Representation

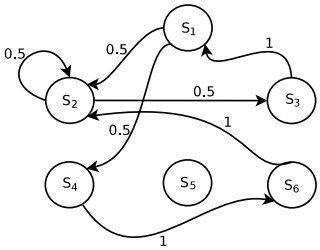
As discussed in previous chapter, we have utilized system-calls to model program behavior during execution. In our approach, the acquired system-call sequences are transformed as system-call graph that is based on Markov model.

The Markov model based graph representation enables the consolidated comparisons in two-dimensional space and maintains the sequential nature of data [66]. Representing system-call sequences in this way hampers malware author's aim of evading detection of any system-call based approach. As the malware authors can very conveniently re-arrange or insert irrelevant system-calls in their malware source code [116]. According to [102] and our traces, there are 284 unique system-calls that can be invoked by any running application in Windows XP.

System-call Graph: Let $\boldsymbol{\xi}$ is an execution trace of a sample that represents the set of system-calls invoked by the sample. $\boldsymbol{\xi}$ is further transformed into weighted directed graph $G = \{V, E\}$, where V is a finite set of 284 unique system-calls and E represents set of edges in G. Every edge e_{ij} indicates a transition from node i to j with transition probability ρ_{ij}. Applying Markov property (Equation 4.2), we built our Transition Probability Matrix (TPM).

$$\sum_{j=1}^{284} \rho_{ij} = \begin{bmatrix} 0 & if\ all\ entries\ in\ i^{th}\ row\ are\ zero \\ 1 & otherwise \end{bmatrix} \tag{4.2}$$

For example, consider the execution trace $\boldsymbol{\xi} = \{S_1, S_2, S_3, S_1, S_4, S_6, S_2, S_2, S_3, S_1\}$ of a program $\mathbb{P}$. Figure 4.2 shows the corresponding graph and matrix for this example. Here, program $\mathbb{P}$ invokes 5 unique system-calls $(S_1, S_2, S_3, S_4, S_6)$ and call S_5 is not utilized in its execution path. The graph contains 5 connected nodes and one isolated node. Edges are directed (showing transition direction) and labeled with transition probability ρ_{ij}. The matrix representation of $\mathbb{P}$ shows a 6×6 square matrix called as TPM. Every row in TPM adds to either 1 or 0.



	S_1	S_2	S_3	S_4	S_5	S_6
S_1	0	0.5	0	0.5	0	0
S_2	0	0.5	0.5	0	0	0
S_3	1	0	0	0	0	0
S_4	0	0	0	0	0	1
S_5	0	0	0	0	0	0
S_6	0	1	0	0	0	0

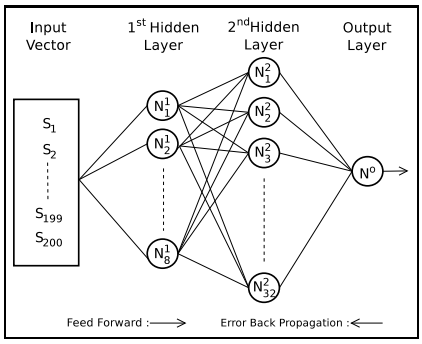
FIGURE 4.2: System-call graph and TPM: an example

In our experimentation, we have constructed 284×284 TPM for each benign and malware executable. These individual TPMs are used to form a composite matrix of a dataset as shown in Equation 4.3. Here, composite matrix (R.H.S.) is constructed by adding two TPMs (L.H.S.) and then dividing each cell (i, j) of resultant matrix by the number of samples (2 in this case). In the similar manner, we have created the composite matrix of our datasets which is further used to construct our input vector. Each cell in this matrix can have a real value ranging from $[0, 1]$ and is indicative of mean transition probability from one state to other in a dataset.

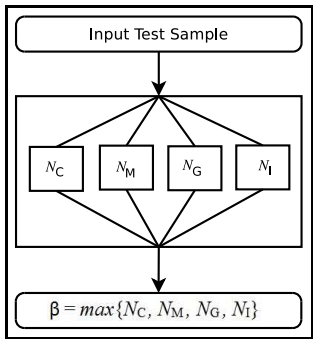
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \frac{1}{2} \begin{bmatrix} (a_{11}+b_{11}) & (a_{12}+b_{12}) & (a_{13}+b_{13}) \\ (a_{21}+b_{21}) & (a_{22}+b_{22}) & (a_{23}+b_{23}) \\ (a_{31}+b_{31}) & (a_{32}+b_{32}) & (a_{33}+b_{33}) \end{bmatrix} \tag{4.3}$$

4.2.3 Decision Model

We used neural network [117, 118] model to categorize the aforementioned behaviors. This model has the ability of learning non-linear discriminant function and recognizing patterns in high-dimensional feature space. It can be structured



(a) Architecture (200:8:32:1)



(b) Decision function

FIGURE 4.3: Decision model

using either single-layer or Multi-Layer Perceptron (MLP). Our initial experiments with single-layer perceptron indicate that our data is not linearly separable. For this reason, we adopted MLP [117] model with error back propagation algorithm for our classification methodology. In our implementation, we have preferred one-against-all (OAA) [118] pattern modeling to one-against-one (OAO) and P-against-Q (PAQ) because other two modeling methods require more number of network structures and shall result into a computational overhead. In our case, we need four bi-class discrimination models and therefore we have developed four different networks, one for each of the behaviors. These networks are

1. Clean vs All (N_C)
2. Malignant vs All (N_M)
3. Guest-crashing vs All (N_G)
4. Infinite-running vs All (N_I)

4.2.3.1 Multilayer Neural Network

Figure 4.3(a) shows our decision model (200 : 8 : 32 : 1). In this figure, the architecture of our proposed network is described by one input vector, two hidden layers, and one output layer. These three components decide the structure of the neural network and are discussed as follows.

1. ***Input Vector:*** Input vector plays a key role in designing a decision network that yields into a better detection accuracy. We have chosen transition states of the composite matrix to be used as input in our constructed input vector. The composite matrix is a 284×284 matrix so our state space will have total 284×284 transitions. This state space is very large and applying it to our model will degrade its performance. The selection of transition states and size of input vector are discussed in Section 4.3.2.

2. ***Hidden Layer and Neuron Count:*** The proposed classification model is designed with two hidden layers instead of one according to the findings of the work in [119]. We have adopted hit and trial strategy to select the number of neurons in the hidden layer as low and high neuron count may result into under-fitting and over-fitting that further lead towards poor learning of training sets. In this quest, we performed an extensive experimentation by using neuron count from 4 to 40 at both the hidden layers. We observed best results with 200:8:32:1 in terms of accuracy and training time and hence decided for 8 neurons in the first hidden layer and 32 neurons in the second hidden layer.

3. ***Output Layer:*** At the output layer, a single neuron is sufficient in our case since for each network, we need a single value that gives a measure of how closely an input sample relates with the corresponding behavior.

Besides these structural issues, there are factors which play a crucial role in designing a decision neural network. These factors are described as follows.

1. ***Activation Function:*** Every neuron is associated with a bias value and makes use of an activation function to transform the value of the activation level into an output signal. If the learning algorithm is back propagation, it is necessary for the activation function to be differentiable. For our model, we have selected $\frac{tanh(x)}{No.\ of\ neurons\ in\ the\ layer}$ as the activation function as i) it is highly differentiable and leads to good gradient descend on error, ii) it requires less mathematical computations on each neuron, and iii) it gives output in the range -1 to $+1$. We have trained our neural network in a way

that a positive value is expected for a positive class and negative value for negative class. For instance, in case of network N_G, the guest-crashing samples belong to positive class and all the other samples are put into negative class.

2. ***Learning Rate and Momentum Rate:*** Learning rate is a numerical factor by which weights are updated in each iteration. Lower the learning rate, finer tuned are the weights, but it requires more iterations, thereby leading to a high training time. Higher learning rate results in less overall training time with the lesser accuracy of detection. Also, during the learning process a momentum factor is introduced to reduce the sensitivity of the network to a local minima with respect to weights and increases the convergence speed. For our model, we have observed good classification results and appreciable training time with 0.01 as the value of the learning rate and 0.5 as the momentum factor value.

3. ***Decision Function:*** When a sample is to be classified, its output from each network is generated and passed through a decision function for a final classification. A high positive output from a network indicates a close match of the input sample to the corresponding behavior. Figure 4.3(b) illustrates the process of determining class of an input test sample. The sample is supplied into all four networks and four output values O_C, O_M, O_G, O_I from networks N_C, N_M, N_G, and N_I respectively are generated in parallel with respect to each network. It is labeled with behavior β where β is the maximum value out of four generated values.

$$\text{class}(S) = \arg\max \begin{bmatrix} O_C(S) = \text{Output of } N_C \text{ on Sample } S \\ O_M(S) = \text{Output of } N_M \text{ on Sample } S \\ O_G(S) = \text{Output of } N_G \text{ on Sample } S \\ O_I(S) = \text{Output of } N_I \text{ on Sample } S \end{bmatrix} \tag{4.4}$$

As described earlier, we have used error back propagation technique to detect the behavior class of a sample. If d is the desired output and O_Y is the obtained

output for a training sample for a network Y, the total error in the network is defined by Equation 4.5:

$$E = \frac{1}{2}(d - O_Y)^2 \tag{4.5}$$

Our aim is to minimize error E by adjusting weights that is done by back propagating the gradient descend on error. At Each iteration, the weights are adjusted. This process is repeated till the error E get minimized.

4.2.3.2 Feature Vector

As discussed earlier, we have trained four behavior networks with respect to each behavior class. According to the selected input vector (containing transition states), each row of feature vector is filled. For example, if input vector contains three transition states $\langle S_1, S_2\rangle$, $\langle S_4, S_6\rangle$, and $\langle S_3, S_5\rangle$ and for any program the respective transition probabilities are ρ_{12}, ρ_{46}, and ρ_{35}.

4.3 Experimental Setup and Results

The experiments were performed on Intel Core i7 2.30 GHz with 8 GB, 1600 MHz DDR3 RAM Macbook Pro. The implementation code is written in JAVA (Eclipse IDE) and executed in JRE environment with 2.5GB and 3.0GB of heap space. We have evaluated the run-time performance of our approach using two different heap sizes.

4.3.1 Dataset Preparation

We have used malware and benign executables as input. A total of 1150 benign executables are gathered from `Windows/system32` directory of freshly installed Windows XP with service pack 2. Our proposed model relies on the system-call sequence gathered from a target binary while it is being executed in Windows XP platform. Although Microsoft abandons its support for Windows XP yet this will

not affect our proposed model because i) the target malicious binaries (win32 PE) affect the all Windows platform, ii) system-call sequence used in Windows XP is a subset of those used in Windows 7 [102].

Our malware dataset consists of 1120 samples collected from online sources and user agencies. All malicious samples are then analyzed by three different AV scanners (Norton, Quick Heal, AVG) to segregate them into their respective malware families (Worm, Trojan, Virus). The training and test sets are derived from benign and malware samples. In our case, training set is 70% and test set is 30%. These sets are non-overlapping. The test set consists of samples whose behavior is known, but these are not used for training. We have labeled each known malware and benign samples with their respective behavior. We labeled 1150 benign samples as Clean, 505 malware samples as Malignant, 329 malware samples as Guest-crashing (∼29%) and 286 (∼25%) malware samples as Infinite-running. Even in our small dataset, more than 50% of malware samples depict environment-reactive behavior reinforcing our motive of detecting environment-reactive malware. Each malware behavior class is divided into training and test set.

We have used one unknown test set, samples of which is not labeled with any behavior to check whether our model makes an accurate behavior prediction for unknown instances. The total number of these samples is 423. These samples do not belong to our previously mentioned benign and malware dataset. We have used these samples to evaluate the performance of decision model with unknown samples. The predicted behavior is verified for each unknown instance in Ether framework, results of which are discussed later.

4.3.2 Input Vector Construction

As discussed earlier, we have used composite matrix for input vector construction. The composite matrix contain a of total 284×284 transitions. This state space is very large and applying it to our model will degrade its performance. Therefore, to decide the appropriate transition states and size of the input vector we adopted following two strategies.

1. Construct *four* input vectors w.r.t. each behavior network using four composite matrices corresponding to each of the behavior datasets. (Strategy 1)

2. Construct *one* input vector for all four behavior networks using one composite matrix created from training samples of all behavior datasets. (Strategy 2)

We considered $50, 100, 150, 200, 250, \cdots, 1000$ transition states for both the strategies. These transition states are having higher transition values than the remaining states. We have considered only top 1000 states out of 284×284 states as considering all states for selection of input vector is not practically feasible. After this, we applied TPMs of our samples and trained all four networks for fixed 10000 iterations with these strategies. The main aim of this experiment is to select the suitable strategy and size for our input vector. Figure 4.4 shows the results of this experiment in terms of overall error rate with respect to each of the 12 (two strategies and six sizes) experiments.

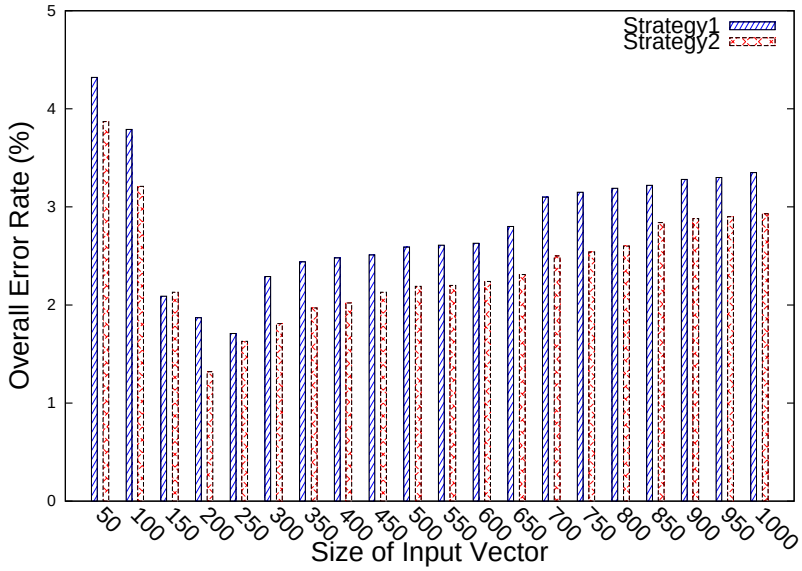


FIGURE 4.4: Input vector construction

As can be seen in the Figure 4.4, we found that Strategy 2 outperforms the Strategy 1 as the individual choice of input vector trains the model for relevant behavior class and ignores the global knowledge. Though, there is a marginal difference in the error rate but for any malware detection system this difference cannot be avoided. Therefore, we have considered Strategy 2 for our experimentation. The minimum error rate is obtained at input size 200 for Strategy 2. We have observed

that the error rate increases as we increase the input size. Because adding more states to our input vector, will also increase the noise in the data and as a result it will lead to poor detection accuracy. Also, we have observed that the training time is directly proportional to input size. But, we decided to sacrifice training time over detection accuracy and fixed the input vector size as 200.

4.3.3 Training Results

In training phase, we randomized samples after each epoch to avoid any biases while adjusting weights. Figure 4.5 shows the graph of error rate v/s number of iterations. This figure indicates that our model converges in atmost 5000 iterations as the overall error rate get stabilized from this point onwards. Though the training time increases with increase in number of iterations and selecting low number of iteration will improve the time performance yet a model not converged reflects an incompletely trained model. Such a model cannot assure an accurate decision model as the error stability is must in such type of networks. As training time is one time cost, a higher training time to achieve higher detection accuracy for unknown test samples is acceptable.

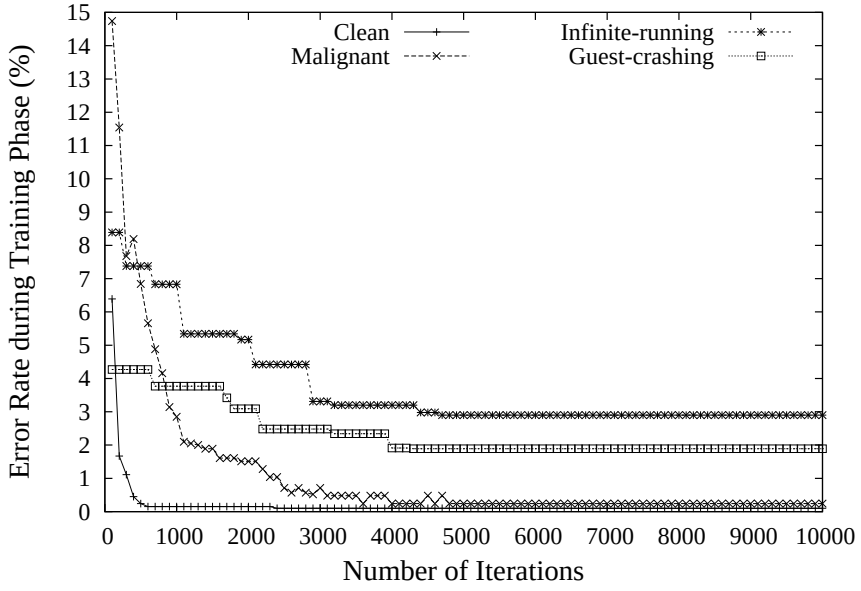


FIGURE 4.5: Number of iteration selection

We evaluated our proposed model's performance using TPR, FPR, TNR, FNR, Accuracy, and Error [120]. Table 4.1 illustrates the performance of constructed trained neural networks. A high value of TPR and low value of FPR indicate that the developed model is performing well. Our model discriminates the clean and malignant behavior with higher accuracy of 99.9% and 99.24%. The false-alarm rate in these cases is negligible which indicates that our selected input vector is significantly diverse in identifying samples with clean and malignant behavior. Remaining two behavior classes indicate the categorization within a malware class, so here we expected the overlapping. The infinite-running and guest-crashing are two environment-reactive behaviors that may not always disclose their maliciousness such malware try to mimic the clean behavior. Therefore, we are observing a false-alarm rate in these two cases.

TABLE 4.1: Performance evaluation with training and known test datasets

Dataset	**Network**	**TPR**	**FPR**	**TNR**	**FNR**	**Accuracy**
Training	*C* vs All	99.8	0	100	0.2	99.9
	M vs All	98.56	0.09	99.91	1.44	99.24
	I vs All	95.12	0.92	99.08	4.88	97.1
	G vs All	97.01	0.24	99.76	2.99	98.39
Test	*C* vs All	97.22	3.4	96.6	2.78	96.91
	M vs All	97.12	1.59	98.41	2.88	97.76
	I vs All	95.56	3.04	96.96	4.44	96.26
	G vs All	92.8	5.53	94.47	7.2	93.64

4.3.4 Testing Phase

We have conducted testing for our known and unknown test datasets. As illustrated earlier, the known test samples are labeled with their respective behavior but are not utilized during the training phase. Table 4.1 shows the results of our known test dataset. We have observed that the testing results are quite similar to the training ones. There is a tolerable difference in detection accuracy of our training and test datasets because of our model is trained with less number of guest-crashing and infinite-running samples as compared to benign samples. We have observed an overall testing accuracy of 96.12%.

Table 4.2 shows our testing results with the unknown test set. We have supplied this set to verify the correctness of proposed model with unknown unlabeled instances of binaries. From statistics, we can deduce that our model is capable in categorizing the unknown instances. We have confirmed the assigned behavior labels of unknown test samples by executing and monitoring these samples in Ether. With an error rate of 4.6%, we have found that designed model automates our manual behavior labeling process. This proves that the proposed model can determine the environment-reactive behavior of unknown instances also.

TABLE 4.2: Detection accuracy with unknown test dataset

Behavior Class	**# Samples**	**# Correct Instances**	**# Incorrect Instances**
Clean	265	258	7
Malignant	103	102	1
Guest-crashing	36	34	2
Infinite-running	19	17	2

4.3.5 Comparison with Existing Approaches

Table 4.3 shows the comparison of our proposed approach with existing malware detection model. It is clearly observed that our model detected malware with higher accuracy as compared to existing methods. The approaches shown in the table use various machine learning techniques to detect malware. Our approach utilizes the multilayer perceptron algorithm that has the ability of learning non-linear discriminant function and recognizing patterns in high-dimensional feature space. Also, proposed approach addresses the environment-reactive behavior of malware.

TABLE 4.3: Comparison with existing approaches

Approach	**Samples (B & W)**	**Technique Applied**	**Detection Accuracy(%)**
Sharma *et. al.* [91]	50 & 50	SVM	50
Xun *et. al.* [93]	722 & 1589	Naive-Bayes	95
Stopel *et. al.* [121]	1512	ANN	90
Nissim *et. al.* [122]	–	SVM	94
Our approach	1150 & 1120	NN	96

4.4 Performance Evaluation

To evaluate the performance of the proposed supervised learning model we considered following metrics.

4.4.1 Detection Accuracy

The detection rate determines the accuracy of a proposed model. The proposed model addresses the malware detection problem that categorizes the malware instances according to their suspicious (Non-environment-reactive and environment-reactive) behavior. The high true positive rate and low false positive rate is desirable in case of any malware detection problem. We obtained high TPR and low FPR in Table 4.1.

4.4.2 Evaluation with Other Dataset

This metric determines how well the trained model performs on any data other than training data. For instance, the FNR value with both training and test samples tables are more as compared to FPR. The reasons are that i) our constructed input vector includes such patterns that can identify the diversity in all four behavior classes and ii) our dataset is not imbalanced. To check whether the dataset is balanced or not, we have applied an imbalance measure derived in [118]. Let Ω be an imbalance measure for OAA. Equation 4.6 decides the value of Ω. Here K=4, denotes the number of classes and n_i is the total number of samples in i^{th} class. If Ω tends to zero means the dataset is imbalanced and the maximum value for Ω will be $1/(K-1)$ that denotes that the dataset is balanced. In our case, Ω is 0.15. It indicates that our training set is not completely imbalanced and thus we have achieved uniformity in training and testing results.

$$\Omega = \min_{i=1,2\cdots,K} \left[\frac{n_i}{\sum_{j=1,\ i\neq j}^{k} n_j} \right] \tag{4.6}$$

4.4.3 System Overhead

We created neural networks equal to the number of behaviors to be detected, which in our case is 4. In order to achieve a high efficiency with respect to training time, we trained all these four networks in parallel by making use of JAVA threads. Hence, training of each network occurs in parallel and the overall training time becomes equal to the maximum value of the time taken by all four networks. We obtained training time in the range of 17 to 1096 seconds for 100 to 10000 iterations respectively with heap space 3GB. We also trained the designed model with heap space 2.5GB (training time ranges from 22 to 1536 seconds) and observed a significant speed up (1.45 for 5000 iterations) with the former. This speedup is achieved due to the lower execution frequency of the JAVA garbage collector in higher heap space. The obtained training time is not a big issue to be considered. But when our proposed method applied with larger datasets this time will increase drastically. To reduce the training time with the larger dataset we can increase the heap space memory of the system and can acquire a significant speed up without making any modifications into our developed model.

4.5 Summary

In conclusion, we can argue that our proposed approach using system-call traces can be used to identify malware having environment-reactive behavior. We evaluated the proposed approach known and unknown samples. The unknown samples were not labeled during behavior monitoring phase. Our decision model predicted the class of these unknown samples that are again executed in Ether for verifying the class predicted. With an error rate of 4.6%, we have found that designed model automates our manual behavior labeling process. We have observed this error rate because the classification is done within a malware class therefore, we have expected this false-alarm.

The proposed model also differentiates malware from benign programs. We have achieved an overall of $\sim$96% of detection accuracy. The false-alarm rate in these cases is negligible which indicates that our selected input vector is significantly

diverse in identifying samples with clean and malignant behavior. The formed input vector is indicative of capturing ordered relationship among system-calls invoked. Furthermore, we have built a multi-layer neural network that assures learning and recognizing patterns in high-dimensional feature space.

The proposed approach is based on system-call traces. To hamper our approach, malware writers can apply system-call injection attack during execution by inserting irrelevant system-calls. This attack will change the ordered sequence of calls as well as the states in input vector. Therefore, in next chapter we present an approach that is resilient to system-call injection attack.

Chapter 5

Program Semantics for Malware Detection

As discussed earlier, the current malicious threats are embedded with numerous anti-detection features to evade dynamic malware detection techniques. In the previous chapter, we addressed the malware carrying environment-reactive behavior. We observed that these malware programs sense the presence of virtual environment and exhibit a reactive, often benign, behavior in response. We exploited their reactive behavior and identified them. For this, we utilized the sequence of system-calls invoked during execution of malware and benign programs. The majority of dynamic behavior detectors rely on system-calls to model the infection and propagation dynamics of malware. However, these approaches do not account an important anti-detection feature of the modern malware, *i.e.*, system-call injection attack.

In this chapter, we present an approach that is resistant to the system-call injection attack. This attack allows the malicious binaries to inject irrelevant and independent system–calls during the program execution thus modifying the execution sequences and defeating the existing system–call based detection. To develop an approach that is not vulnerable to the system-call injection attack, we characterize program semantics in terms of semantically-relevant paths. These paths are

extracted from a dependency graph that is modelled using system-call traces. Our proposed approach specifies the program semantics using Asymptotic Equipartition Property (AEP) mainly applied in information theoretic domain. The AEP allows us to extract the information-rich call sequences which are further quantified to detect the malicious binaries.

We experimentally demonstrate that the proposed approach is resilient to call-injection attacks. We show that even with thousands of call-injection, our approach sustains and performs in a similar way. Moreover, we observe that our approach is comparable to other existing behavior-based malware detection approaches.

5.1 Introduction

The majority of dynamic behavior-based malware detectors [61, 80, 81, 101] make use of system-calls as these are only available gateways for an application's interaction with Operating System (OS). A system call is an interface between a user-level application and kernel-level services. These services include hardware, input-output related activities, creation/deletion of processes and many more. To infect the host system, malware needs to invoke a sequence of system-calls as these are nonbypassable. Therefore, capturing malware by employing system-calls will allow devising a reliable detection solution. The present malware programs are, however, equipped with advanced anti-detection techniques which can evade even system-call based malware detectors [123]. To counter system-call based approaches, malware authors make use of shadow attacks [124] and system-call injection attacks [125], discussed in following paragraphs.

5.1.1 Shadow Attack

Shadow attack was first demonstrated by Ma *et al.* [124]. The authors in their approach have shown that the critical system-call sequences of malware can be divided and exported into separate shadow processes. The functionality of malware remains unchanged when its system-call sequences are exported to shadow

processes. Examining an individual process shall not be able to detect the maliciousness. Reliable detection required that all shadow processes be identified and their execution sequence be known to extract the run-time trace of system-calls. Figure 5.1 illustrates the shadow attack.

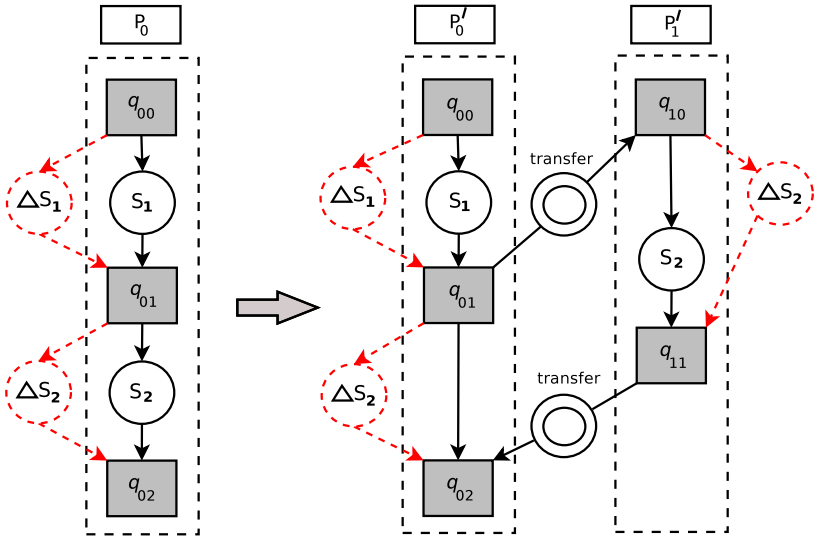


FIGURE 5.1: An illustration of shadow attacks [124].

In this figure, process P_0 is exported into two shadow processes, P'_0 and P'_1. Every q_{ij} denotes the state of process P_i and S_i denotes system-call. To achieve the functionality of P_0, the same input parameters are sent from P'_0 to P'_1; and return values are passed from P'_1 to P'_0. The shadow processes individually act in benign manner and collectively these depict malicious behavior. These shadow processes communicate with rewritten malicious code to deliver their malicious payload.

5.1.2 System-call Injection Attack

The system-call injection attacks are deployed by inserting irrelevant and independent calls in the actual execution flow of malware binaries. By doing so, detection approaches based on graph matching or path similarity analysis are defeated. These attacks are the variant of code-injection attacks [125, 126]. These attacks modify the control-flow of running application such that sequence of system-calls looks benign to malware detection system.

To implement these attacks, the injection code needs to be stored in application's address space. Moreover, this code must invoke a reasonable amount of system-calls to alter the execution-path of running application. A common place for injection code is a program's runtime stack [127]. In [128], Kruegel *et al.* proposed a method, in which memory locations (and registers) are corrupted to transfer control back to running application from stack. Parampalli *et al.* [129] discussed following locations where injection code may reside and work properly.

- ***Stack space:*** If running program has very limited stack space requirement, then remaining stack space can be utilized for injection code. While using injection code from stack space, special handling is required, which ensures that each page fault handled properly. Using page faults, the control is returned to program's code and, therefore, running program can continue its execution.
- ***Global Buffers:*** Buffers are used to store data of programs. The buffer size is relatively larger than the data stored. Therefore, buffers can also be used for call-injection attack. There are certain global arrays containing rarely-used data (code masqueraded as data) than can be replaced without modifying program behavior.
- ***Heap:*** If running program has larger memory requirements, then heap-allocated buffers can be used in place of global buffers. These buffers also hold rarely-used data (code masqueraded as data) that can be overwritten.

5.1.3 Feasibility of Attacks

In literature, researchers have shown the feasibility of system-call injection attacks [125–127, 129]. Shadow attacks have been theoretically proved by [124, 130]. The shadow attacks suffer from the following limitations that restrict its applicability in practice.

1. The shadow attacks lead to multi-process malware, which is slower than the original single process malware. Such a malware, cannot be used in various real time attack situations (such as chain attacks) [131].

2. The implementation of these attacks requires division of malware; the communication between multiple processes; the bootstrap, and the execution sequence of multiple processes. Failure of any shadow process will result into the failure of entire process [130].

The aforementioned challenges limit the employability of the shadow attacks by malware writers. On the other hand, the system-call injection attacks are free of these limitations, and, therefore, to earn more revenue, malware authors would prefer these attacks. Taking this fact into consideration, we present an effective system-call based malware detection approach that is resistant against system-call injection attacks.

5.2 Proposed Approach

To evade detection, malware is continuously being evolved and equipped with anti-detection techniques such as code obfuscation, polymorphism, metamorphism, anti-debugging, anti-VM, code-injection, to name a few. By incorporating these techniques into malicious code, malware authors try to extend the lifetime of malicious code and hide its malicious intent. The anti-detection techniques have instigated a never-ending arms-race between malware detectors and malware authors.

In this chapter, we propose a novel detection technique for identifying detection-aware malicious threats. Our proposed approach employs the program semantics to identify the information-rich components of malware and benign files. The complexity of sophisticated malware codes makes difficult to detect and analyze them. These programs can be found in multiple statically diverse forms having the same functionality. Such metamorphic forms can be detected only by behavior-based detection methods to analyze the behavior of a malware program we need to understand its semantics instead of the syntax.

The semantics (behavior) of any program can be explored by exploiting its execution-flow. During execution, the malicious programs try to infect host machine with actual malicious payload (if it is not environment-aware [112, 113, 132]

or having trigger-based behavior [133]). Our prime objective is to define a metric that can quantify the program semantics. For this, we consider AEP that is based on Shannon's entropy [134] that is a measure of information present in a program object [135] and remains unchanged when one-to-one function is applied [136]. Using entropy, we extract semantically-relevant call sequences and quantify them to construct feature space of proposed model.

Our notion of characterizing program semantics is not vulnerable to call-injection attacks or behavior obfuscation as the discriminating components are composed of 1) multiple call sequences, and 2) non-string based statistical features. Malware programs with different call sequences may exhibit similar behavior. For example, self-replication behavior of malware in which it copies its content to either a new file or into an already existing file, can be represented in one of the following system-call based path sequences:

1. `NtCreateFile`→`NtOpenFile`→`NtReadFile`→`NtWriteFile`,
2. `NtOpenFile`→`NtCreateFile`→`NtReadFile`→`NtWriteFile`, and
3. `NtCreateSection`→ `NtMapViewOfSection`→`NtCreateFile`→`NtSetInformationFile`→`NtWriteFile`.

A behavior is manifested as a path in execution of system-calls and may be composed of many edges; each edge encapsulating requisite transition from one call to another. Same behavior can be captured through multiple such paths in an execution trace. To determine the most likely path, we use its entropy as an information measure. In our method, to capture these paths exhibiting similar behavior, we represent the program behavior through multiple paths carrying almost the same information quotient. These paths are statistically mapped to a non-string based feature space to avoid string-based evasions [1]. The problem statement is composed of sub-problems listed as follows:

1. Encapsulating program behavior into semantically-relevant paths through AEP concept and extract them via ALBF.

2. Verify and validate the specified behavior by constructing a learning-based detection model.

The proposed approach enables us to transform the input binary programs into two forms; one that characterizes the most relevant information; and the other that exploits this relevant information to construct a detection model and thus verifies the effectiveness of extracted behavior.

5.2.1 Encapsulating Program Behavior

In this phase, execution traces of binaries are transformed into Ordered System-Call Graph (OSCG) derived from the sequence of invoked system-calls. A vertex of OSCG corresponds to a system-call in program trace. An edge from vertex u to vertex v of OSCG corresponds to the occurrence of the pair $\langle S_u, S_v \rangle$ in the sequence. Here, S_u and S_v are system-calls corresponding to vertices u and v respectively. The graph preserves order in which these calls are invoked. So, a pair $\langle S_1, S_2 \rangle$ shall add an edge from vertex 1 to 2, whereas $\langle S_2, S_1 \rangle$ shall add an edge from vertex 2 to 1. An OSCG is constructed for each input binary. Consider the following trace of system calls $S_1 S_3 S_2$. This trace has only two pairs $\langle S_1, S_3 \rangle$ and $\langle S_3, S_2 \rangle$.

In order to specify program behavior, the OSCGs are used to determine all reachable paths from initial node (the first call invoked) to the final node (last call invoked) of the sample. We apply AEP on each path to check if it is semantically-relevant or not. The detailed description is given in subsequent paragraphs.

5.2.1.1 Transforming Program Binaries as OSCG

To transform binaries into ordered system-call graph (OSCG), each binary is executed in a virtualized environment. In particular, we have employed Ether for executing binaries [46]. Execution of binaries is monitored, and invoked system-calls are logged. Execution of binaries using Ether is performed in similar fashion as discussed in Chapter 3 and Appendix A.

The acquired traces are used in extracting the sequence of invoked system-calls. Consider an execution trace $\boldsymbol{\xi} = \{S_1, S_2, S_3, S_1, S_2, S_2, S_3, S_3, S_2\}$. This trace has three distinct system-calls S_1, S_2 and S_3. So, we construct OSCG with three nodes. As the sequence has pairs $\langle S_1, S_2\rangle, \langle S_2, S_3\rangle, \langle S_3, S_1\rangle$ and $\langle S_3, S_2\rangle$, edges are added from node 1 to 2, node 2 to 3, node 3 to 1, and node 3 to 2.

Graph-based representation such as OSCG, also, captures the sequential nature of the data [66]. Representing execution traces in the form of directed labeled graph is not new. In the past, many approaches have used graph-based representations to detect malicious files [60, 61, 80, 81]. In OSCG, we ignore all the system-call parameters to avoid the sensitivity towards handles, arguments and other system artifacts.

We shall be using $\boldsymbol{\xi}$ to represent execution trace of a sample and $\mathbb{S}$ to represent the set of all possible (distinct) system-calls. In our case $|\mathbb{S}| = 284$, *i.e.*, $\mathbb{S} = \{S_1, S_2, \cdots, S_{284}\}$ as only 284 possible system-calls can be invoked on Windows XP (SP2) [102]. Each call in $\mathbb{S}$ performs a service at the kernel level that is requested by running binary. For instance, routine `NtMapViewOfSection` is only invoked to map the view of a section into the virtual address space of running process, `NtWriteFile` is called to write into a file and `NtClose` is the routine invoked to close the handles created by other routines.

Successive calls to these routines collectively depict program behavior. To characterize the program behavior through OSCG, we preserve the order in which system-calls are invoked.

Definition 5.1. An Ordered System-Call Graph (OSCG) $\mathbb{G} = (\mathbb{S}, \mathbb{E})$ is a directed graph, where $\mathbb{S}$ is the set of vertices and each vertex represents a system-call. $\mathbb{E} = \{E_{ij} | S_i \xrightarrow{\rho_{ij}} S_j;\ S_i, S_j \in \mathbb{S}\}$, where ρ_{ij} denotes the transition probability from system-call S_i to system-call S_j.

It is assumed that paths in the graph $\mathbb{G}$ are Markov chains, *i.e.*, the future state depends on the present state only and not on past states. The transition probability

ρ_{ij} is computed as follows.

$$\rho_{ij} = \frac{\text{count}(S_i \rightarrow S_j)}{\sum_{k=1}^{284} \text{count}(S_i \rightarrow S_k)} \tag{5.1}$$

where, $S_i \rightarrow S_j$ represents a transition from S_i to S_j. As discussed earlier, the paths of the graph $\mathbb{G}$ are Markov chains. Therefore, the computed transition probability must satisfy Markov property [137] as given in Equation 5.2.

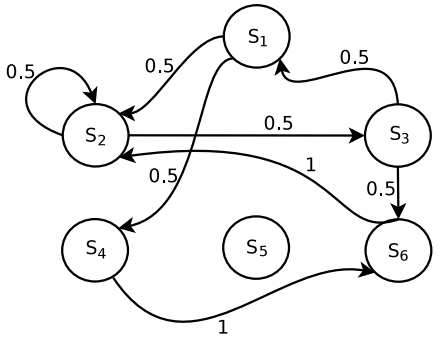
$$\forall_i \quad \sum_{j=1}^{284} \rho_{ij} = \begin{cases} 0 & S_i \text{ is isolated node} \\ 1 & \text{otherwise} \end{cases} \tag{5.2}$$

For example, consider the execution trace $\boldsymbol{\xi} = \{S_1, S_2, S_3, S_1, S_4, S_6, S_2, S_2, S_3, S_6\}$ of a program $\mathcal{P}$. For $\mathbb{S} = \{S_1, S_2, \cdots, S_5, S_6\}$, Figure 5.2 shows the corresponding graph $\mathbb{G}$ and matrix for this example. In $\mathcal{P}$, the set of distinct system-calls invoked is $\{S_1, S_2, S_3, S_4, S_6\}$. The system-call S_5 is an isolated node as it is not invoked during execution of $\mathcal{P}$. Edges are directed and labeled with transition probability ρ_{ij}. For instance, in the execution trace $\boldsymbol{\xi}$, two transitions $S_3 \rightarrow S_1$ and $S_3 \rightarrow S_6$ occurred from node S_3, therefore, both the edges are labeled with equal probability, *i.e.*, 0.5.

Figure 5.2 also shows a 6×6 square matrix called transition probability matrix (TPM) for $\mathcal{P}$. Every row in TPM adds either to 1 or to 0. TPM in our case is 284×284 as $|\mathbb{S}|$ for Windows XP (SP2) is 284. As discussed, there are two transitions from node S_3, *i.e.*, $S_3 \rightarrow S_1$ and $S_3 \rightarrow S_6$. Therefore, both the entries ([3,1] and [3,6]) in TPM are filled with equal probability *i.e.*, 0.5.

5.2.1.2 Specifying Program Behavior

The proposed approach is based on semantically-relevant paths. This concept is inherited from information theoretic model that was introduced by Cui *et al.* [135] in the domain of software testing. According to AEP, "for a random process there exists few paths that carry much more information than the other paths of the graph" [136]. The authors have proved this concept and named these paths as



	S_1	S_2	S_3	S_4	S_5	S_6
S_1	0	0.5	0	0.5	0	0
S_2	0	0.5	0.5	0	0	0
S_3	0.5	0	0	0	0	0.5
S_4	0	0	0	0	0	1
S_5	0	0	0	0	0	0
S_6	0	1	0	0	0	0

FIGURE 5.2: An example of Ordered System-Call Graph (OSCG) and Transition Probability Matrix (TPM).

'typical paths'. In literature, AEP has been applied successfully to the identically independent distributed processes and Markov chains [136].

In this chapter, we also follow the same concept and hypothesize that there exist paths that are more probable than the other paths of OSCG. A path $\mathbb{P}$ of a graph $\mathbb{G}$ is defined as follows.

Definition 5.2. A path $\mathbb{P} = \{S_1, S_2, \cdots S_n\}$ is an alternate sequence of nodes and edges of $\mathbb{G}$ which starts from S_1 and ends at S_n.

Here, S_1 denotes S_{start} and S_n represents S_{end}. S_{start} is the first system-call invoked and S_{end} is the last system-call invoked during execution of a program $\mathcal{P}$. Each link in a path is expressed by its transition from one system-call to the other. For any path between two nodes of OSCG, path probability is computed from transition probability of its constituent links.

The path probability $Pr(\mathbb{P})$ of a path $\mathbb{P}$ is given by $Pr(\mathbb{P}) = Pr(S_1).Pr(S_n = S_n|S_{n-1} = S_{n-1}, \cdots, S_1 = S_1) = Pr(S_1) \cdots Pr(S_n = S_n|S_{n-1} = S_{n-1})$. The $Pr(S_1)$ is the initial probability of node S_1. The initial probability of a node S_i is the probability of occurrence of S_i among all the system-calls invoked in the execution trace $\boldsymbol{\xi}$. Equation 5.3 gives the initial probability of node S_i, *i.e.*, $Pr(S_i)$. $|S_i|$ is the total occurrence of node S_i in the system-call trace and $|\mathbb{S}|$ is the number of distinct calls invoked during execution. Paths containing links with high transition probability are likely to contribute more to the semantic quotient.

$$Pr(S_i) = \frac{|S_i|}{|\mathbb{S}|} \tag{5.3}$$

We aimed at computing all the paths originating from S_{start} to S_{end} nodes of formed OSCG for extracting semantically-relevant paths of a program $\mathcal{P}$. Computing all-paths between two nodes is an NP-complete problem [138]. To resolve this, we approximate this phase by extracting candidate paths instead of all paths. In order to determine if a path is semantically-relevant, we apply AEP on each candidate path $\mathbb{P}$ of the sample. For this, we first determine the maximal entropy rate λ^* of the binary program under consideration as follows [135].

$$\lambda^* = \max\left\{\lim_{n\to\infty}\frac{\log(T_n)}{n}\right\}, \tag{5.4}$$

Here, T_n is the total number of paths of length n in $\mathbb{G}$. Using λ^*, we extract the semantically-relevant paths that are richer in information than other paths. Now, in order to define ϵ-semantically-relevant paths with $\epsilon > 0$, we apply following two properties (Equation 5.5 and Equation 5.6) on each path $\mathbb{P}$:

$$\left|\tfrac{1}{n}\log\tfrac{1}{\mathsf{Pr}(S_1,S_2,\cdots,S_n)} - \lambda^*\right| < \epsilon, \qquad Property\ 1 \tag{5.5}$$

$$\frac{\log B(S_1,S_2,\cdots,S_n)}{n-1} > \frac{1}{2}(\lambda^* - \epsilon), \qquad Property\ 2 \tag{5.6}$$

Where $B(S_1,S_2,\cdots,S_n) = \prod_{1\le i\le n} b(S_i)$ and $b(S_i)$ is the branching factor of the node S_i. The left hand side (LHS) of Equation 5.6 is average logarithmic branching factor used for constructing our feature space. We select ALBF metric of each path to construct our feature space as the branching factor is a good indicator of semantic relatedness [139]. Now, we can define ϵ-semantically-relevant paths as follows: (Definition 5.3).

Definition 5.3. A path $\mathbb{P} = \{S_1, S_2, \cdots, S_n\}$ is ϵ-semantically-relevant if it satisfies *Property* 1 and *Property* 2 (Equations 5.5 and 5.6).

We use $\mathbb{T}(\epsilon)$ to denote ϵ-relevant set, a set of all ϵ-semantically-relevant paths of the program $\mathcal{P}$. The paths in $\mathbb{T}(\epsilon)$ vary according to the value of ϵ. If $\epsilon_1 < \epsilon_2 \cdots < \epsilon_k$, $\mathbb{T}(\epsilon_1) \subseteq \mathbb{T}(\epsilon_2) \cdots \subseteq \mathbb{T}(\epsilon_k)$.

A very small value of ϵ may not capture all paths needed to encapsulate information content whereas a high value of ϵ may include irrelevant/redundant path lowering the information content of $\mathbb{T}$. We have kept the value of ϵ ranging from 0.5 to 7.5. The upper limit of ϵ is the maximum value (7.59) of *Property* 1 (LHS) in all paths of malware datasets. The initial non-zero value of ϵ taken as 0.5 in our relevant set selection approach. With respect to each value of ϵ, we train our model with features relevant for specifying the malicious behavior.

Cui *et al.* in [135], have proved two theorems, which ensure that typical (semantically-relevant) paths carry relevant information of the graph. The theorems are stated as follows:

Theorem 5.4. *Let $\epsilon > 0$. The ϵ-typical paths take probability 1, asymptotically; i.e.,*

$$\limsup_{n\to\infty} \mathsf{Pr}\left(\left|\frac{1}{n}\log\frac{1}{\mathsf{Pr}(S_1,S_2,\cdots,S_n)} - \lambda^*\right| < \epsilon\right) = 1.$$

Theorem 5.5. *Let $\epsilon > 0$. For any path $\mathbb{P}$ of $\mathbb{G}$ achieves λ^*,*

$$\limsup_{n\to\infty} \mathsf{Pr}\left(\frac{logB(S_1,S_2,\cdots,S_n)}{n-1} > \frac{1}{2}(\lambda^* - \epsilon)\right) = 1.$$

Theorems 5.4 and 5.5 have been proved by employing limsup definition of entropy rate instead of limit definition. Proving AEP with the limit definition, as in Shannon-Mcmillan-Breiman theorem [136], is difficult as it requires a strong side condition (ergodicity). In the present context, malware does not constitute same behavior averaged over time and does not exhibit ergodicity. Therefore, we can also consider the limsup definition. Assuming the theorems and proofs are valid, we apply their concept of typical paths towards semantically-relevant paths in our approach.

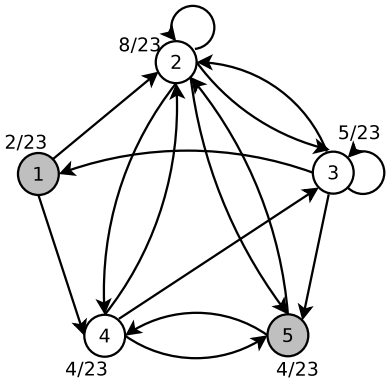
The set of semantically-relevant paths is not unique as a given program may have multiple execution traces due to the presence of conditional constructs (triggering of different constructs can invoke different executions). Any execution trace that results in invocation of malicious activity should suffice to extract the semantically-relevant paths capturing malicious behavior. We have constructed OSCG from a

single execution trace as exploring more execution traces shall add to monitoring overhead.

5.2.1.3 Semantically-relevant Path Extraction

Consider TPM and graph $\mathbb{G}$ as shown in Figure 5.3. In the example, S_{start} is the node 1 and S_{end} is the node 5. There are nine cycle-free paths from node 1 to 5. We determine the value of λ^* considering all possible path lengths of 2, 3, and 4 as computed by the application of Equation 5.5 (refer Table 5.1). These values range from 1.85 to 2.765 so we can select the values for ϵ in the specified range. With ϵ=2.6, we get $\mathbb{P}_2$, $\mathbb{P}_3$, $\mathbb{P}_6$, $\mathbb{P}_7$, and $\mathbb{P}_8$ as candidates for semantically-relevant paths.

Selecting the different values of ϵ and applying Equation 5.6 will give us different sets of semantically-relevant paths. For instance, if we apply Equation 5.6 with $\epsilon = 2.1$ then both the paths $\mathbb{P}_7$ and $\mathbb{P}_8$ are included in semantically-relevant set. With these different sets of paths, we train our model and observe the detection accuracy.



	1	2	3	4	5
1	0	1/2	0	1/2	0
2	0	2/8	3/8	1/8	2/8
3	1/5	2/5	1/5	0	1/5
4	0	2/4	1/4	0	1/4
5	0	1/3	0	2/3	0

FIGURE 5.3: Ordered System-Call Graph (OSCG) and Transition Probability Matrix (TPM).

5.2.2 Verifying and Learning Malware Detection

This section presents our learning-based model that discriminates benign and malware programs. For our proposed learning-based detection model, we construct feature space $\mathbb{F}$ by utilizing extracted semantically-relevant paths. We have used

TABLE 5.1: Paths from node 1 to 5 showing values w.r.t. Equation (5.5).

Paths	**Probability of the paths**	**Values of Property 1**
$\mathbb{P}_1$: 1–2–5	$\mathsf{Pr}(\mathbb{P}_1) : 0.0108$	2.765
$\mathbb{P}_2$: 1–2–3–5	$\mathsf{Pr}(\mathbb{P}_2) : 0.0032$	2.100
$\mathbb{P}_3$: 1–2–4–5	$\mathsf{Pr}(\mathbb{P}_3) : 0.0013$	2.530
$\mathbb{P}_4$: 1–2–4–3–5	$\mathsf{Pr}(\mathbb{P}_4) : 0.0002$	2.675
$\mathbb{P}_5$: 1–4–2–3–5	$\mathsf{Pr}(\mathbb{P}_5) : 0.0016$	2.925
$\mathbb{P}_6$: 1–4–3–5	$\mathsf{Pr}(\mathbb{P}_6) : 0.0021$	2.330
$\mathbb{P}_7$: 1–4–2–5	$\mathsf{Pr}(\mathbb{P}_7) : 0.0054$	1.850
$\mathbb{P}_8$: 1–4–3–2–5	$\mathsf{Pr}(\mathbb{P}_8) : 0.0010$	2.095
$\mathbb{P}_9$: 1–4–5	$\mathsf{Pr}(\mathbb{P}_9) : 0.0108$	2.765

'histogram binning' technique [140] (mainly applied in the fields of information retrieval, image processing and text processing) as it incorporates approximate matching and reduces sensitivity to slight changes in system-call sequences. This avoids the possibility of evasion encountered due to detection-aware malware.

ALBF metric has been employed to determine the bin to which a semantically relevant path belongs to. In our case, each bin corresponds to a range of ALBF values. These bins are spaced at uniform intervals and hold the frequency count of respective semantically-relevant paths. These bins are considered as features. For example, if feature space consists of three bins b_1, b_2, b_3 and for program $\mathcal{P}_1$, respective bin frequency counts are f_1, f_2, f_3, its feature vector shall be $\langle f_1, f_2, f_3 \rangle$.

Selecting appropriate number of bins for building our feature space involves a tradeoff between less detailed features (small number of bins imply coarser granularity and loss of information) and overly detailed features (too many bins result in loss of generalization and flexibility).

We determine maximum ALBF value corresponding to malicious binaries. Dividing this by bin size yields number of bins. We constructed and evaluated feature space with bin sizes of 1, 5, 10 and 15 and observed the detection accuracy. The initial results indicate that the bins formed with an interval range of 5 (bin size is five) identifies benign and malware samples more accurately. The feature vector containing bins with higher intervals merges the relevant paths of different ALBF values and may result in information loss. This merging also reduces the number

of elements in a feature vector. Therefore, we set the interval of 5 and consider 310 non-overlapping bins as features into our feature vector.

The constructed Feature Vector Table (FVT) is trained using the learning algorithms. We use an ensemble-based learning algorithm, *i.e.*, Random Forest, [141, 142], for differentiating malware and benign samples. It is a collection of many decision trees that contribute towards the classification of instances. Also, it provides a better generalization of information even in the presence of noise. It is primarily used when the data set is very large. The decision tree is constructed by randomly selecting subsets of the training data and attributes that can partition the available dataset.

5.3 Experimental Setup and Results

The experiments are performed on Intel Core i3 2.40 GHz with 2.8 GiB RAM, running on Ubuntu 12.04 operating system. For capturing the system-call traces we use Xen hypervisor [143] and create a virtual environment using Ether. The underlying guest OS in Ether is Windows XP (SP2). Therefore, we have built our prototype model by executing binaries in Windows XP. Although, Microsoft abandoned its support for XP but still it is a popular OS widely used in various government agencies, banks and in ATMs. As a result, existing recent similar approaches [75, 80, 125, 144] also utilize XP. However, the proposed approach is not specific to particular OS and analysis framework as

1. the target malicious binaries (PE format) affect all Windows platforms, and
2. system-call sequence used in Windows XP is a subset of those utilized in Windows 7 [102].

The proposed approach will perform in a similar fashion if system-call traces are collected with different Windows OS and some other analysis framework (Anubis, Cuckoo, GFISandbox, to name a few). In this section, we present the implementation details and evaluation of our proposed approach. The experiments are carried out using benign and malware executable samples. The proposed approach detects

the malicious Windows PE binaries (PE is the most popular file format among malware authors as reported by the *virustotal.com* [3]).

5.3.1 Experimental Dataset

We utilize real instances of malware and benign samples. The majority of malware detection approaches [75, 76] make use of one malware dataset to evaluate the performance of their approach. These approaches perform well on selected dataset, however, do not generalize well to other datasets and result in performance degradation. Therefore, to evaluate generalization of our approach, we used two different malware datasets and label them as D_{old} (old dataset) and D_{new} (new dataset).

The former dataset consists of 1209 samples. This set also includes samples utilized in [145] for their work of detecting metamorphic malware. The types of samples in this set include packed, polymorphic and metamorphic malware. We selected this dataset for two reasons: 1) to represent the class of malware samples discovered prior to 2012, and 2) to estimate the performance of our method with morphed and packed samples.

The latter set (D_{new}) of samples is downloaded from the malware repository system, *i.e.*, *VirusShare.com*. The mentioned repository system labels each uploaded sample after scanning it with 55 AV scanners. We can rely on the labeling process of *VirusShare.com* as it is akin to comparing with large number of AV scanners. This dataset consists of 1226 malware samples each of which was discovered from January 2013 to March 2014 and it is labeled as 'new'. Both datasets are divided into training (70%) and test (30%) set.

We used one benign dataset that contained total number of 1316 samples. Benign samples are scanned by uploading them to the web portal *VirusTotal.com* to verify their non-maliciousness. Our benign dataset consists of different kinds of software applications such as browsers, games, FileZilla, googletalk setup, iTunes, youtubedownloader, Media players, wireshark, to name a few. We used these benign software programs to evaluate the accuracy of proposed model.

As discussed earlier, we monitored the execution of each benign and malware sample in the controlled environment created using Ether. We observed that during execution there were some malware samples that did not generate any log and few benign and malware samples that cause executional errors. The malware samples embedded with anti-detection features (VM-aware and trigger-based) do not generate any log. The execution errors occurred due to OS compatibility issues. Our final datasets (benign and malware) include the samples that are executed in guest OS without abnormal termination and without execution errors.

To generate system-call logs, each sample is permitted to execute for 10 minutes. According to [146], five minutes is sufficient duration for the execution monitoring. We doubled this execution time to capture the malware equipped with capability of carrying out time-out attacks. We observed that in all samples, benign as well as malicious, the execution sequences are mostly made of 160 different calls out of 284 calls. We refer these calls as 'frequent' calls. Remaining 124 calls are regarded to as 'rare' in following discussions. The experiments are performed with training sets of benign and malware datasets, and the performance evaluation is carried out using test samples.

5.3.2 Approximate All Path Computation

We constructed OSCG for each sample as discussed earlier to determine the paths between S_{start} and S_{end}. Identifying all paths between two nodes of a graph is an NP-complete problem [138]. The time complexity of computing all paths is exponential in case of a complete graph. To reduce the time complexity, we approximate the 'all path computation phase' of our approach. We, first investigate if an OSCG is sparse. We observe the average link-count in OSCGs of benign and malware samples of our datasets. The average link-count of benign samples is 174 and that of malware samples is 282 indicative of the sparse nature of our OSCGs as the number of edges is in $\mathcal{O}(|\mathbb{S}|)$, where $\mathbb{S}$ is the number of vertices (284).

To approximate the all-paths phase, we conducted an experiment with 500 malware and 500 benign samples. The samples are selected in a manner that they

cover the entire range of link-counts. With these samples, we exhaustively computed all paths from S_{start} to S_{end} and computed average number of paths for a given path-length.

Figure 5.4 shows this distribution for both benign and malware samples. As can be seen in Figure 5.4, average number of paths is normally distributed for benign as well as malicious files. However, in path-length ranging from 10 to 31, the average number of paths in malware samples is more than the benign samples. Paths in this range can be used for discriminating a malware from benign. We have used the paths in this particular range as candidate paths for extracting semantically-relevant paths. Instead of computing all paths for all the samples we computed the candidate paths (approximation of all paths). This reduces the path computation time for all the samples.

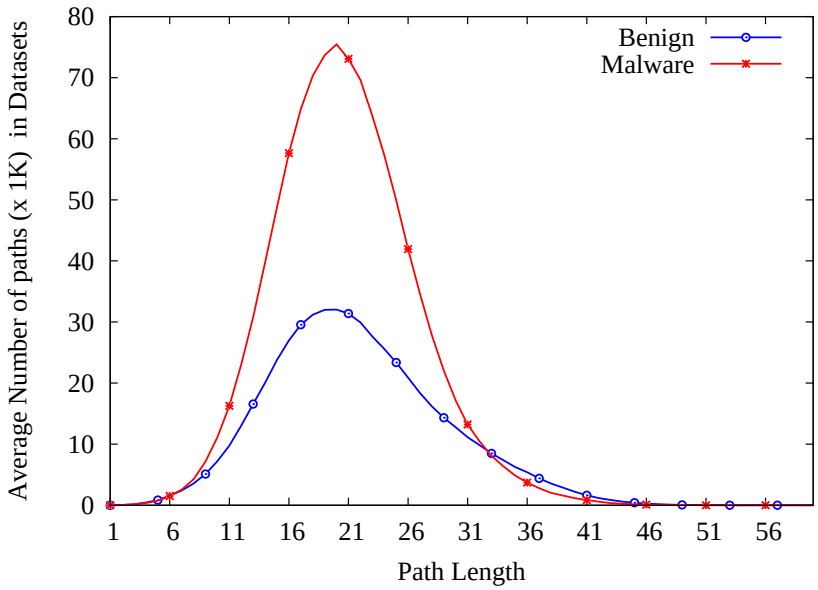


FIGURE 5.4: Path distribution w.r.t. lengths in benign and malware datasets.

Table 5.2 shows the average time consumed in determining all paths and candidate paths. We observed that time taken in computing all paths was higher than the candidate paths. Also, using candidate paths we can reduce the overall computation time. The maximum time consumed was ∼11 hours and ∼9.7 hours for all-path computation and candidate-path computation respectively. We also noticed that some samples took higher processing time than the samples having higher link-count. This indicates that processing time is not directly proportional

to link-count. In the majority of samples (∼88.3%), candidate paths have a link-count of atmost 450 that result into a total time of ∼1.83 hours.

TABLE 5.2: Processing time of all-paths and candidate-paths.

Link-count up to	**Avg. Time of All-paths (in sec.)**	**Avg. Time of candidate-paths (in sec.)**	**% of Samples Covered**
50	0.01	0.005	0.98
51-150	1.78	0.47	6.07
151-250	450.64	162.89	37.89
251-350	9271.65	1145.72	27.32
351-450	19080.20	5292.86	16.08
451-550	24848.32	15565.60	9.84
551-650	33482.56	28384.77	1.33
651-750	39524.45	35109.9	0.49

We determined the candidate-paths for all the samples and, then, extracted the semantically-relevant paths as explained earlier. We constructed $\mathbb{T}(\epsilon)$, ϵ-relevant set of benign and malware samples and feature vectors for all binaries using the frequency distribution of ALBF values of their $\mathbb{T}(\epsilon)$. The constructed feature vector is trained using Random Forest as described earlier. The above process has been repeated for different ϵ values.

5.3.3 Detection Accuracy

We have evaluated the performance of our proposal in terms of popular evaluation metrics [147] – *i.e*, True Positive Rate (TPR), False Positive Rate (FPR), True Negative Rate (TNR) and False Negative Rate (FNR). In the present context, we designate malware class as positive and benign class as negative. TPR (FPR) is the fraction of malware instances correctly (incorrectly) classified. Similarly, TNR (FNR) denotes the fraction of benign instances correctly (incorrectly) classified. For any malware detection model, it is desired that TPR should be high, and FPR and FNR should be low. For two datasets D_{old}, D_{new} considered in our evaluation, Table 5.3 summarizes TPR and TNR for different values of ϵ.

As can be seen from Table 5.3, $\epsilon \in \{2.3, 2.6, 2.9, 3.2\}$ yields higher detection accuracy. It can be easily deduced from the table statistics that our constructed

TABLE 5.3: Detection accuracy of D_{old} and D_{new}.

ϵ	D_{old}		D_{new}		**Overall**
	TPR	**TNR**	**TPR**	**TNR**	**Acc**
0.5	0.3	100.0	1.7	100.0	50.50
1.0	4.5	32.7	18.1	99.2	38.62
1.5	64.0	71.3	44.6	62.7	60.65
2.0	81.8	88.4	77.9	78.1	81.55
2.3	92.3	91.5	88.4	93.3	91.37
2.6	96.2	95.3	94.6	93.8	94.97
2.9	**95.9**	**96.1**	**94.3**	**95.4**	**95.42**
3.2	90.3	94.3	94.7	96.1	93.85
3.5	91.4	95.1	92.3	93.9	93.17
4.0	88.6	92.6	89.1	91.5	90.45
4.5	87.9	88.9	90.3	89.0	89.02
5.5	87.1	89.2	89.3	89.6	88.80
6.5	84.1	88.6	87.1	90.3	87.52
7.5	84.6	87.1	85.0	87.2	85.97

feature space has the ability to discriminate between malware and benign samples. Our model achieves the highest accuracy of 95.42% at $\epsilon = 2.9$. Therefore, we have selected it as a threshold. We conducted this experiment extensively with various values of ϵ to validate our hypothesis that lower values of ϵ exclude some of information-rich paths. This is reflected in poor performance exhibited by initial rows of Table 5.3. Too many paths, as happens at higher values of ϵ, can lead to generalization and, therefore, result into the decrease in detection accuracy. The misclassified instances are shown in Table 5.4.

As can be seen in Table 5.4, our model performs best at selected threshold of ϵ=2.9. For malware classification, FPR and FNR should be low as a high value of FPR shall result in malware being considered benign. A high FNR may prohibit execution of legitimate applications. With D_{old} samples, we achieved 3.9% and 4.6% of FPR and FNR respectively. Similarly, FPR of 4.1% and FNR of 5.7% is obtained with D_{new}.

Some of the malware samples yield only partial logs, and this has contributed towards FPR. During runtime, these samples terminated very quickly and did not reveal their actual payload. In our case, this behavior was observed with samples belonging to `worm.autorun` malware family. The samples of this family try to infect the system by creating `.inf` file on root directory of system. When these

TABLE 5.4: False rate with D_{old} and D_{new}.

	D_{old}		D_{new}	
ϵ	**FNR**	**FPR**	**FNR**	**FPR**
0.5	99.7	0	98.3	0
1	95.5	67.3	81.9	0.8
1.5	36	28.7	55.4	37.3
2.0	18.2	11.6	22.1	21.9
2.3	7.7	8.5	11.6	6.7
2.6	3.8	4.7	5.4	6.2
2.9	**4.1**	**3.9**	**5.7**	**4.6**
3.2	9.7	5.7	5.3	3.9
3.5	8.6	4.9	7.7	6.1
4	11.4	7.4	10.9	8.5
4.5	12.1	11.1	9.7	11
5.5	12.9	10.8	10.7	10.4
6.5	15.9	11.4	12.9	9.7
7.5	15.4	12.9	15	12.8

files detect the presence of virtual environment, they do not reveal their malicious payload and terminate the execution. Hence, these instances were misclassified.

A false negative is observed when a legitimate monitored application shows high similarity with the malicious samples. We found that the system-calls related to memory access, process and thread handling activities were common to malware samples. Any benign application using these calls may show high correlation with malware samples and may be misclassified. This aids to FNR.

To reduce the false alarm rate in our approach, we need to identify and remove the call-transitions that are common to OSCGs of most of the malware and benign samples. For this, we may first extract the common subgraph (using graph isomorphism [148]) from all malware and benign samples and then the edges of this subgraph can be removed from all the OSCGs. However, the false alarm rate in our approach is considerably low when compared to approaches [75, 144], and [55] in which the false alarm rate of 10.9%, 9.8% and 9.7% is observed respectively.

The detection capability of our approach with unknown samples (test samples that are not used in training phase) is evaluated using two test datasets with both the training models prepared with ϵ value as 2.9. We performed testing of our both the test sets. For the first test set, we observed overall detection accuracy

of 94.2% (D_{old}: 94.8%, D_{new}: 94.7%). In case of second test set, we achieved an overall accuracy of 93.4% (D_{old}: 93.7%, D_{new}: 93.1%).

The detection accuracy of test samples is approximately similar to that of our trained model. There is a minor difference in detection accuracy of both the sets and this was expected because the learning-based models always perform better with training samples due to the implicit knowledge about the samples. Similar trends of false detection rate are observed with test samples. Our experimental results indicate that the proposed method is effective in discriminating the benign and malware instances.

5.3.4 Resilient against Dynamic Obfuscation

As discussed earlier, modern malware inserts irrelevant and independent system-calls to evade the system-call based detection approaches that rely on either signature or exact pattern matching. These solutions are evaded by malware authors as these methods directly work on raw system-features such as opcodes, instructions, hexbytes, and *etc.* that can be obfuscated or replaced by equivalent alternate features. The discriminating components are clearly visible and hence tampered by malware writers. On the other hand, our method provides a solution by employing a feature space that is not linearly related to raw system features and hence opaque to malware writers.

To measure the robustness of proposed approach in the presence of system-call injection attack, we performed experiments on two sets of system-calls, *i.e.*, rarely invoked system-calls (*RISC*) and frequently invoked system-calls (*FISC*). The former set consists of calls that are rarely invoked by malware or benign applications in our datasets. As stated earlier, the malware and benign applications mostly utilize 160 calls out of 284. The latter set includes frequent calls invoked by benign and malware applications.

Malware writers attempt to disguise their malware programs as benign. So, the call-trace of malware should be similar to that of a benign program. For this, we have selected a set containing 100 benign programs each having large trace size.

For injection, we considered trace of a randomly selected benign sample from this set. As, for different malware, injection may come from traces of different benign samples.

We inserted system-calls into random locations of the execution trace of randomly selected malware programs. The malware samples considered for this are the test samples of D_{new} dataset as these samples belong to the class of latest malware attacks. The number of calls in these malware traces ranges from 674 to 124652. We inserted total calls that are 10%, 20%, $\cdots$, 100%, 150%, 200% of malware traces. For inserting system-calls, we adopt the strategy followed by authors in [80]. These calls are injected one at a time and at random positions in the malware traces. For both the experiments, we observed the performance of our model with ϵ value of 2.9 and results are shown in Figures 5.5 and 5.6.

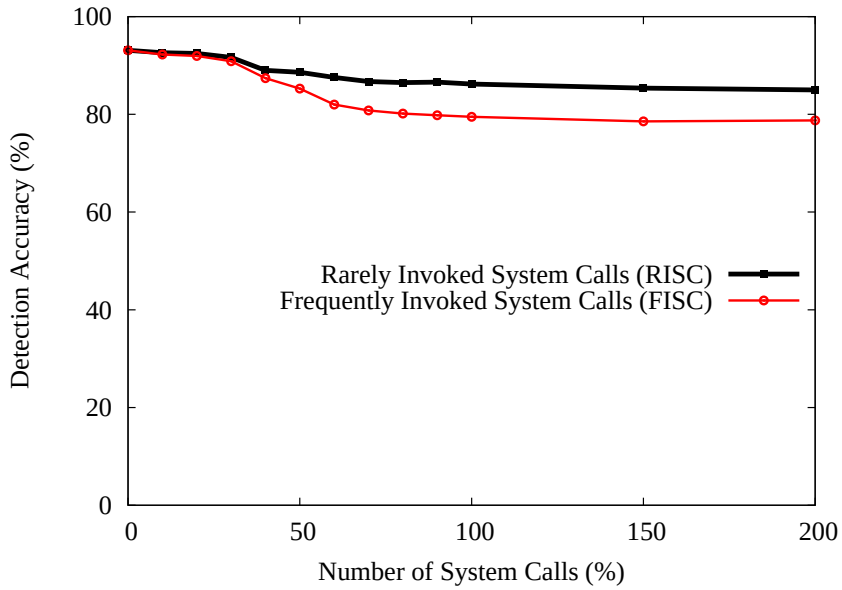


FIGURE 5.5: Detection capability in presence of behavior obfuscation with *RISC* and *FISC*.

Figure 5.5 illustrates the performance of our model in terms of detection accuracy that does not vary up to 30% of call-injection rate. For some malware samples, this translates to injection of ∼30000 calls. We exhaustively injected calls into malware traces and in this way injection also occur in extracted semantically-relevant paths and therefore beyond 30% call injection rate we observed the fall in detection accuracy.

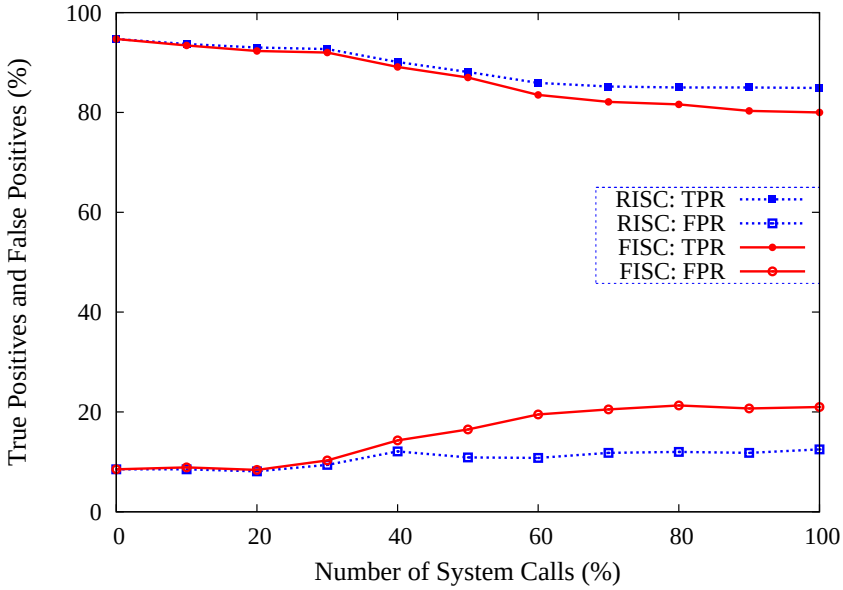


FIGURE 5.6: True positives and false positives with *RISC* and *FISC*.

Figure 5.6 shows TPR and FPR values with respect to both the experiments. It means that the discriminating patterns of proposed approach are not affected by the injected calls. However, when we increase the injection rate, the detection accuracy starts decreasing. This fall in the detection accuracy is expected as after insertion of calls beyond a certain limit, TPM no longer matches the modified samples as more paths are added into semantically-relevant set affecting its frequency distribution. This is expected as insertion of rarely invoked system-calls does not affect TPM as it is akin to adding some transition to almost isolated nodes and such paths are unlikely to be included in semantically-relevant set unless a large number of injection takes place.

With *RISC*, our model performs better when compared to *FISC*. By inserting benign call sequences, we observe an increase in false detection rate of our model. The maximum decrease of 6.9% and 13.6% in the detection accuracy is observed for RISC and FISC experiments respectively. The feasibility of inserting calls of RISC set is more than the FISC as the latter set of calls can affect the prime objective of malware. Therefore, our method shall work without much loss in detection accuracy.

The other important concern of call-injection in our approach is to modify the S_{start} and S_{end}. For this, we closely inspect the variation in ALBF value of paths

after adding two irrelevant calls, *i.e.*, S'_{start} and S'_{end} at initial and last position of malware trace. By doing so, we observed that the semantically-relevant set contains same paths with two additional links, *i.e.*, $S'_{start} \rightarrow S_{start}$ and $S_{end} \rightarrow S'_{end}$ showing single outgoing transition. The ALBF metric (Equation 4) is sensitive to path-length as well as the branching factor. If two additional links are added that were not there in the previous OSCG, then only path-length is affected and increased by 2. The branching factor remains same as the link-count is 1 for both the links. A negligible fall in ALBF is observed due to increase in the path-length. This fall in the majority of cases does not change the bins of those modified paths hence it will not affect our approach. Now, if a long sequence of unrelated calls is added (pre/post) to just increase the path-length then in very few cases it will affect the ALBF as we have restricted the path-lengths (from 10 to 31). To evade the approach, malware authors have to append and prepend a long sequence of unrelated calls with higher outgoing transitions, which modify the bins of all the paths in such a way that increases the false alarm rate.

5.3.5 Comparison with Existing Approaches

Here, we present a comparative evaluation with current state-of-art dynamic malware detection techniques. Moreover, we analyze the impact of call-injection attack on our approach to one proposed by Park *et al.* [80].

Figure 5.7 shows the TPR and FPR of every approach. As can be seen from this figure, our proposed approach is shown to outperform other methods, with the highest true positives and the lowest false positives. The better performance of our approach is due to semantically-relevant paths, which represent the program semantics that cover the most relevant behavior of malware and benign programs.

Park *et al.* [80] proposed a graph clustering method [149] for deriving the common behavior of malware samples. The authors performed an abstraction from system-call traces and used kernel objects [150] to represent malware behavior. Further, they applied graph matching and determined a threshold to assess the detection rate of their approach. Moreover, the authors have built a kernel object behavior graph (KOBG) to exploit the dependency between the kernel objects. The kernel

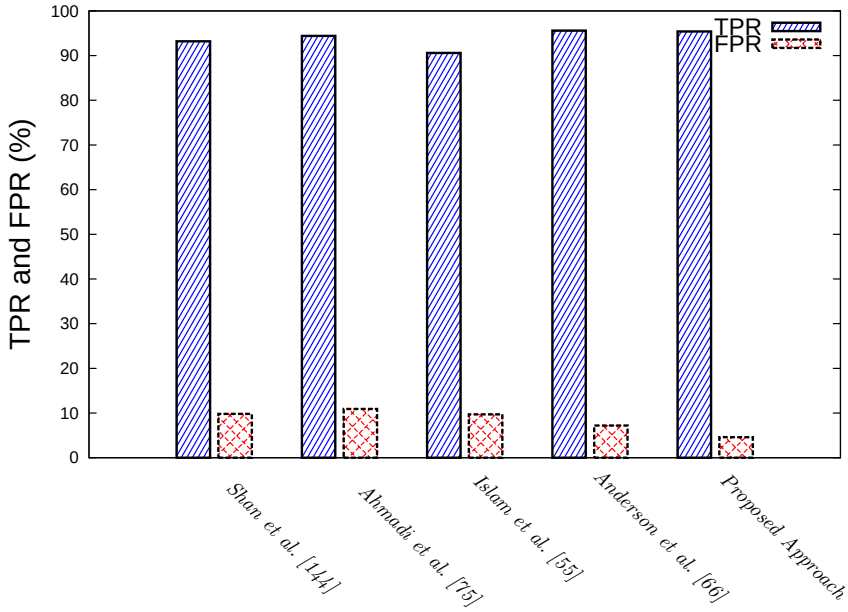


FIGURE 5.7: Comparison with existing malware detection approaches.

objects and their dependencies information are extracted from the system-call traces acquired using Ether framework.

To evaluate Park's approach on our samples, we adapted their approach as mentioned in [80]. We constructed KOBG in similar fashion and built a weighted common behavior graph (WCBG) using McGregor algorithm. To implement the algorithm, we made use of graph `C++` Library provided by the Boost Software [151].

Figure 5.8 contrasts the performance decay of our proposed with the one in [80] for call-injection rate ranging from 0% to 100%. It is quite clear that our approach outperforms the approach in [80]. In our approach, the maximum fall observed is ∼13% while in [80] the observed maximum fall is ∼23%. The detection accuracy of Park's approach with 0% injection rate is observed as 92.45%. The false alarm rates of their approach are 8.2 (FNR) and 6.9 (FPR). The approach by Park *et al.* [80] is based on exact-pattern matching as a result of which the call-injection attack and false alarm rates result into higher performance deterioration. However, our approach is not based on exact pattern matching, but abstracts semantically-relevant paths as bins of branching factor. Therefore, it is less vulnerable to call-injection attack.

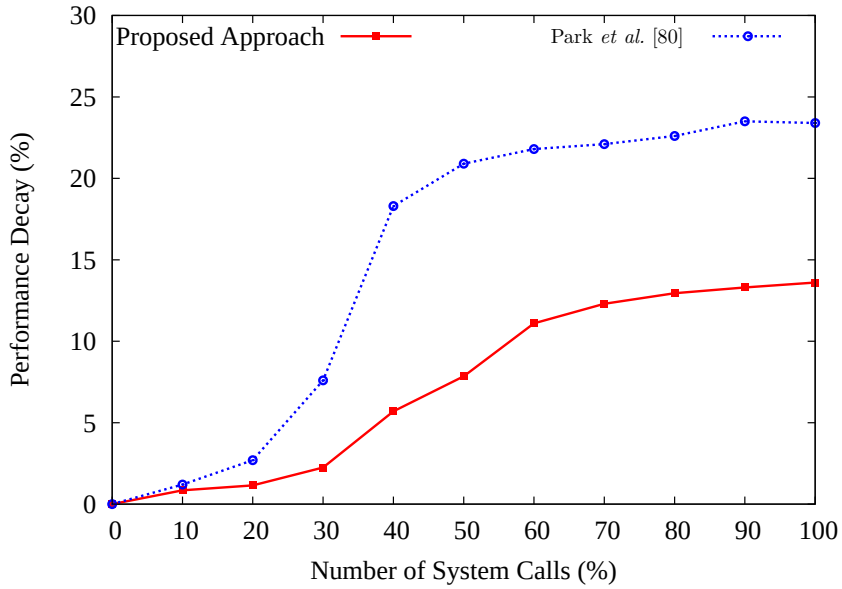


FIGURE 5.8: Comparison of proposed approach and approach in [80].

5.4 Performance Evaluation

In this chapter, we introduced the concept of specifying program semantics in order to discriminate the malicious from non-malicious binaries. To address this, we abstracted the system-calls to a higher level and created sets containing semantically-relevant paths. These semantically-relevant paths cumulatively represent the program semantics since each path sequence exhibits a specific functionality of the program. In this section, we discuss the merits and demerits of proposed approach. For any malware detection approach, it has to address the issues such as evaluation with other datasets, resiliency, stealthiness and associated overhead.

5.4.1 Evaluation with Other Datasets

This performance metric determines the ability of detection model to scale uniformity with 1) comprehensive set of malware samples, and 2) known (training) and unknown (test) malware instances. We used two different datasets that include a wide spectrum of malware samples. The overall detection accuracy with both the datasets D_{old} and D_{new} are observed as 96% and 94.85% respectively. In both cases, there is a marginal difference of 1.15% due to the presence of malware samples in D_{new}, which do not manifest their malicious behavior during runtime.

These samples are termed as detection-aware malware as it senses the presence of instrumented virtual environment.

In the proposed approach, to create the virtual environment, we used Ether. Ether can be detected as the BIOS data strings for Ether make use of emulated variant from Bochs virtual machine. Moreover, the Ethernet card that is emulated by underlying Xen system can be analyzed easily. The detection-aware malware exploits these variations and ensures that it cannot be analyzed. As a result, it generates partial log or no log during runtime. Our datasets do not include the samples with no logs. Therefore, the only concern is the generated partial logs that result into misclassification. Although, in our case the false alarm rates were significantly low as compared to other existing approaches [55, 66, 75, 144]. This particular limitation is common to the majority of dynamic malware detection approaches.

In future, we can substitute Ether with more resilient framework or we may augment more than one framework (emulated, virtualized, and instrumented) to retrieve complete logs of detection-aware malware.

The other factor that assures the performance of our detection model with known and unknown malware samples. We observed the uniformity in our training (95.42%) and testing (93.8%) results. In our case, the difference in training and testing results is only 1.6% which is negligible. Hence, our model is capable of detecting a wide range of malware instances.

5.4.2 Resiliency

Resiliency refers to the robustness of the proposed approach in the presence of possible evasion embedded into malware files. As our proposed model relies on system-call traces, one possible evasion technique to thwart our model is system-call injection attack. Using this attack, malware authors modify the system-call sequences of malware binaries at run time. For incorporating this, the malware authors either make modifications into the malware program or create new binaries through injection of system-calls.

We ran two different experiments to evaluate the robustness of our approach against behavior obfuscation. The experimental results indicate that the detection accuracy remain invariant up to 30% of call injection rate. Beyond this, we observed a fall in detection accuracy that stabilizes above an injection rate of 70%. The system-call sequences of our malware dataset contain on an average more than 10^5 calls. Inserting even 10K, 20K, and 30K independent calls into malware traces does not affect the proposed mechanism. Furthermore, our dataset consists of packed, polymorphic and metamorphic samples, which indicate that the proposed approach can complement existing static malware detection methods.

5.4.3 Stealthiness

Stealthiness refers to the detection capability by which our approach operates with high detection accuracy without disclosing discriminating patterns to malware attackers. The discriminating component of our approach is neither a sequence of system-calls nor a feature space linearly derived from these sequences. The discriminating component of our method is composed of the ranges of ALBF values. As these values are accumulated in bin, our feature space is non-linearly related to sequence of calls.

Multiple semantically-relevant paths imply different subsequence of calls being used in construction of feature space. Modification in one path shall not impact performance of the proposed model. Only large modifications in transitions of all semantically-relevant paths will affect our model. The modification is complex as the attacker needs to identify all semantically-relevant paths and modifying the path sequences in a way that it substantially modifies ALBF bins. Hence, our proposed approach provides stealthiness and is resilient against present and future malicious threats.

5.4.4 System Overhead

In conjunction to its detection accuracy and resiliency against call-injection attacks, we also discuss the associated overheads of the proposed approach. Table 5.5

shows the best, average, and worst time per sample during the main steps of our approach. The total time shown in the Table 5.5 does not include monitoring time as it is common to all behavior-based approaches. Each step is discussed as follows.

TABLE 5.5: Best, average, and worst processing time per sample

Main Steps	**Best Time (in sec.)**	**Average Time (in sec.)**	**Worst Time (in sec.)**
System-Call Monitoring	–	–	600
OSCG Construction	0.001	0.02	1.2
Candidate-Paths Computation	0.005	3355.86	35109.9
Semantically-relevant Paths Extraction	0.07	0.18	0.67
Training Time	1.54	1.54	1.54
Total Time*	1.616	3357.6	35113.31

*Exclusive of monitoring time.

5.4.4.1 System-call Monitoring

Execution tracing of benign and malware binaries in our approach depends on Ether. Therefore, the overheads associated with Ether are inherited into our approach. We fixed the time-out of 10 minutes (600 seconds), therefore we observe this overhead of collecting system-call traces. Monitoring executables from Ether is a time consuming task. Ether uses exceptions whenever a running application makes a system-call to access system services. These exceptions result into significant performance overhead. To reduce this overhead, we can use a faster analysis framework. The proposed approach is not specific to a given monitoring environment and can be generalized by applying the same methodology with other operating systems as well as virtual/sandboxing environments.

To investigate this, we conducted a small experiment using 20 malware samples of D_{new} dataset. We collected execution traces of these samples from Cuckoo sandbox with 10 minutes of timeout. To extract the run-time traces of executables, we submit the sample via `submit.py`. The inline "Cuckoo Agent" (`agent.py`)

receives the executable and analyzes it. After analysis is over, a behavior report is generated which includes logs containing various parameters such as API, system-calls, static attributes, DLL invoked, PE header, and processes created during runtime.

We used the generated behavior report and extracted the system-calls that are invoked. We then created the feature space for these samples in a similar fashion as mentioned earlier and closely observed the variation in frequency distribution of Ether generated traces with Cuckoo generated traces. We found out that there is a negligible variance in the frequency distribution in both the cases.

5.4.4.2 OSCG Construction

In this phase, we extract the system-call sequences from the acquired traces and then build the TPM using the transitions of system-calls. The processing time for TPM construction is negligible (average: 0.02 sec. and worst: 1.2 sec.) as it depends on the trace length. For samples with larger trace-length, OSCG is constructed in few seconds and for average trace length, it is constructed in few milliseconds. We can say that this phase does not lead to higher processing time.

5.4.4.3 Candidate-path Computation

In our approach, we determined the paths between S_{start} to S_{end}. In this quest, we observed that the time complexity for determining all paths is very high. We have shown that our OSCGs are sparse in nature as the average link-count of our samples is less than the total number of nodes. Therefore, our approach does not result into exponential time complexity. However, the processing time for computing all-paths is significantly high that it affects the applicability of our approach in real-time situations.

To reduce this processing, we approximated all-path computation and determined the candidate paths by restricting the path-length. The average processing time for computing candidate-paths is $\sim$3355 seconds and the worst processing time is $\sim$35109 seconds. Though, this approximation improves the processing time of

our approach yet when compared to other existing approaches it is slightly high. In order to minimize the time complexity of this phase, we can use more efficient path-computation algorithm.

Quinn *et al.* [138], proposed a survey of various parallel graph algorithms using systolic arrays, associative processors, array processors, and multiple CPU computers. General-Purpose Computing on Graphics Processing Units (GPGPU) provides a powerful platform to implement the data-intensive algorithm. Kaczmarski *et al.* [152] proposed an approach for accelerating the Breadth First Search (BFS) algorithm with CUDA implementation on GPU. The authors have shown the significant improvement over CPU based implementation of BFS. The results we present, show great promise in using semantically-relevant paths to classify malware, the computational complexity would be prohibitive in a real-time setting. In future, we will also create the parallel version of our path-computation algorithm and reduce the incurred overhead.

5.4.4.4 Training Time

Training time includes the time of feature vector construction and learning time. This time is one time cost of the order ∼1.54 seconds. Although, to keep our system up-to-date, we need to train our model with newer samples within fixed time interval (monthly or quarterly).

5.5 Summary

Malware detection is the first line of defence against malicious threats. Modern malware is detection-aware therefore detecting all types of malware is a daunting task.

In this chapter, we proposed a new mechanism for identifying current malicious binaries that are resilient to static and dynamic obfuscation techniques. To carry out this objective, we captured the execution flow of malicious binaries in terms of system-calls and transformed them into Ordered System-Call Graph (OSCG).

Then, we applied the concept of Asymptotic Equipartition Property (AEP) inherited from information theory. Using AEP, we produced a set of semantically-relevant paths from each OSCG. These paths cumulatively describe the average behavior of a binary. Our experimental results demonstrate that semantically-relevant paths can be used to infer the malicious behavior and to detect numerous new and unseen malware samples.

The proposed approach shows its robustness against system-call injection attacks. In addition, we compared our method with existing solutions. We observed that our approach was more efficient in terms of malware detection rate. Our future work will focus on the development of path computation algorithms to reduce the overhead.

Chapter 6

Conclusions and Future Work

Over the last decade, malware has emerged as a crucial security threat. The proliferation of advanced computing and networking technology has empowered malware programs with advanced anti-detection and anti-analysis features. The advanced malware programs instigate a variety of attacks such as Distributed Denial of Service (DDoS) attacks, social engineering attacks and clickfraud attacks, to name a few. These software programs have a disruptive impact on our applications, service providers, storage, servers, and networks.

Static malware detection approaches are effective in identifying known malware binaries. These approaches are not sufficient to detect new and unseen malware samples and result in high false alarm rate. It has been reported that code-obfuscation and simple encryption/compression techniques are capable to evade signature-based approaches. To overcome the limitations of static approaches, we have developed behavior-based dynamic malware detection approaches that are not vulnerable to code-obfuscation, polymorphism, packing to name a few.

In this thesis, we have shown that the present malicious threats are bundled with multiple behavior payloads needed for evading detection through employment of various anti-detection features. We have developed techniques that address two anti-detection features of modern malware *i.e.*, environment-reactiveness and system-call injection attack.

In this chapter, we summarize conclusions drawn from our research work along with the possible directions for future. The major aim of our research is the development of novel malware detection solutions that also address detection-aware malware. To achieve this goal, we have designed non-signature based approaches, which support generality and provide better detection accuracy with the new and old generation of malware samples.

6.1 Conclusions

In this research work, we have presented behavior-based malware detection approaches. We have analyzed system-call sequences acquired after executing malware and benign samples in Ether framework. Discriminant system-call sequences are extracted using 1) DTW algorithm, 2) multilayer neural network, and 3) program semantics. We transformed these sequences into Ordered System-call Graphs (OSCGs) and observed that OSCGs can preserve the ordered relationship between system-calls. Therefore, we were able to detect malicious samples more effectively as compared to other existing approaches. The proposed approaches are evaluated using performance measures detection accuracy, evaluation with other dataset, and system overhead.

1. Our DTW-based approach enables us to explore multiple behaviors within a malware family and also can differentiate malware from benign programs. DTW algorithm can capture variability in behavior of malware samples. To validate the DTW formed clusters, we have applied the single-linkage hierarchical algorithm in conjunction with Davies-Bouldin index. This validation also confirms our heuristic and the effectiveness of DTW.

    (a) We have evaluated our detection model with worm and virus datasets and achieved 98.58% (worm) and 92.95% (virus) of detection accuracy respectively. Therefore, the overall accuracy is ∼95%. The false alarm rates (worm) – FPR (1.27%) and FNR (0.91%) – are evaluated with benign samples.

(b) The proposed approach is evaluated with two malware datasets containing worm and virus samples. Our worm samples show four different behavior that are all distinct from benign samples. Similarly, the virus samples also show different behaviors from each other. The false alarm rate with virus samples is more when compared to worm samples. Because some of the virus samples mimic benign behavior therefore show similarity with benign samples.

(c) We have observed that DTW is a computationally expensive and require long time when the source and the target sequences are large. Through parallelization (P-DTW), quadratic complexity of sequential DTW can be reduced to linear complexity. We have obtained a speedup of 30.55 by parallelized DTW over the sequential DTW.

2. Our neural network based approach identifies the environment-aware malware through monitoring and classifying its reactions during execution in the virtual environment. The constructed multi-layer perceptron model is capable in recognizing patterns in high-dimensional feature space.

(a) We tested our approach with known and unknown samples. With known samples, we achieved overall ∼96%. The detection accuracy achieved with unknown samples is ∼95.4%.

(b) Through extensive experimentation, we determined the appropriate input vector that includes 200 transition states with higher probabilities. The selected input vector produces a minimum error of ∼1.3% that contributes into false-alarm rate of proposed approach. Though, the overall false alarm rate of our approach is ∼2.6% that is negligible as compared to other approaches. With effective rate of detection accuracies, our approach can segregate samples with clean, malicious, guest-crashing, and infinite-running behavior.

(c) In order to achieve a high efficiency with respect to training time, we trained all four networks in parallel by making use of JAVA threads. We achieved 1.45 speedup with heap space of 3 GB. This speedup was achieved due to the lower execution frequency of the JAVA garbage collector in higher heap space.

3. Finally, we presented a novel malware detection approach that is resilient to the system-call injection attack. We showed that the performance of our approach was not degraded even after inserting thousands of system-calls. The proposed approach is based on the concept of Asymptotic Equipartition Property (AEP) that is adopted from information theory domain. Using this concept, we have extracted the semantically-relevant paths that represent the program behavior. Our experiment results indicate that semantically relevant paths can be used to infer malicious program behavior for detecting new and unseen malware samples.

   (a) We observed an overall accuracy of ∼95% with TPR = 95.4% and FPR= 4.6%. We also compared the false alarm rate of our approach with other existing approaches. Our approach outperformed existing approaches as the specified program behavior of malware samples was discriminative from benign samples.

   (b) The proposed approach also shows its robustness against system-call injection attacks. We injected thousands of rare and frequent system-calls into malware traces and then tested those traces with our model. We have observed that our approach remain unaffected up to call-injection of 30%(more than 30 thousand calls).

   (c) We also compared our approach with existing state-of-the-art approaches and found that our approach was more efficient in terms of detection rate and resiliency towards system-call injection attacks.

6.2 Limitations and Future Work

In this thesis, we present approaches that detect malware and also address the anti-detection features of the modern malware. However, our approaches are not free from limitations. Here, we sketch limitations and their solutions as future work.

1. The proposed research work relies on one analysis framework to capture system-call traces of program binaries. During behavior monitoring, we observed that there are few malware samples for which logs are not generated. Therefore, to acquire the logs of complete malware dataset, we should use multiple analysis frameworks. However, in Chapter 5, we also used Cuckoo sandbox and showed that our approach was not specific to Ether. Our methods can be used with other analysis frameworks as well.

2. The present malicious threats are equipped with many other anti-detection features. We, in our work, have addressed two anti-detection features. In future, we must consider other anti-detection features such as trigger-based malware behavior. For this, user input during execution is required as trigger-based malware samples show their behavior under certain trigger conditions (date, time-stamps, URLs, file created/deleted, and to name a few). It will also allow us to explore multiple execution paths.

3. We have shown that our approaches can be used to detect Windows malware. To make our approach platform independent, in future, we can map our models to other platforms such as Android, Linux, and Mac. Presently, the Android malware is gaining popularity among security researchers. Therefore, to detect Android malware we can apply our approaches with malicious android applications.

Appendix A

Ether: Dynamic Behavior Monitoring Tool

A.1 Introduction

Malware analysis using sandbox environment or controlled environment is gaining prominence. The primary advantage of analyzing in a controlled environment is the host operating system remains unaffected. We have used Ether patched XEN capable of executing multiple virtual machines, each having its own operating system, on a physical system. In case of system virtualization the hypervisor or Virtual Machine Monitor (VMM) manages multiple operating system and the resources. The performance of using XEN is close to using the native system. XEN based hypervisor acts as an interface between the guest operating system, which in our case is Windows XP2 and a host operating system (Debian Lenny).

A.1.1 Ether Components

To boot the guest OS, hypervisor needs disk, kernel image and configuration file consisting of IP address, amount of memory to use. The hypervisor includes

elements such as hypervisor layer, interrupt handler, page mapper, and scheduler. In subsequent paragraphs, each element is discussed in brief.

1. Hypervisor Layer: It keeps the guest and the host operating systems connected. Using the hypercall[1] layer, the guest operating system interacts with the host operating system.

2. Interrupt Handler: This component of a typical hypervisor route the interrupt to/from a guest operating system and virtual devices. Likewise, the hypervisor is designed to identify and understand faults or exception occurring at specific guest operating system. The faults occurring at a guest are not transferred to the hypervisor to prevent interruption in the working of the hypervisor.

3. Page Mapper: It maps the hardware to pages of specific guest operating system. The guest domain's memory mapping is created and updated only by hypervisor.

4. Scheduler: Transfers the control between multiple guest operating systems and itself back and forth.

Dynamic analysis is performed on the DomU Xen machine(XP SP2), and its footprints are recorded on the DomO system(Debian Lenny).

A.1.2 System-call Monitoring Using Ether

For system-call monitoring, we employed Ether [46]. We prefer Ether to other analysis frameworks as it provides host-based tracing by employing hardware virtualization. It is resilient to anti-debugging, anti-emulation and code-obfuscation, and in-guest changes are also made hidden [99].

Ether produces a page fault or exception to intercept the system-calls made by the target application. Whenever this application requires a system service, it executes `SYSENTER` that transfers the control to kernel space where it copies the value

[1] A hypercall (hypervisor call) is a paravirtualization interface allowing guest OS to access hypervisor services.

(address) stored in a special register SYSENTER_EIP_MSR into instruction pointer (IP). Ether sets SYSENTER_EIP_MSR to a default value. Accessing this value causes a page fault and in this way Ether knows that a system-call has been made. The SYSENTER_EIP_MSR is changed back to its original value, and the target application continues its execution. Ether mediates all access to the SYSENTER_EIP_MSR register and can, therefore, hide any modifications of the register from the analysis target.

To generate system–call logs, each sample is permitted to execute for 10 minutes. According to Quist *et al.* [146], five minute is enough duration for the execution monitoring. We doubled this execution time to capture the malware equipped with capability of carrying out time–out attacks.

Figures A.1,A.2,A.3,A.4,A.5 shows the process of system-call tracing in Ether framework.

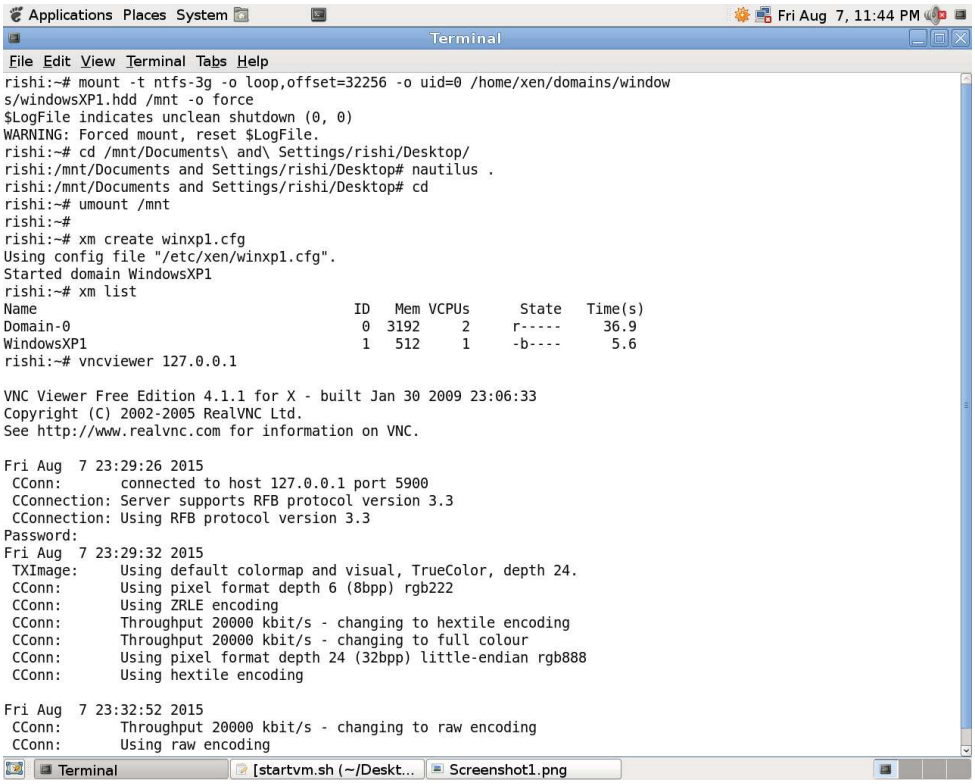


FIGURE A.1: Booting guest-OS

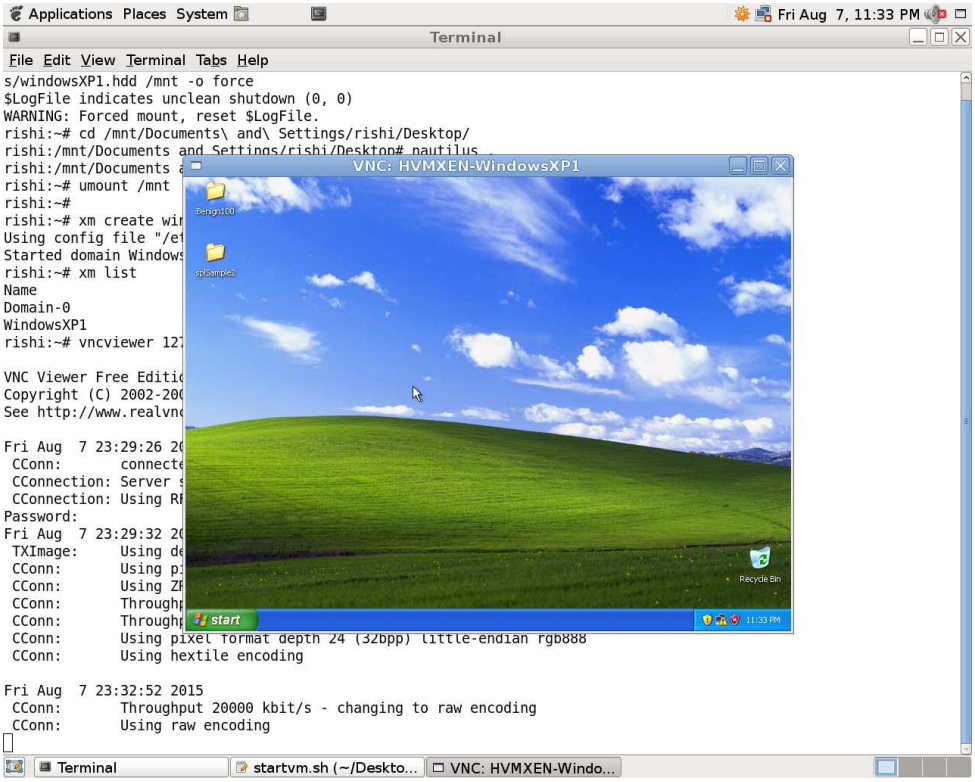


FIGURE A.2: Windows-XP has started

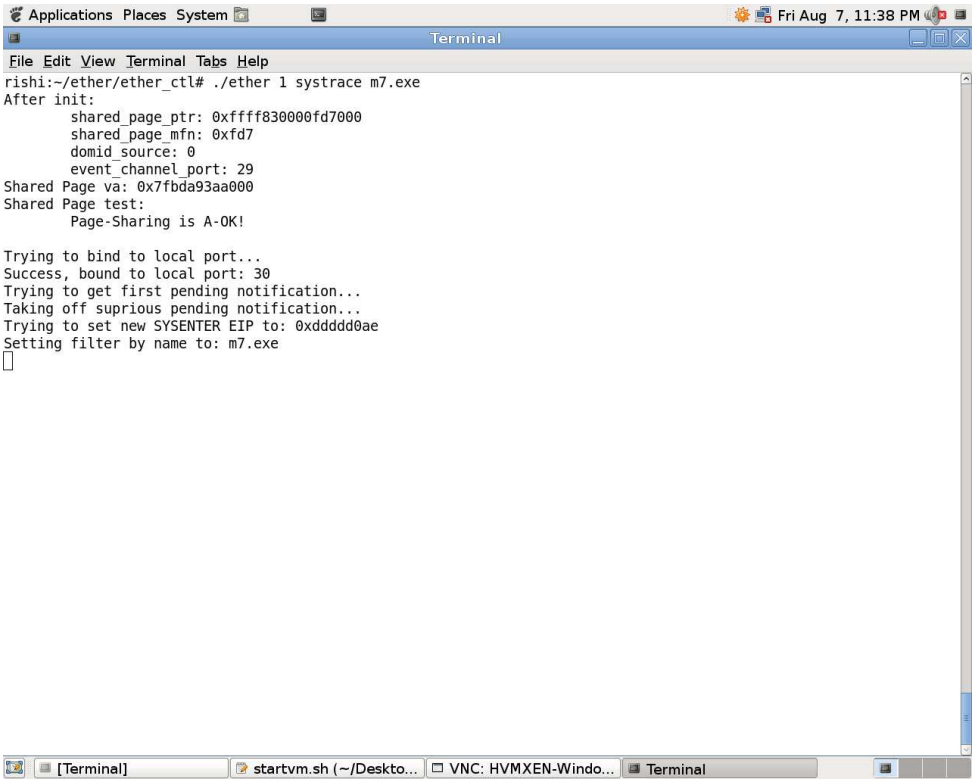


FIGURE A.3: Starting Ether agent at host for executing sample m7.exe

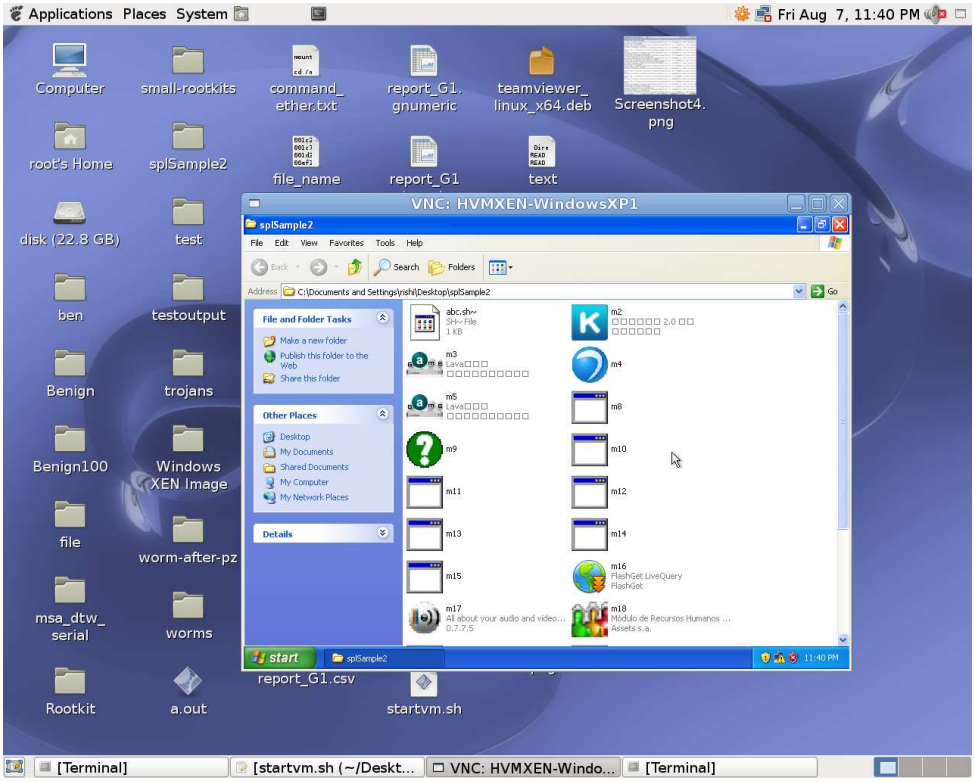


FIGURE A.4: Executing m7.exe in guest-OS

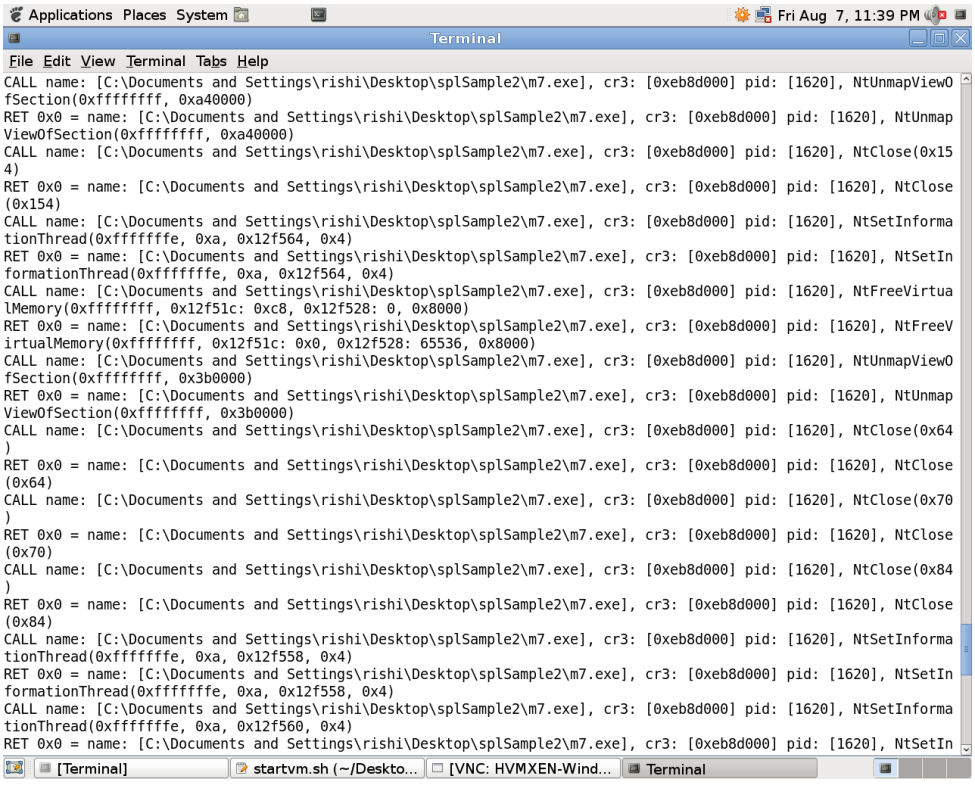


FIGURE A.5: System-call logs of sample m7.exe

Appendix B

Malware Executables

In this appendix, we present a categorization of malware programs.

B.1 Malware

"Malware" is a software program that fulfills the malicious intent of an attacker. These malicious programs can be put into two categories as shown in Figure B.1, *i.e.*, blackware and grayware, based on how dark their malicious intent is. A malware program is categorized into virus, worms, trojan horse rootkits, bots, and spywares. These malware families employ various mechanisms to exploit the target systems. Following paragraphs describe the characteristics [24] of these malware families.

B.1.1 Virus

A virus is a program code that attaches itself to other executable programs. It relies on other host program to accomplish its hostile intentions. Execution of a virus on a host can infect other programs/applications. This self-replication into existing executable code is the key characteristic of a virus [24]. It requires user intervention to spread, and that is why it is parasitic in nature. Viruses

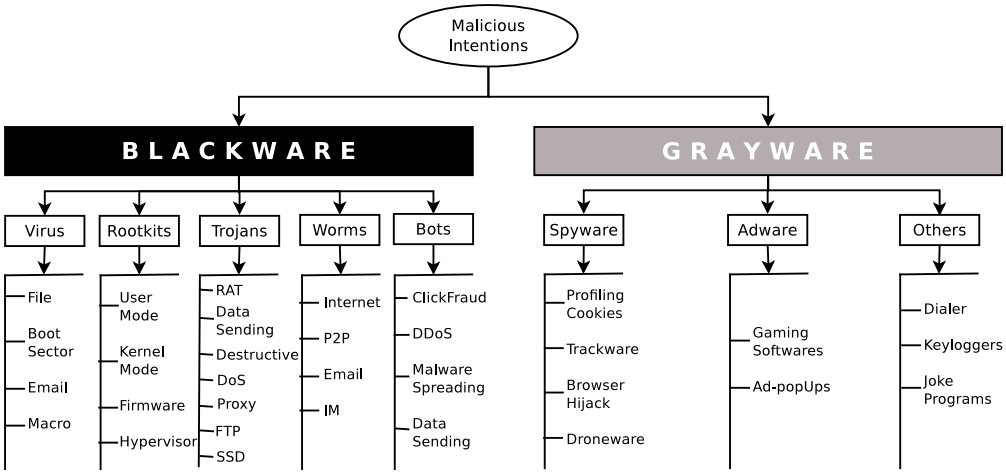


FIGURE B.1: Malware classification

can contain multiple malicious payloads for data stealing, damaging and other ill consequences. Viruses can be categorized according to the programs they infect. Virus infects local files and/or system files, emails, scripts, and boot-sectors to improve chances of its survival as well as hide its hide its presence. Infecting multiple entities increases complexity of virus removal and disinfecting the host. Virus types on the basis of objects it infects are:

- *File viruses* can infect the program files with extension *.exe, .dll, .bin, .sys, .bat*. `Sunday`, `Cascade`, `Professor`, and `Jerusalem` are the examples of such type of viruses.
- *Boot sector viruses* infect the boot records on the hard disks. Boot record is a program that is responsible for system start-up. Examples are `Parity boot`, `Disk Killer`, `Stoned`, `Brian`, `Michelangelo`, `Empire`, `Form`, and `Azusa`.
- *Email viruses* are the piece of code that spread as an email attachment. `Melissa` and `I Love You` viruses come under this category.
- *Macro viruses* spread through Microsoft office applications like word, excel, and power point. These kind of viruses are platform independent and can infect operating systems like Linux, Mac and Windows.

B.1.2 Worms

A worm shares some similar characteristics with virus like self-replication and population growth. Worms are independent programs that do not rely on other executable programs for replication and execution. Worms spread infection by targeting other vulnerable machines in the network. A worm uses Internet or network connections to propagate itself and copies its code to other computer programs. Worms locate the vulnerable system in the network to exploit it. Yong Tang *et al.* [29] have classified worms into following four categories according to the regions where worm searches the exploitable target.

- *Internet worm* performs scanning of IP addresses to identify a weak target in the network. `Code Red I`, `Code Red II`, `Nimda`, `Blaster`, `Slammer`, and `Benjamin` are some of the examples of Internet worm.
- *P2P worm* infects file sharing networks (Gnutella, Kazaa, eDonkey2000, Poisoned, Freenet and BitTorrent) using peer to peer connectivity. These worms copy themselves into the shared directory on a local machine. `Benjamin` worm is an example of P2P worm.
- *Email worm* exploits E-messages and spreads through Internet. It searches the address book of the victim to propagate infected email to other computer systems. `Worm.ExploreZip`, `Melissa`, `W32/Waledac.A`, `love letter`, `MSIL/Agent.MXK`, `MyDoom`, `Blackmail`, and `W32/Brontok.N` belong to this category of worm.
- *Instant Messaging (IM) worm*, typically spreads via instant messaging (network targeting IM users and protocols). It infects the IM contact list of the users. `OSX.Leap.a` [153] is the IM worm that infects the system running with Mac OS. This worm uses Apple's "iChat" application for infection. Other examples are `Choke`, `JS Menger`, and `Serflog`.

B.1.3 Trojan Horse

Trojan is a program that pretends to be benign but performs malicious task(s) in the background. It disguises itself into useful programs like screen-savers, plug-ins or games, utilities and various freewares [26]. It also contains malicious code that once installed, can accomplish ill purposes like stealing sensitive data and passwords. A Trojan remains invisible to various scanning tools. It does not replicate itself but it is parasitic by nature. Trojan payload is designed for data destruction and remote access to other malware programs. Dancho in [25, 28] categorized trojans into following categories according to the payload contained:

- *Remote Access Trojans (RATs)* provide unauthorized access of victim's system to some hacker who can remotely administrate the system. Examples are `Nuclear-RAT`, `Netbus`, `Poisonivy-RAT`, and `Back Oriface`.
- *Data-Sending Trojans* transfer useful information such as passwords, credit card numbers and logs of chats, keystrokes and browsing history to the trojan writers through an installed spyware software. This software logs all the mentioned activities on our system. `Badtrans.B` is an example of the data-sending trojan.
- *Destructive Trojans* contain payload that can wipe out either hard drive contents or corrupt/remove selected data and system files from the target system.
- *Denial of Service (DoS) Attack Trojans* have the ability to launch DoS attack to multiple victims. These programs spread by attacking email addresses or infecting ADSL users for blocking Internet sites.
- *Proxy Trojans* are kind of trojans that convert victim's computer into a proxy server and allow attackers to access that machine. This infected machine can be used for illegal activities such as Cyber-bullying, Cyber-terrorism across the globe.
- *FTP Trojans* gain access to a target machine through a FTP server. Once access is obtained, these trojans may upload / download applications and data to/from server.

- *Security Software Disablers* disable security software installed or configured in our system. These trojans target application software such as AV scanners and firewalls.

B.1.4 Rootkits

Rootkits have the ability to hide themselves from the owner of the system. Rootkits execute with root privileges to access full set of system resources. Attackers need to have administrative access to install the rootkit in our systems. The rootkits can replace existing programs and system libraries with their modified malicious versions. The rootkits can operate at both user and kernel levels. These programs can be installed as a kernel module and get the root privilege to modify system resources. Rootkits are used in conjunction with trojans to exploit the vulnerabilities in the system. Many malware binaries employ rootkits' hiding mechanism to hide their presence. Rootkit writers are aware of the fact that discovery of their malicious intention on victim system may result in sealing of the vulnerability and loss of access. To gain access, a new rootkit shall be needed. Rootkits erase the login and logout data so that a security researcher is unable to notice its presence. The rootkits can modify kernel data structures, system call table, and system service descriptor Table (SSDT). Rootkits are classified according to the system area where they reside:

- *User mode rootkits* stay in hidden system folders, registry. User mode rootkits can hide themselves by hooking process viewer such as Windows task manager [22]. `Qoolaid`, `lkr`, `trOn`, and `ark` are the some examples of user mode rootkits.

- *Kernel mode rootkits* load their code into kernel address space. Usually kernel programs are accessed by device drivers and system libraries. These programs are used as an interface to the hardware. Kernel mode rootkits and operating system operate on the same security level and thus they can intercept kernel mode objects. `Da Ios` is the known example of kernel rootkit.

- *Firmware rootkits* are actually embedded within the firmware of devices such as network card, system BIOS, hard drive [22]. Rootkits stay undetected in firmware as it is not checked for code integrity. John Heasman demonstrated the viability of firmware rootkits in both ACPI firmware routines and in a PCI expansion card ROM [46].
- *Hypervisor rootkits* exploit the hardware virtulization features of Intel VT and AMD-V. Such kind of rootkits can intercept hardware calls to operating systems as they have the Ring-1 privileges.

B.1.5 Bot

Bot permits its creator (Bot master) to remotely access the infected machine. A computer that has been compromised by a bot is referred to as zombie or drone [27]. Bots can create a network termed as *botnet* by which every instance of bot can communicate with each other. Botnets are created if bots replicate itself to other systems. The bots are classified by their attacking mechanism:

- *ClickFrauds:* These type of bots abuse Pay Per Click (PPC) advertising. The website owner publishes certain ads on their webpages and get paid by the advertiser as per number of clicks. The rival advertisers of same product and website publishers use these bots to kill the competition and to gain financial benefit respectively.
- *DDoS:* Distributed denial of service attack is one of the primary attacks by bots. Through such attacks, bots render computer resources unavailable to the authorized user. `Hameq`, `pushbot`, and `waledac` are the common types of such bots.
- *Spamming:* Botnets are the main source of spam circulation. The botnet authors get email addresses either by crawling web or purchasing the list from other spammers. These kind of bots are also called as spambots.
- *Phising:* Phishing is incorporated to steal personal information such as login credentials and bank details by diverting URL's of popular websites to lookalike websites. `Pushbot` is one of the example of such bots.

- *Distributing and Installing malware:* Botnets are used to distribute and install other malware programs. It adopts social engineering, drive-by-downloads and spams to fulfill their objective. `Rimecud` is the bot used for these attacks.

- *Data stealing:* As the name suggest, these attacks are incorporated to steal victim's personal information such as licence key of software product, browsing history, and saved passwords. `Rbot, Zbot` are popular data stealing bots.

B.1.6 Grayware

Spyware collects sensitive information from target system and transmits this information to other systems. Recipient of this information is the attacker who forces spyware to retrieve the information. Such an attacker is often interested in usernames, passwords, email addresses, software license keys, bank account and credit card numbers. Stealing these credentials allow attacker authenticated access to victim's financial resources.

Grayware is a software that installs components on a computer for the purpose of recording Web surfing habits (primarily for marketing purposes). Spyware sends this information to its author or to other interested parties when the computer is online. Spyware often gets downloaded with items identified as 'free downloads' and does not notify the user of its existence or ask for permission to install the components. The spyware components gather information such as user keystrokes. This exposes private information such as login names, passwords, and credit card numbers to theft.

Appendix C

CUDA & GPGPU

C.1 CUDA

GPU (Graphics Processing Unit) are being used to manipulate computer graphics to improve performance of image creation. GPUs are continuously outperforming the CPUs with respect to execution speed due to a large number of fast computing cores on high-end graphics cards. The GPU architecture is somewhat similar to a CPU. The difference is that GPU is designed to handle streaming data. As shown in Figure C.1, a GPU devotes more transistors to data processing while a CPU utilizes the transistors for data caching and flow control [154]. Since streaming data is already sequential, or cache–coherent, the GPU does not need a large amount of cache. This gives the GPU an advantage in highly parallel computations, where the number of arithmetic operations is far greater than memory operations.

For many years, GPU functionality was limited to accelerating some parts of graphics pipeline. Later it was observed that the multiple cores available on GPU can be utilized to do the processing in parallel. If we use the GPU for processing non graphical data, then it is known as the General Purpose GPU or GPGPU.

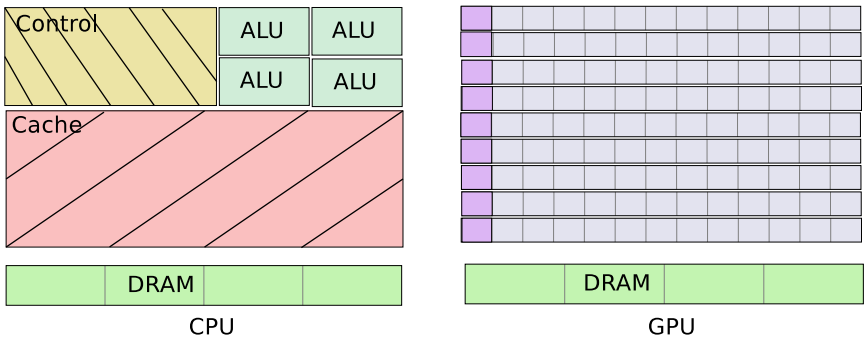


FIGURE C.1: Central Processing Unit (CPU) vs Graphics Processing Unit (GPU)

GPGPU is used for performing computationally intensive operations in parallel for achieving low time complexity.

CUDA (Compute Unified Device Architecture) is NVIDIA's GPU architecture featured in the GPU cards, positioning itself as a new means for general purpose computing with GPUs. CUDA C/C++ is an extension of the C/C++ programming languages for general purpose computation. CUDA gives the advantage of massive computational power to the programmer.

C.1.1 CUDA Programming Model

CUDA programming model is the seat of serial and parallel execution as shown in Figure C.2. The serial code is run on CPU also called as *Host*, and parallel execution is done on GPU also called as *Device*. The host code is simply a C Code and compiled with standard C compiler. The device code uses CUDA variables and functions, calls kernels. The GPU (device) contains a set of multiprocessors. Each of the multiprocessors includes a group of stream processors which are operable on SIMD (Single Instruction Multiple Data) programs [155]. Figure C.2 shows the programming model of CUDA. CUDA architecture provides three basic components to utilize fully the capability of graphics card in the system. These three parts are grids, blocks, and threads.

- **Grid**

  A grid is formed by arranging multiple one, two or three-dimensional thread blocks. Multiple grids can be run at a time but to start a grid CPU performs

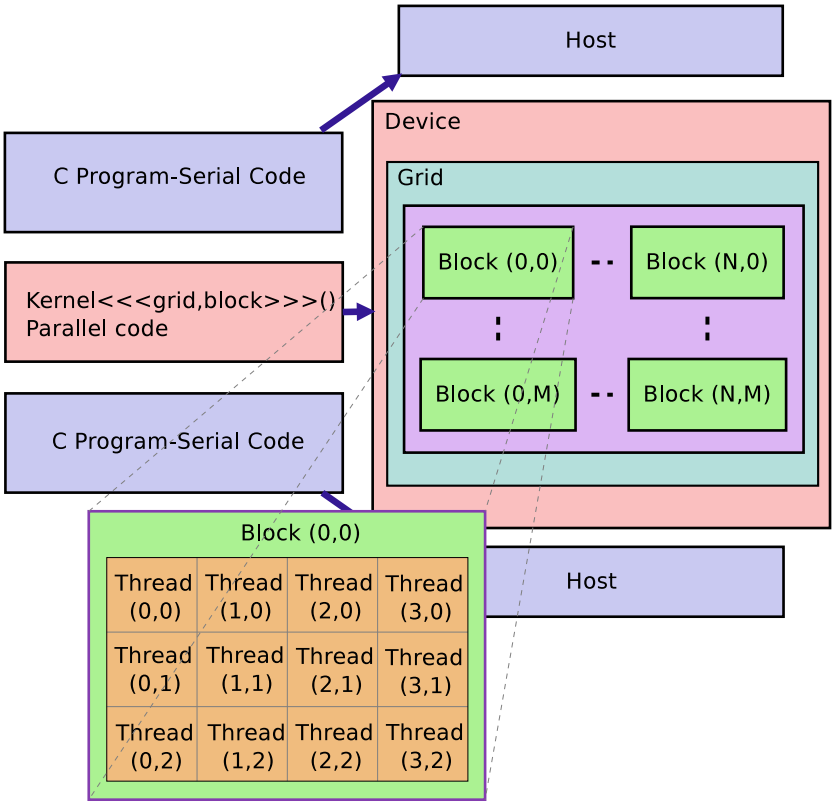


FIGURE C.2: CUDA programming paradigm

synchronous operations. Grids cannot be shared between multiple GPUs in multi-GPU systems.

- **Block**

  A thread block is comprised of several one or two dimensional threads which can communicate only within their own block. Each block has its own shared memory that can be shared only by its comprising threads. Also, blocks cannot be shared between multiprocessors. In a grid, there are multiple blocks and all blocks use the same program. To identify the block, a variable is used called *blockIdx*.

- **Thread**

  Thread blocks are composed of threads. Unlike grids and blocks, threads can be shared between cores. To identify the particular thread, a variable named as *threadIdx* is used which can be one, two or three dimensional based on block dimension. Generally, there are 512 threads in each block. These

threads are managed by multiprocessor in group of 32 called warps. Also, these threads are responsible for executing CUDA kernel.

Execution of a typical CUDA program starts with the CPU (host) execution that is serially coded. To transfer the control from host to device, a kernel function is invoked. This kernel code is a C code that is run by each thread. The work starting from thread creation till thread termination is automatically handled by GPU only. The user can give only the number of thread blocks in a grid and threads in a block for running the kernel function in call to kernel within three angular brackets i.e <<< grid, block >>>. Here grid variable contains a number of thread blocks in a grid and block variable is used for a number of threads in a block. These values of grid and block variables must be less than the maximum number of thread blocks and threads. The kernel function always has a return type *void* and a qualifier __global__ which means that the execution of this function is done on GPU. When the kernel is called, the execution is moved to the device where the code is executed in parallel mode. Once all threads synchronously complete their execution in parallel, the corresponding grid terminates and the execution continues on the host.

C.1.2 CUDA Memory Model

Programmers can utilize the hierarchy of memory architecture available on GPU [156]. CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure C.3. According to CUDA programming guide [157], CUDA memory model includes:

- *Registers*: It is the fastest *read-write* memory per thread.
- *Local Memory*: This type of memory is local to a particular thread and readable and writable by only that thread. This memory is not cached, so it is as slow as the global memory. Its use is avoided in maximum conditions. Local memory is used only when the size of the data is large enough not to fit in registers.

- *Constant Memory*: It is cached *read–only* memory per grid. Constant memory is situated off the GPU chip so it can be accessed by all active threads on GPU as well as CPU. But write access is not allowed for GPU to this memory. It can only be written by the host, and it remains same/constant for all kernel launches. For all threads of a particular warp, reading from the constant cache is equivalent to reading from a register as long as all threads read the same address.

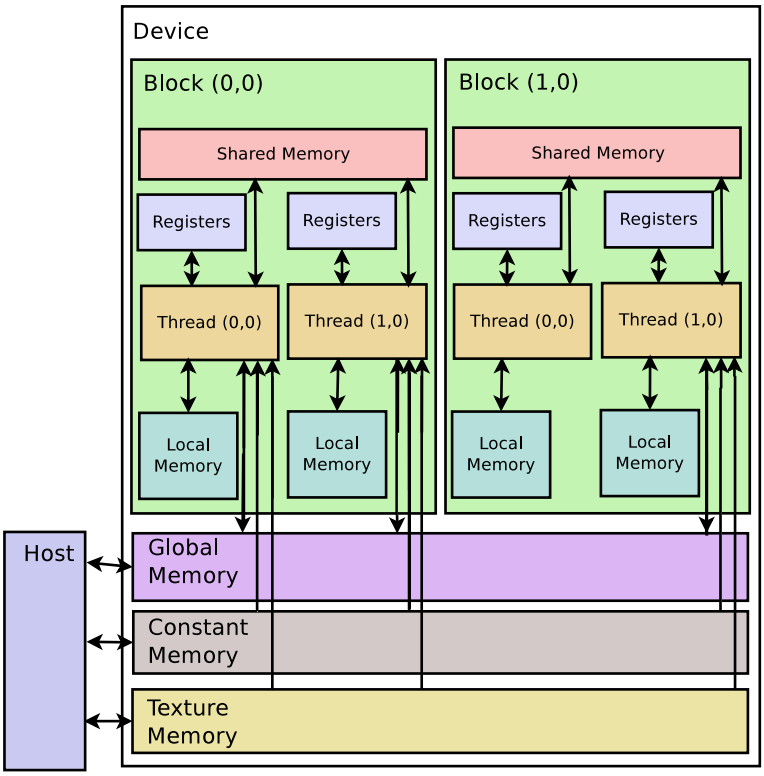


FIGURE C.3: CUDA hardware model memory layout

- *Shared Memory*: This is *read-write* memory for accessing data shared by threads in the same block. As long as a block is persisted in multiprocessor, this memory is also persisted for threads in the block. This memory is divided into equal sized banks with each bank accessed in parallel. If bank conflicts are not there, then this memory is as fast as registers. However, the capacity of shared memory is limited to 16 KB/multiprocessor.

- *Texture Memory*: Like constant memory, texture memory is cached on chip and each cache is of size 8KB. This memory is readable by device and host

but writable only by host. The host binds the data to the texture variable, and this data is accessed by the kernel using the function. It is designed in such a way that threads reading addresses with close proximity will achieve better transfer rate. Maximum efficiency is achieved when all threads read the close texture addresses.

- *Global Memory*: It is non-cached memory which has *read* and *write* access by the device as well as host. The persistence of global memory retains from its allocation to deallocation. Since it is not cached, it decreases the performance. Global memory bandwidth is the most efficiently used when memory accesses by a thread half-warp are combined into a single memory transaction maximizing PCIe bandwidth [157].

C.1.3 NVCC Compiler

NVCC is a compiler driver provided with the CUDA Toolkit. NVCC executes all of the necessary tools and compilers included with the CUDA toolkit required to compile device code. Kernels can be written using either CUDA instruction set architecture called PTX or a high-level language like C. In both the cases, these kernels must be compiled by NVCC into binary (cubin) code before being executed on the device [157]. It provides simple and familiar command line options and invokes the tools that implement the different compilation stages and execute the code.

Source code of the program contains two type of codes in mixed form. First, host code that is run on the host and second Device code that is intended to run on device. NVCC compiler is responsible for separating host code from device code. Device code is compiled in assembly form *i.e.* PTX code. Host code is modified by replacing the kernel syntax code i.e. $<<< \cdots >>>$ into function calls and then each function call is invoked by PTX code. The modified host code is output either as C code that is left to be compiled using another tool or as object code directly by letting NVCC invoke the host compiler during the last compilation stage [157].

C.1.4 NVIDIA Tesla C2075

We have used NVIDIA Tesla C2075 [158] for our experimnets. Table C.1 shows the harware specifications od the Tesla C2075.

TABLE C.1: Tesla C2075 hardware specifications

Hardware item	**Value**
Chipset	Tesla C2075
Core Clock	575 MHz
Number of Streaming Processors	448
Memory Interface	384 bit
Memory Type	GDDR5
Memory Size	6 GB
Memory Speed	1.5 GHz
Shader Clock	1150 MHz

On our target GPU, the NVIDIA Tesla C2075 [158], there are a total of 448 streaming processors (CUDA cores) [158]. It delivers up to 515 Gigaflops of double precision peak performance in each GPU, enabling a single workstation to deliver a Teraflop or more of performance. As shown in Table C.1, the GPU has upto 6GB of GDDR5 memory per GPU, which reduces data transfers by keeping larger data sets in local memory that is attached directly to the GPU. This architecture maximizes the throughput by faster context switching that is $10X$ faster than previous architecture and provides concurrent kernel execution and improved thread block scheduling [158].

List of Contributions

A. Journal Publications

[J-1] "Employing Program Semantics for Malware Detection", *IEEE Transactions on Information Forensics and Security*, vol. 10, No. 12, pages 2591-2604, December 2015. doi 10.1109/TIFS.2015.2469253.

[J-2] "An Efficient Block-discriminant Identification of Packed Malware", *Sadhana Academy Proceedings in Engineering Science*, , Volume 40, Issue 5, pages 1435-1456, August 2015. doi doi=10.1007/s12046-015-0399-x

[J-3] "Dynamic Behavior-based Malware Detection Techniques: A Survey", Communicated in *Computer Science Review, Elsevier*.

B. Conference Publications

[C-1] "Exploring Worm Behaviors using DTW", 7^{th} *International Conference on Security of Information and Networks (SIN '14)*, Glasgow, Scotland, UK. pages 379:379–379:384, Sept, 7-11, 2014.

[C-2] "Environment-Reactive Malware Behavior: Detection and Categorization", 7^{th} *International Workshop on Autonomous and Spontaneous Security (SETOP '14)*, Wroclaw, Poland. pages 167-182, Sept, 11, 2014.

[C-3] "P-SPADE: GPU Accelerated Malware Packer Detection", 12^{th} *Annual Conference on Privacy, Security and Trust (PST '2014)*, Toronto, Canada. pages 257-263, Jul 23-24, 2014.

[C-4] "Poster Paper: Signature based Packer Detection", *In the proceedings of SIS-SNDA*, BITS Pilani Hyderabad, India. Nov, 2013.

[C-5] "MCF: MultiComponent Features for Malware Analysis", 27^{th} *International Conference on Advanced Information Networking and Applications Workshops (WAINA '13).* pages 1076-1081, Mar 25-28, 2013.

[C-6] "Relevant Hex Patterns for Malcode Detection". *In Proceedings of International Conference on Intelligent Systems and Signal Processing (ISSP '13).* Gujrat, India, pages 334-337, Mar 1-2 2013.

[C-7] "Random Block Entropy Analysis of Packed Executables". 27^{th} *National Convention and National Seminar on Latest Advancements in Computer Engineering (LACE 2013).* Jaipur, India, February 2013.

Bibliography

[1] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *Proc. of the 22Nd USENIX Conference on Security (SEC'13)*, pages 637–652, 2013.

[2] Eugene H. Spafford. The internet worm program: An analysis. *SIGCOMM Comput. Commun. Rev.*, 19(1):17–57, January 1989.

[3] VirusTotal. File types statistics. https://www.virustotal.com/en/statistics/, Feb. 2015.

[4] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to own the internet in your spare time. In *Proc. of the 11th USENIX Security Symposium*, pages 149–167, Berkeley, CA, USA, 2002.

[5] Kaspersky Lab. http://www.kaspersky.com/about/news/virus/2013/, 2013.

[6] Kaspersky. The red october campaign an advanced cyber espionage network targeting diplomatic and government agencies. http://www.securelist.com/en/blog/785/, January 2013.

[7] Alexander Gostev. What is flame? http://www.kaspersky.co.in/flame, May 2014.

[8] David Kushner. The real story of stuxnet. http://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet, February 2013.

[9] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.*, 19(4):639–668, 2011.

[10] Silvio Cesare, Yang Xiang, and Wanlei Zhou. Malwise: An effective and efficient classification system for packed and polymorphic malware. *IEEE Trans. Comput.*, 62(6):1193–1206, Jun 2013.

[11] Jusuk Lee, Kyoochang Jeong, and Heejo Lee. Detecting metamorphic malwares using code graphs. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1970–1977, New York, NY, USA, 2010. ACM.

[12] Wei Yan, Zheng Zhang, and Nirwan Ansari. Revealing packed malware. *IEEE Security and Privacy*, 6(5):65–69, sep 2008.

[13] Aniket Kulkarni and Ravindra Metta. A code obfuscation framework using code clones. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 295–299, New York, NY, USA, 2014. ACM.

[14] D. Spinellis. Reliable identification of bounded-length viruses is np-complete. *IEEE Transactions on Information Theory*, 49(1):280–284, Jan 2003.

[15] Yong Tang and Shigang Chen. An automated signature-based approach against polymorphic internet worms. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):879–892, 2007.

[16] P. Vinod, V. Laxmi, M.S. Gaur, and G. Chauhan. Momentum: Metamorphic malware exploration techniques using msa signatures. In *Proc. of International Conference on Innovations in Information Technology (IIT)*, pages 232–237, March 2012.

[17] Raymond Canzanese, Moshe Kam, and Spiros Mancoridis. Inoculation against malware infection using kernel-level software sensors. In *Proc. of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 101–110, New York, NY, USA, 2011. ACM.

[18] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. Semantics-aware malware detection. In *Proc. of IEEE Symposium on Security and Privacy (SP'10)*, pages 32–46, 2005.

[19] AV Test. New malware received last 10 years. https://www.av-test.org/en/statistics/malware/tab-6872-1, Aug. 2015.

[20] Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen. Splitscreen: Enabling efficient, distributed malware detection. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 25–25, Berkeley, CA, USA, 2010. USENIX Association.

[21] Robert Moskovitch, Yuval Elovici, and Lior Rokach. Detection of unknown computer worms based on behavioral classification of the host. *Computational Statistics and Data Analysis*, 52(9):4544 – 4566, 2008.

[22] Mcafee Report: Rootkits Part 2: A Technical Primer. www.mcafee.com.

[23] James P. Anderson. Computer security technology planning study. Technical report, Deputy for Command and Management System, USA, 1972.

[24] John Aycock. Computer Viruses and Malware. 2006.

[25] Dancho Danchev. *The Complete Windows Trojans Paper*. 2003.

[26] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, March 2008.

[27] Theodore Winograd Karen Mercedes Goertzel. Tools Report on Anti Malware. 2009.

[28] GFI Software. *A white paper on The corporate threat posed by email trojans-"How to protect your network against trojans"*. 2011.

[29] TANG Yong, LUO Jiaqing, XIAO Bin, and WEI Guiyi. Concept, Characteristics and Defending Mechanism of Worms. *IEICE Transaction of Information and Systems*, May 2009.

[30] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proc. of DARPA Information Survivability Conference and Exposition, (DISCEX'00).*, volume 2, pages 119–129. IEEE, 2000.

[31] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[32] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *USENIX Security*, volume 3, 2003.

[33] G.A. Di Lucca, A.R. Fasolino, M. Mastoianni, and P. Tramontana. Identifying cross site scripting vulnerabilities in web applications. In *Proc. of Sixth IEEE International Workshop on Web Site Evolution (WSE'04)*, pages 71–80, Sept 2004.

[34] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 135–144, New York, NY, USA, 2010. ACM.

[35] Michael Sikorski and Andrew Honig. Covert Malware Launching. 2012.

[36] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In Wenke Lee, Cliff Wang, and David Dagon, editors, *Botnet Detection*, volume 36 of *Advances in Information Security*, pages 65–88. Springer US, 2008.

[37] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proc. of the 1st ACM Workshop on Virtual Machine Security*, VMSec '09, pages 11–22, New York, NY, USA, 2009. ACM.

[38] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007.

[39] Procmon. Process monitor. http://live.sysinternals.com/, 2013.

[40] Gary HB. Responder pro. https://hbgary.com/products/responderPro/, 2013.

[41] Bayer Ulrich, Habibi Imam, Balzarotti Davide, Kirda Engin, and Kruegel Christopher. Insights into current malware behavior. In *Proc. of 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Boston, MA, 2009.

[42] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proc. of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[43] Emulator. Qemu. http://qemu.weilnetz.de/.

[44] Yin Heng and Song Dawn. Automatic malware analysis:an emulator based approach. Springer, 2013.

[45] Lok Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2e: Combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *Proc. of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, pages 227–238, New York, NY, USA, 2012.

[46] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proc. of the 15th ACM Conference on Computer and Communications Security*, pages 51–62, New York, NY, USA, 2008.

[47] Timothy Vidas. Volatile memory acquisition via warm boot memory survivability. *2013 46th Hawaii International Conference on System Sciences*, 0:1–6, 2010.

[48] Brian D. Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digit. Investig.*, 1(1):50–60, 2004.

[49] Stefan VöMel and Felix C. Freiling. A survey of main memory acquisition and analysis techniques for the windows operating system. *Digit. Investig.*, 8(1):3–22, 2011.

[50] SCOPE. WindowsSCOPE. www.windowsscope.com/.

[51] Sysinternals. NotMyFault.exe. http://live.sysinternals.com/Files/.

[52] John Hoopes. Virtualization for Security, December 2008.

[53] Ying Cao, Qiguang Miao, Jiachen Liu, and Lin Gao. Abstracting minimal security-relevant behaviors for malware analysis. *J. of Computer Virology and Hacking Techniques*, 9(4):193–204, 2013.

[54] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proc. of the 16th ACM conference on Computer and communications security*, CCS '09, pages 566–577, New York, NY, USA, 2009. ACM.

[55] Islam Rafiqul, Tian Ronghua, M. Batten Lynn, and Versteeg Steve. Classification of malware based on integrated static and dynamic features. *J. of Network and Computer Applications*, 36(2):646 – 656, 2013.

[56] Nizar Kheir. Behavioral classification and detection of malware through {HTTP} user agent anomalies. *J. of Information Security and Applications*, 18(1):2 – 13, 2013. SETOP'2012 and FPS'2012 Special Issue.

[57] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest–transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proc. of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[58] P. Trinius, T. Holz, J. Gobel, and F.C. Freiling. Visual analysis of malware behavior using treemaps and thread graphs. In *Proc. of 6th International Workshop on Visualization for Cyber Security, VizSec 2009*, pages 33–38, 2009.

[59] Liu Shun-Te, Huang Hui-ching, and Chen Yi-Ming. A system call analysis method with mapreduce for malware detection. In *Proc. of IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 631–637, Dec 2011.

[60] Domagoj Babić, Daniel Reynaud, and Dawn Song. Recognizing malicious software behaviors with tree automata inference. *Form. Methods Syst. Des.*, 41(1):107–128, aug 2012.

[61] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 45–60, Washington, DC, USA, 2010. IEEE Computer Society.

[62] Dai Jianyong, Guha Ratan, and Lee Joohan. Efficient virus detection using dynamic instruction sequences. In *J. OF COMPUTERS*, volume 4, pages 405–414, May 2009.

[63] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proc. of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 31–39, New York, NY, USA, 2010. ACM.

[64] SalvatoreJ Stolfo, Ke Wang, and Wei-Jen Li. Towards stealthy malware detection. In *Malware Detection*, volume 27 of *Advances in Information Security*, pages 231–249. Springer US, 2007.

[65] Christian Wressnegger, Guido Schwenk, Daniel Arp, and Konrad Rieck. A close look on n-grams in intrusion detection: Anomaly detection vs. classification. In *Proc. of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, pages 67–76, New York, NY, USA, 2013. ACM.

[66] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-based malware detection using dynamic analysis. *J. Comput. Virol.*, 7(4):247–258, 2011.

[67] A.F. Shosha, C. Liu, P. Gladyshev, and M. Matten. Evasion–resistant malware signature based on profiling kernel data structure objects. In *Proc. of 7th International Conference on Risk and Security of Internet and Systems (CRiSIS)*, pages 1–8, Oct 2012.

[68] Rhee Junghwan, R. Riley, Lin Zhiqiang, Jiang Xuxian, and Xu Dongyan. Data-centric os kernel malware characterization. *IEEE Transactions on Information Forensics and Security*, 9(1):72–87, Jan 2014.

[69] Chaoting Xuan, John Copeland, and Raheem Beyah. Toward revealing kernel malware behavior in virtual execution environments. In *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 304–325. Springer Berlin Heidelberg, 2009.

[70] Florian Mansmann, Fabian Fischer, Daniel A. Keim, and Stephen C. North. Visual support for analyzing network traffic and intrusion detection events using treemap and graph representations. In *Proc. of the Symposium on Computer Human Interaction for the Management of Information Technology*, CHiMiT '09, pages 19–28, New York, NY, USA, 2009. ACM.

[71] Yongzheng Wu and RolandH.C. Yap. Experiments with malware visualization. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, volume 7591, pages 123–133. Springer Berlin Heidelberg, 2013.

[72] Josh Saxe, David Mentis, and Chris Greamo. Visualization of shared system call sequence relationships in large malware corpora. In *Proc. of the Ninth International Symposium on Visualization for Cyber Security*, VizSec '12, pages 33–40, New York, NY, USA, 2012. ACM.

[73] D.A. Quist and L.M. Liebrock. Visualizing compiled executables for malware analysis. In *Proc. of 6th International Workshop on Visualization for Cyber Security (VizSec'09)*, pages 27–32, Oct 2009.

[74] Tian Ronghua, M.R. Islam, L. Batten, and S. Versteeg. Differentiating malware from cleanware using behavioural analysis. In *5th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 23–30, Oct 2010.

[75] Ahmadi Sami, Hossein Rahimi, and Babak Yadegari. Malware detection by behavioural sequential patterns. *Comput. Fraud and Secur.*, 2013(8):11 – 19, 2013.

[76] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Proc. of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[77] Blake Anderson, Curtis Storlie, and Terran Lane. Improving malware classification: Bridging the static/dynamic gap. In *Proc. of the 5th ACM Workshop on Security and Artificial Intelligence (AISec'12)*, pages 3–14, 2012.

[78] Bayer Ulrich, Comparetti Paolo-Milani, Hlauschek Clemens, Krügel Christopher, and Kirda Engin. Scalable, behavior–based malware clustering. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, 2009.

[79] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proc. of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 26–43, Berkeley, CA, USA, 2010. USENIX Association.

[80] Younghee Park, Douglas S. Reeves, and Mark Stamp. Deriving common malware behavior through graph clustering. *J. Comput. Secur.*, 39 Part B:419 – 430, 2013.

[81] Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. Jackstraws: Picking command and control connections from bot traffic. In *Proc. of the 20th USENIX Conference on Security*, SEC'11, pages 29–48, Berkeley, CA, USA, 2011. USENIX Association.

[82] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Proactive detection of computer worms using model checking. *IEEE Transactions on Dependable and Secure Computing*, 7(4):424–438, Oct 2010.

[83] Fu Song and Tayssir Touili. Efficient malware detection using model-checking. In *FM 2012: Formal Methods*, volume 7436, pages 418–433. Springer Berlin Heidelberg, 2012.

[84] Fu Song and Tayssir Touili. Pushdown model checking for malware detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214, pages 110–125. Springer Berlin Heidelberg, 2012.

[85] Fu Song and Tayssir Touili. Ltl model-checking for malware detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795, pages 416–431. Springer Berlin Heidelberg, 2013.

[86] Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. Abstraction-based malware analysis using rewriting and model checking. In *Proc. of 17th European Symposium on Research in Computer Security (ESORICS)*, volume 7459, pages 806–823. 2012.

[87] Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. Behavior abstraction in malware analysis. In *Proc. of Runtime Verification*, volume 6418, pages 168–182. 2010.

[88] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Proc. of the 10th International Conference on Recent Advances in Intrusion Detection (RAID' 07)*, pages 178–197, Berlin, Heidelberg, 2007. Springer-Verlag.

[89] M.Zubair Rafique and Juan Caballero. Firma: Malware clustering and network signature generation with mixed network behaviors. In *Research in Attacks, Intrusions, and Defenses*, volume 8145, pages 144–163. Springer Berlin Heidelberg, 2013.

[90] Perdisci Roberto, Ariu Davide, and Giacinto Giorgio. Scalable fine-grained behavioral clustering of HTTP-based malware. *Computer Networks*, 57(2):487–500, 2013.

[91] Oliver Sharma, Mark Girolami, and Joseph Sventek. Detecting worm variants using machine learning. In *Proceedings of the 2007 ACM CoNEXT Conference*, CoNEXT '07, pages 2:1–2:12, New York, NY, USA, 2007. ACM.

[92] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS' 11)*, pages 309–320, New York, NY, USA, 2011.

[93] Wang Xun, Yu Wei, Adam Champion, Fu Xinwen, and Xuan Dong. In *Proc. of Third International Conference on Security and Privacy in Communications Networks and the Workshops, (SecureComm 2007).*

[94] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Comput. Netw. ISDN Syst.*, 29(8-13):1157–1166, sep 1997.

[95] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *Proc. of the 23rd USENIX Conference on Security Symposium (USENIX' 14)*, pages 287–301, Berkeley, CA, USA, 2014. USENIX Association.

[96] Wei Yu, Xun Wang, P. Calyam, Dong Xuan, and Wei Zhao. Modeling and detection of camouflaging worm. *Dependable and Secure Computing, IEEE Transactions on*, 8(3):377–390, May 2011.

[97] Yu Wei, Zhang Nan, Fu Xinwen, and Zhao Wei. Self-disciplinary worms and countermeasures: Modeling and analysis. *IEEE Transactions on Parallel and Distributed Systems*, 21(10):1501–1514, 2010.

[98] KasperSky. Malware family tree. https://usa.kaspersky.com/internet-security-center/threats/, Jun. 2015.

[99] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. nether: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, pages 3:1–3:6, New York, NY, USA, 2011. ACM.

[100] Abhinav Srivastava, Andrea Lanzi, and Jonathon Giffin. System call api obfuscation (extended abstract). In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 421–422, Berlin, Heidelberg, 2008. Springer-Verlag.

[101] Huabiao Lu, Xiaofeng Wang, Baokang Zhao, Fei Wang, and Jinshu Su. Endmal: An anti-obfuscation and collaborative malware detection system using syscall sequences. *Mathematical and Computer Modelling*, 58(5–6):1140–1154, 2013.

[102] J00ru. Windows win32k.sys system call table, april 2014.

[103] Meinard Muller. Dynamic time warping. In *Information Retrieval for Music and Motion*, pages 69–84. Springer Berlin Heidelberg, 2007.

[104] Lalit Gupta, Dennis L. Molfese, Ravi Tammana, and Panagiotis G. Simos. Nonlinear alignment and averaging for estimating the evoked potential. *Biomedical Engineering, IEEE Transactions on*, 43(4):348–356, April 1996.

[105] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. On clustering validation techniques. *J. Intell. Inf. Syst.*, 17(2–3):107–145, dec. 2001.

[106] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, sep 1999.

[107] Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons, 2009.

[108] Hiroaki Sakoe and Seibi Chiba. Dynamic programming algorithm optimization for spoken word recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 26(1):43–49, 1978.

[109] Fumitada Itakura. Minimum prediction residual principle applied to speech recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 23(1):67–72, 1975.

[110] Alisa Shevchenko. Malicious code detection technologies. http://latam.kaspersky.com/sites/default/files/, 2008.

[111] Joanna Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction. http://www.hackerzvoice.net/ouah/Red_%20Pill.html.

[112] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In *Proc. of the Network and Distributed System Security Symposium, NDSS*, pages 1–16, San Diego, California, USA, 2010.

[113] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *Proc. of Recent Advances in Intrusion Detection (RAID'11)*, volume 6961, pages 338–357, 2011.

[114] Ming-Kung Sun, Mao-Jie Lin, Michael Chang, Chi-Sung Laih, and Hui-Tang Lin. Malware virtualization-resistant behavior detection. In *Proc. of the 17th International Conference on Parallel and Distributed Systems(IEEE)*, ICPADS '11, pages 912–917, Washington, DC, USA, 2011.

[115] Russinovich Mark, solomon David A, and Lonescu Alex. Windows internal part 2. 2012.

[116] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proc. of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 351–366. USENIX Association, 2009.

[117] Hornik Kurt, Stinchcombe Maxwell, and White Halbert. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989.

[118] Ou Guobin and Lu Murphey Yi. Multi-class pattern classification using neural networks. *J. Pattern Recognition*, 40(1):4 – 18, 2007.

[119] D L Chester. Why two hidden layers are better than one. In *Proc. International Joint Conference on Neural Networks, IJCNN-90-WASH-DC*, 1990.

[120] P. Vinod, Laxmi V, and Gaur M.S. REFORM:Relevant Feature for Malware analysis. In *Proc. of sixth IEEE international conference of security and Multimodality in Pervasive Environment (SMPE-2012)*, pages 26–29, Fukuoka Institute of technology (FIT), Fukuoka, Japan, 2012.

[121] Dima Stopel, Robert Moskovitch, Zvi Boger, Yuval Shahar, and Yuval Elovici. Using artificial neural networks to detect unknown computer worms. *Neural Computing and Applications*, 18(7):663–674, 2009.

[122] Nir Nissim, Robert Moskovitch, Lior Rokach, and Yuval Elovici. Detecting unknown computer worm activity via support vector machines and active learning. *Pattern Analysis and Applications*, 15(4):459–475, 2012.

[123] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, February 2006.

[124] Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh Charn Liu. Shadow attacks: automatically evading system-call-behavior based malware detection. *J. Comput. Virol.*, 8(1-2):1–13, 2012.

[125] Thomas Barabosch, Sebastian Eschweiler, and Elmar Gerhards-Padilla. Bee master: Detecting host-based code injection attacks. In *Proc. of 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'14)*, volume 8550, pages 235–254. 2014.

[126] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 272–280, 2003.

[127] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *Proc. of the 2003 IEEE Symposium on Security and Privacy, SP '03*, pages 62–, Washington, DC, USA, 2003. IEEE Computer Society.

[128] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *Proc. of the 14th conference on USENIX Security Symposium-Vol. 14*. USENIX Association, 2005.

[129] Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *Proc. of the 2008 ACM Symposium on Information, Computer and Communications Security (ASIACCS '08)*, pages 156–167, New York, NY, USA, 2008. ACM.

[130] M. Ramilli, M. Bishop, and Shining Sun. Multiprocess Malware. In *Proc. of 6th International Conference on Malicious and Unwanted Software (MALWARE'11)*, pages 8–13, Oct 2011.

[131] Yuede Ji, Yukun He, Dewei Zhu, Qiang Li, and Dong Guo. A mulitiprocess mechanism of evading behavior-based bot detection approaches. In *Information Security Practice and Experience*, volume 8434 of *Lecture Notes in Computer Science*, pages 75–89. Springer International Publishing, 2014.

[132] Naval Smita, Laxmi Vijay, Gaur Manoj, Raja Sachin, Rajarajan Muttukrishnan, and Conti Mauro. Environment-reactive malware behavior: Detection and categorization. In *Proc. of 7th International Workshop on Autonomous and Spontaneous Security (SETOP'14)*, pages 167–182, 2014.

[133] Guillermo Suarez-Tangil, Mauro Conti, JuanE. Tapiador, and Pedro Peris-Lopez. Detecting targeted smartphone malware with behavior-triggering stochastic models. In *Proc. of 19th European Symposium on Research in Computer Security (ESORICS'14)*, volume 8712, pages 183–201. 2014.

[134] C. Shannon and W. Weaver. The Mathematical Theory of Communication. 1963.

[135] Cewei Cui, Zhe Dang, and Thomas R. Fischer. Typical paths of a graph. *J. Fundam. Inf.*, 110(1-4):95–109, Jan. 2011.

[136] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory 2nd Edition (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, July 2006.

[137] J. R. Norris. *Markov Chains*. Cambridge University Press, 1998.

[138] Michael J. Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Comput. Surv.*, 16(3):319–348, sep 1984.

[139] Stefan Edelkamp and Richard E. Korf. The branching factor of regular search spaces. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence (AAAI'98 / IAAI'98)*, pages 299–304, 1998.

[140] S. Maji, A.C. Berg, and J. Malik. Classification using intersection kernel support vector machines is efficient. In *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR'08)*, pages 1–8, June 2008.

[141] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[142] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(8):832–844, Aug 1998.

[143] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oct. 2003.

[144] Zhiyong Shan and Xin Wang. Growing grapes in your computer to defend against malware. *IEEE Trans. on Inf. Forensics and Secur.*, 9(2):196–207, Feb 2014.

[145] P. Vinod, V. Laxmi, M.S. Gaur, and G. Chauhan. Momentum: Metamorphic malware exploration techniques using MSA signatures. In *Proc. of International Conference on Innovations in Information Technology (IIT'12)*, pages 232–237, March 2012.

[146] Daniel Quist, Lorie Liebrock, and Joshua Neil. Improving antivirus accuracy with hypervisor assisted analysis. *J. Comput. Virol.*, 7(2):121–131, may 2011.

[147] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Lett.*, 27(8):861–874, 2006.

[148] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, jan 1976.

[149] Horst Bunke, P. Foggia, C. Guidobaldi, and M. Vento. Graph clustering using the weighted minimum common supergraph. In *Graph Based Representations in Pattern Recognition*, volume 2726 of *Lecture Notes in Computer Science*, pages 235–246. Springer Berlin Heidelberg, 2003.

[150] Microsoft. Kernel object. https://msdn.microsoft.com/en-us/library/, Feb. 2015.

[151] Boost-Software. Graph library. http://sourceforge.net/projects/boost/files/boost/, Feb. 2015.

[152] Krzysztof Kaczmarski, Piotr Przymus, and Pawe Rzazewski. Improving high-performance GPU graph traversal with compression. In *New Trends in Database and Information Systems II*, volume 312 of *Advances in Intelligent Systems and Computing*, pages 201–214. Springer International Publishing, 2015.

[153] Instant messanger worm: Osx.leap.a. http://www.securelist.com/en/descriptions/147387/IM-Worm.OSX.Leap.a, Jun. 2014.

[154] NVIDIA. *NVIDIA CUDA Programming Guide 2.0.* 2008.

[155] Lodovico Marziale, Golden G. Richard, III, and Vassil Roussev. Massive threading: Using gpus to increase the performance of digital forensics tools. *Digit. Investig.*, 4:73–81, sep 2007.

[156] Ali Akoglu and Gregory M. Striemer. Scalable and highly parallel implementation of smith-waterman on graphics processing unit using cuda. *Cluster Computing*, 12(3):341–352, September 2009.

[157] NVIDIA CUDA. NVIDIA corporation: NVIDIA CUDA compute unified device architecture programming guide. http://www.nvidia.com/object/cuda-develop.html, 2014.

[158] TESLA. Nvidia Tesla C2075 Companion Processor. http://www.nvidia.in/page/tesla-product-literature.html, 2013.